Chapitre 2

Les automates à états finis (AEF)

Les AEF sont les plus simples des machines d'analyse de langages car ils ne comportent pas de mémoire. Par conséquent, les langages acceptés par ce type d'automates sont les plus simples des quatre classes de Chomsky, à savoir les langages réguliers (type 3). Par ailleurs, les automates à états finis peuvent être utilisés pour modéliser plusieurs problèmes dont la solution n'est pas très évidente. La série de TD propose quelques exercices dans ce sens.

1. Généralités sur les AEF

Définition 2.1 : Un automate à états finis est machine abstraite définie par le quintuplet (X, Q, q_0, F, δ) tel que :

- X est l'ensemble des symboles formant les mots en entrée (l'alphabet des mots à analyser) ;
- Q est l'ensemble des états possibles ;
- q_0 est l'état initial ($q_0 \in Q$) ;
- F est l'ensemble des états finaux ($F \subseteq Q$) ou encore les états d'acceptation ;
- δ est une fonction de transition qui permet de passer d'un état à un autre selon l'entrée en cours :

  $\delta : Q \times (X \cup \{\varepsilon\}) \mapsto 2^Q$
  $\delta(q_i, a) = \{q_{j_1}, q_{j_2}, ..., q_{j_k}\}$ ou $\emptyset$ ($\emptyset$ signifie que la configuration n'est pas prise en charge ou encore que la transition n'existe pas)

Un mot est accepté par un AEF si, après avoir lu tout le mot, l'automate se trouve dans un état final ($q_f \in F$). En d'autres termes, lorsqu'un AEF n'a plus de transition à exécuter, un mot est rejeté dans deux cas :

- L'automate est dans l'état q_i, l'entrée courante étant a et la transition $\delta(q_i, a)$ n'existe pas (on n'arrive pas à lire tout le mot) ;
- L'automate arrive à lire tout le mot mais l'état de *sortie* n'est pas un état final.

Un AEF A est donc un *séparateur* (ou classifieur) des mots de X^* en deux parties : l'ensemble des mots acceptés par l'automate (notons-le par $L(A)$) et le reste des mots ($X^* - L(A)$).

Exemple 2.1 : un automate à états finis

Soit l'AEF défini par $(\{a, b\}, \{0, 1\}, 0, \{1\}, \delta)$ tel que :

$$\delta(0, a) = \{0\} \quad \delta(0, b) = \{1\}$$
$$\delta(1, a) = \emptyset \quad \delta(1, b) = \{1\}$$

Voici comment se fait l'analyse de différents mots :

— Le mot aab : on a la suite des configurations suivantes : $(0, a) \to (0, a) \to (0, b) \to (1, \varepsilon)$. Notons que ε dans la dernière configuration signifie que l'on est arrivé à la fin du mot. aab est accepté car l'état de sortie 1 est final et le mot a été entièrement lu. Schématiquement, nous pouvons imaginer le fonctionnement de l'automate selon la figure 2.2 :

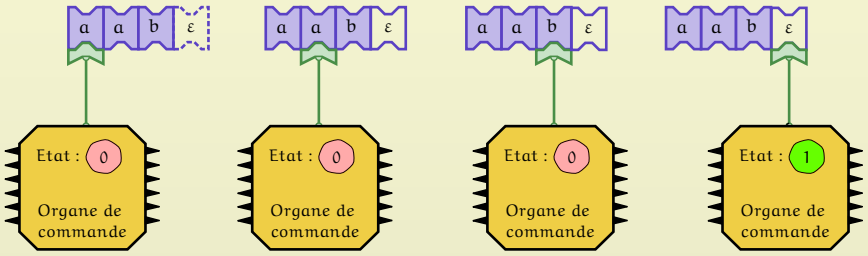

FIGURE 2.1 – Représentation imaginaire de l'analyse du mot aab

— Le mot ε : la seule configuration est $(0, \varepsilon)$. Le mot est rejeté car 0 n'est pas un état final même s'il a été entièrement lu. Schématiquement, nous pouvons imaginer le fonctionnement de l'automate selon la figure 2.2 :

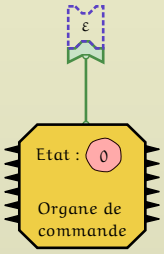

FIGURE 2.2 – Représentation imaginaire de l'analyse du mot ε

— Le mot ba : on a la suite des configurations suivantes : $(0, b) \to (1, a)$. Le mot n'est pas accepté car il n'a pas été entièrement lu (même si l'état de sortie est bien final). Schématiquement, nous pouvons imaginer le fonctionnement de l'automate selon la figure 2.3 :

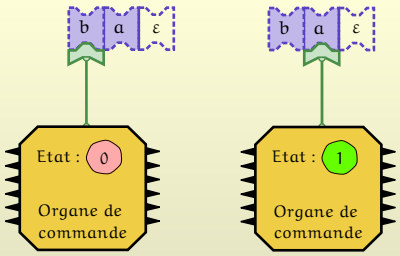

FIGURE 2.3 – Représentation imaginaire de l'analyse du mot ba

On peut facilement voir que le langage accepté par cet automate est $a^m b^n (m \geqslant 0, n > 0)$.

Un AEF peut être représenté de deux manières : soit par une table définissant la fonction de transition soit par un *graphe orienté*.

1..1 Représentation par table

La table possède autant de lignes qu'il y a d'états dans l'automate de telle sorte que chaque ligne corresponde à un état. Les colonnes correspondent aux différents symboles de l'alphabet. Si l'automate est dans l'état i et que le symbole j est le prochain à lire, alors l'entrée (i, j) de la table donne l'état auquel l'automate passera après avoir lu j. Notons que la définition par table n'est pas suffisante pour définir l'AEF entièrement étant donné que la table ne donne ni l'état initial ni les états finaux.

Exemple 2.2 : représentation tabulaire

L'automate précédent est représenté comme suit :

État	a	b
0	0	1
1	-	1

1..2 Représentation graphique

La représentation graphique consiste à représenter l'automate par un graphe orienté. Chaque état de l'automate est schématisé par un rond (ou sommet). Si la transition $\delta(q_i, a) = \{q_j\}$ est définie, alors on raccorde le sommet q_i au sommet q_j par un arc décoré par le symbole a. L'état initial est désigné par une flèche entrante au sommet correspondant tandis que les états finaux sont marqués par un double rond. Le schéma de la figure 2.4 reprend l'automate précédent.

Exemple 2.3 : représentation graphique

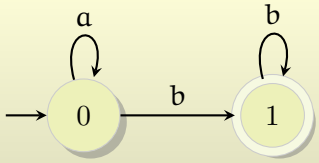

FIGURE 2.4 – L'automate acceptant $a^m b^n (m \geqslant 0, n > 0)$

Lorsqu'il y a plusieurs symboles $a_1, ...a_k$ tels que $(q_i, a_l) = \{q_j\}(l = 1..k)$ alors on se permet de décorer l'arc (q_i, q_j) par l'ensemble $a_1, ..., a_k$.

La définition d'un AEF en tant que graphe a un avantage, car elle permet de définir autrement l'acceptation d'un mot. Mais, on préférera la représentation tabulaire dans le cas où l'on fait des calculs sur les automates.

Définition 2.2 : Soit l'AEF $A = (X, Q, q_0, F, \delta)$. On appelle *chemin* toute séquence d'états $s_0, s_1, s_2, ..., s_n$ tels que $\forall i : s_i \in Q$ et $\forall i, \exists a_i \in X : s_{i+1} \in \delta(s_i, a_i)$. Son mot correspondant est alors : $a_0 a_1 a_2 ... a_{n-1}$.

Dans l'automate précédent, on peut, entre autres, définir les chemins suivants :

— Le chemin $0, 0, 0, 0$ correspondant au mot aaa ;
— Le chemin $0, 0, 0, 1$ correspondant au mot aab ;
— Le chemin $0, 1, 1, 1, 1, 1$ correspondant au mot $abbbb$;
— Le chemin $1, 1, 1, 1$ correspondant au mot bbb.

On peut maintenant donner une deuxième définition à l'acceptation d'un mot.

Définition 2.3 : Soit l'AEF $A = (X, Q, q_0, F, \delta)$. Un mot w est accepté par A s'il existe un chemin de A correspondant à w tel que le premier état est q_0 et le dernier état est final.

Ainsi, dans les mots précédents, le mot aaa est rejeté alors que aab et $abbbb$ sont acceptés. Le mot ba est, à son tour, rejeté car il ne correspond à aucun chemin de l'automate considéré. Le mot bbb est accepté aussi car on peut lui trouver un chemin partant de l'état initial et se terminant par un état final.

2. Les automates et le déterminisme

2..1 Notion de déterminisme

Un programme informatique est, en général, déterministe : c'est-à-dire que l'on connaît avec précision son comportement dans les différentes situations possibles. Ceci rend l'analyse des programmes plus simples. Malheureusement, tous les programmes ne peuvent pas être de la sorte car on peut parfois ignorer les conditions d'exécution (état d'une machine distante par exemple). Il est à noter que l'on préfère toujours une version déterministe d'un programme plutôt qu'une version

non-déterministe (le non-déterminisme peut induire des coûts importants surtout pour l'analyse et la maintenance).

Un AEF, étant une forme spéciale d'un programme informatique, agit similairement. En d'autres mots, étant donné un mot à analyser, on souhaite connaître avec certitude le prochain état de l'automate. Ceci revient à dire que si l'automate est dans l'état q_i et que l'entrée courante est a alors il existe au plus seul un état q_j tel que $\delta(q_i, a) = \{q_j\}$. Le cas échéant, on dit que l'automate est déterministe parce qu'il sait **déterminer** le prochain état à tout moment. Dans le cas inverse, l'automate doit choisir une action et la tester à terme, si l'acceptation n'est pas possible l'automate doit tester les autres éventualités.

Le non-déterminisme peut également provenir des transitions (arcs). En effet, rien n'interdit dans la définition des AEF d'avoir des ε-transitions, c'est-à-dire des transitions décorées avec ε. L'existence d'une ε-transition entre les états q_i et q_j signifie que l'on n'a pas besoin de lire un symbole [1] pour passer de q_i vers q_j (attention ! l'inverse n'est pas possible).

Voyons maintenant une définition formelle de la notion du déterminisme pour les AEF.

Définition 2.4 : Un AEF (X, Q, q_0, F, δ) est dit déterministe si les deux conditions sont vérifiées :

— $\forall q_i \in Q, \forall a \in X$, il existe au plus un état q_j tel que $\delta(q_i, a) = \{q_j\}$;

— L'automate ne comporte pas de ε-transitions.

Exemple 2.4 : automate non-déterministe sans ε-transition

Soit le langage des mots définis sur $\{a, b\}$ possédant le facteur ab. La construction d'un AEF non-déterministe est facile. La table suivante donne la fonction de transition (l'état initial est l'état 0 et l'état 2 est final).

État	a	b
0	0,1	0
1	-	2
2	2	2

La figure 2.5 reprend le même automate :

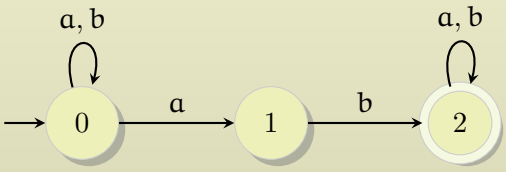

FIGURE 2.5 – L'automate des mots sur $\{a, b\}$ contenant le facteur ab

Exemple 2.5 : automate non-déterministe avec des ε-transitions

L'automate donné par la figure 2.6 accepte le même langage que le précédant mais en utilisant des ε-transitions.

1. Par abus de langage, on dit qu'on lit ε

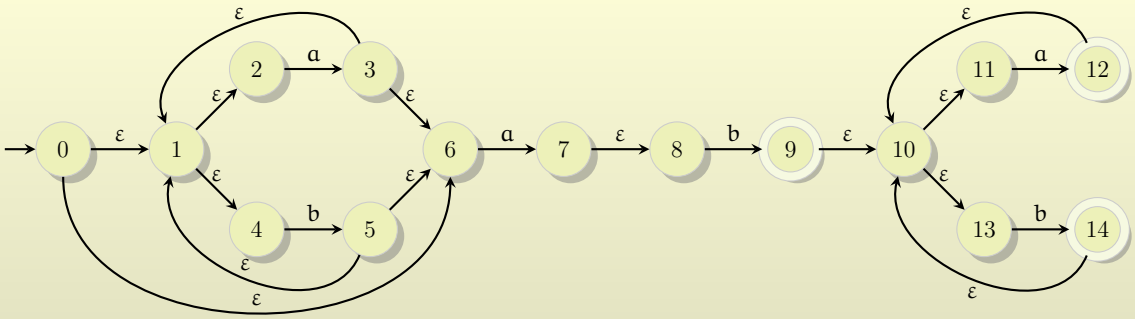

FIGURE 2.6 – L'automate acceptant les mots sur $\{a, b\}$ contenant le facteur ab (avec des ε-transitions)

2..2 Déterminisation d'un automate à états fini

Si on analyse le mot aba par l'AEF de la figure 2.5, on aura la première configuration $(0, a)$ suite à laquelle deux états sont possibles : soit 0 ou 1. À ce stade et vu que l'on ne connaît pas la suite du mot, on est incapable de déterminer quel état permet-il d'accepter le mot. Étant donné que dans un AEF, la tête L/E est censée bouger dans un seul sens, on se trouve dans la situation suivante : chaque fois que l'on n'arrive pas à déterminer quel est le prochain état, on en choisit un arbitrairement puis on continue l'analyse. Si cette dernière permet d'accepter le mot alors le choix était bon, sinon on revient au point de non-déterminisme pour choisir un autre état et continuer l'analyse. Pour le mot aba, cela se passe comme suit :

$$
\begin{array}{lllllll}
(0,a) & \to & (0,b) & \to & (0,a) & \to & (0,\varepsilon)\ \textbf{échec} \\
 & \downarrow & & & & \searrow & (1,\varepsilon)\ \textbf{échec} \\
 & \downarrow & & & & & \\
 & \downarrow & & & & & \\
 & \to & (1,b) & \to & (2,a) & \to & (2,\varepsilon)\ \textbf{succès}
\end{array}
$$

Cet exemple illustre bien le problème des automates non-déterministes. En effet, pour un mot de longueur n, la complexité de l'analyse peut, au pire cas, être $f(n) = a^n$ $(a > 1)$ avant de dire si le mot est accepté ou non. Pour $a = 2$ et un mot de 100 symboles (ce qui n'est pas important), le nombre de configurations peut atteindre l'ordre de 2^{100}, ce qui équivaut à 10^{30} configurations. L'analyse peut alors durer des milliers de siècles même sur l'ordinateur le plus rapide au monde. En revanche, si l'AEF est déterministe, le nombre de configurations est simplement $f(n) = n + 1$ seulement.

Nous pouvons alors déduire, dans un premier temps, que les automates non-déterministes ne sont guère intéressants vu qu'ils peuvent générer un coût très élevé lors de l'analyse, mais ce n'est pas le cas en réalité. Le plus souvent, il se trouve qu'il est beaucoup plus simple de concevoir un AEF non-déterministe que de construire un qui soit déterministe. De plus, comme on le verra dans le prochain chapitre, les langages réguliers sont souvent notés en utilisant *les expressions régulières*. Un algorithme (dit de Thompson) permet alors de construire l'AEF du langage à partir des expressions régulières mais l'AEF est presque toujours non-déterministe avec beaucoup d'ε-transitions.

Heureusement, lorsqu'il s'agit des langages réguliers, un théorème nous sera d'une grande utilité

car il établit l'équivalence entre les automates déterministes et ceux non-déterministes (la démonstration de ce théorème sort du cadre de ce cours).

Théorème 2.1

(dit de Rabin et Scott) Tout langage accepté par un AEF non-déterministe peut également également être accepté par un AEF déterministe.

Une conséquence très importante de ce théorème peut déjà être citée (en réalité, elle découle plutôt de la démonstration de ce théorème) :

Proposition 2.1

Tout AEF non-déterministe peut être transformé en un AEF déterministe.

Ce résultat établit que si l'on veut construire l'automate à états fini déterministe qui accepte les mots d'un certain langage, alors on peut commencer par trouver un AEF non-déterministe (ce qui est plus facile). Il suffit de le transformer, après, pour obtenir un automate à états finis déterministe.

2..3 Déterminisation d'un AEF sans ε-transition

En réalité, l'algorithme de déterminisation d'un AEF est général, c'est-à-dire qu'il fonctionne dans tous les cas (qu'il y ait des ε-transitions ou non). Cependant, il est plus facile de considérer cet algorithme sans les ε-transitions. Dans cette section, on suppose que l'on a un AEF A ne comportant aucune ε-transition.

Le principe de l'algorithme est de considérer des ensembles d'états plutôt que de simples états (dans l'algorithme suivant, chaque ensemble d'états représente un état futur de l'automate déterministe).

Algorithme de déterminisation d'un AEF sans les ε-transitions

1. Partir de l'état initial $E^{(0)} = \{q_0\}$ (c'est l'état initial du nouvel automate)
2. Construire $E^{(1)}$ (considéré comme étant un nouvel état) l'ensemble des états obtenus à partir de $E^{(0)}$ par un symbole a : $E^{(1)} = \bigcup_{q' \in E^{(0)}} \delta(q', a)$
3. Recommencer l'étape 2 pour tous les symboles possibles et pour chaque nouvel ensemble $E^{(i)}$: $E^{(i)} = \bigcup_{q' \in E^{(i-1)}} \delta(q', a)$
4. Tous les ensembles contenant au moins un état final du premier automate sont des états finaux
5. Renuméroter les états en tant qu'états simples

Pour illustrer cet algorithme, nous allons l'appliquer à l'automate donné par la figure 2.5. La table 2.1 donne les étapes d'application de l'algorithme (les états en gras sont des états finaux).

La figure 2.7 donne l'automate obtenu (remarquons qu'il n'est pas optimal). Cet automate n'est pas évident à trouver mais grâce à l'algorithme de déterminisation, on peut le construire automati-

État	a	b
0	0,1	0
0,1	0,1	0,2
0,2	0,1,2	0,2
0,1,2	0,1,2	0,2

$\Longrightarrow$

État	a	b
0′	1′	0′
1′	1′	2′
2′	3′	2′
3′	3′	2′

TABLE 2.1 – Déterminisation d'un AEF sans ε-transition

quement.

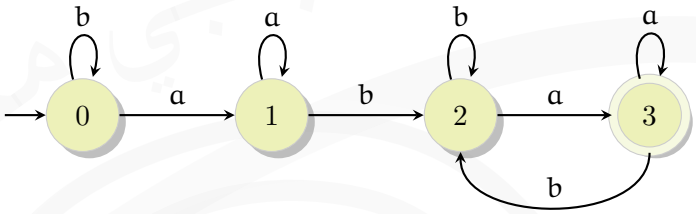

FIGURE 2.7 – L'automate déterministe qui accepte les mots ayant le facteur ab

2..4 Déterminisation avec les ε-transitions

La déterminisation d'un AEF contenant au moins une ε-transition est un peu plus compliquée puisqu'elle fait appel à la notion de l'ε-fermeture d'un ensemble d'états. Nous commençons donc par donner sa définition.

Définition 2.5 : Soit E un ensemble d'états. On appelle ε-fermeture de E l'ensemble des états incluant, en plus de ceux de E, tous les états accessibles depuis les états de E par un chemin correspondant au mot ε.

Algorithme de calcul de ε-fermeture ϕ d'un ensemble E

```
1 ϕ ← E
2 répéter
3 |   pour tout état q de ϕ faire
4 |   |   ϕ ← ϕ ∪ δ(q, ε)
5 jusqu'à ce que l'ensemble ϕ ne change plus;
```

Exemple 2.6 : calcul des ε-fermetures

Considérons l'automate donné par la figure 2.6, calculons un ensemble d'ε-fermetures :

— ε-fermeture($\{0\}$) = $\{0, 1, 2, 4, 6\}$

— ε-fermeture($\{1, 2\}$) = $\{1, 2, 4\}$

— ε-fermeture($\{3\}$) = $\{1, 2, 3, 4, 6\}$

L'algorithme de déterminisation dans le cas général est le suivant (la fonction ε_f représente l'ε-fermeture) :

Algorithme de déterminisation d'un AEF avec des ε-transitions

1 Partir de l'état initial $E^{(0)} = \varepsilon_f(\{q_0\})$ (c'est l'état initial du nouvel automate)

2 Construire $E^{(1)}$ l'ensemble des états obtenus à partir de $E^{(0)}$ par un symbole a : $E^{(1)} = \varepsilon_f\left(\bigcup_{q' \in E^{(0)}} \delta(q', a)\right)$

3 Recommencer l'étape 2 pour tous les symboles possibles et pour chaque nouvel ensemble $E^{(i)}$: $E^{(i)} = \varepsilon_f\left(\bigcup_{q' \in E^{(i-1)}} \delta(q', a)\right)$

4 Tous les ensembles contenant au moins un état final du premier automate sont des états finaux

5 Renuméroter les états en tant qu'états simples

Exemple 2.7 : déterminisation avec des ε-transitions

Appliquons maintenant ce dernier algorithme à l'automate de la figure 2.6.

État	a	b
0,1,2,4,6	1,2,3,4,6,7,8	1,2,4,5,6
1,2,3,4,6,7,8	1,2,3,4,6,7,8	1,2,4,5,6,9,10,11,13
1,2,4,5,6	1,2,3,4,6,7,8	1,2,4,5,6
1,2,4,5,6,9,10,11,13	1,2,3,4,6,7,8,10,11,12,13	1,2,4,5,6,10,11,13,14
1,2,3,4,6,7,8,10,11,12,13	1,2,3,4,6,7,8,10,11,12,13	1,2,4,5,6,9,10,11,13,14
1,2,4,5,6,10,11,13,14	1,2,3,4,6,7,8,10,11,12,13	1,2,4,5,6,10,11,13,14
1,2,4,5,6,9,10,11,13,14	1,2,3,4,6,7,8,10,11,12,13	1,2,4,5,6,10,11,13,14

ce qui produit l'automate suivant (l'état initial est 0, les états finaux sont : 3, 4, 5 et 6) :

État	a	b
0	1	2
1	1	3
2	1	2
3	4	5
4	4	6
5	4	5
6	4	5

3. Minimisation d'un AEF déterministe

Si on prend deux automates déterministes acceptant le même langage, le nombre de configurations nécessaires pour analyser un mot ne dépend que de la taille de ce mot. A priori, on peut dire alors que le nombre d'états d'un automate importe peu pour déterminer le coût d'analyse d'un mot. Mais, ce n'est pas tout à fait correct.

D'abord signalons que l'opération de déterminisation a la fâcheuse tendance de produire beaucoup d'états. Pour un AEF non-déterministe comportant n états, la procédure de déterminisation peut produire jusqu'à $2^n - 1$ états possibles (ce qui est un nombre considérable). Or, stocker et explorer une petite structure mémoire n'est pas comme explorer une grande structure. Pour n assez grand, il se peut que la représentation mémoire d'un AEF nécessite plusieurs pages de mémoires, allant jusqu'à causer des défauts de pages (ce qui augmente le temple d'analyse). Sur un autre plan, dans les systèmes embarqués (où la dimension énergie est des plus capitales), nous avons tout l'intérêt à réduire le nombre de cellules mémoires occupées afin de minimiser la puissance électrique nécessaire pour maintenir les données.

En conclusion, il est tout à fait justifiable de vouloir s'intéresser à la réduction du nombre d'états nécessaires pour accepter un langage. D'ailleurs, on s'intéressera dans ce qui suit au calcul du nombre minimal d'états nécessaires à l'analyse d'un langage donné. Il est à noter, enfin, que si on peut trouver une multitude d'automates déterministes pour analyser le même langage, on ne peut trouver qu'un seul automate déterministe et minimal (avec un nombre minimal d'états) acceptant le même langage.

La minimisation s'effectue en éliminant les états dits inaccessibles et en *confondant* (ou fusionnant) les états acceptant le même langage.

3..1 Les états inaccessibles

Définition 2.6 : Un état est dit inaccessible s'il n'existe aucun chemin permettant de l'atteindre à partir de l'état initial.

D'après la définition, les états inaccessibles sont improductifs (il existe néanmoins des états improductifs qui ne sont pas inaccessibles), c'est-à-dire qu'ils ne participeront jamais à l'acceptation d'un mot. Ainsi, la première étape de minimisation d'un AEF consiste à éliminer ces états. L'étudiant peut, en guise d'exercice, écrire l'algorithme qui permet de trouver les états inaccessibles d'un AEF (indice : pensez à un algorithme de marquage d'un graphe).

3..2 Les états β-équivalents

Définition 2.7 : Deux états q_i et q_j sont dits β-équivalents s'ils permettent d'atteindre les états finaux à travers les mêmes mots. On écrit alors : $q_i \ \beta \ q_j$.

Par le même mot, on entend que l'on lit la même séquence de symboles pour atteindre un état final à partir de q_i et q_j. Par conséquent, ces états acceptent le même langage. La figure 2.8 montre un exemple d'états β-équivalents car l'état q_i atteint les états finaux via les mots a et b, de même pour l'état q_j. L'algorithme de minimisation consiste donc à fusionner simplement ces états pour n'en faire qu'un.

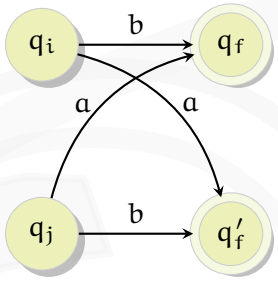

FIGURE 2.8 – Exemple d'états β-équivalents

Remarque : La relation β-équivalence est une relation d'équivalence. De plus, si q_i et q_j sont β-équivalents, alors $\forall x \in X$ (X étant l'alphabet), $\delta(q_i, x)$ et $\delta(q_j, x)$ sont également β-équivalents (puisque q_i et q_j acceptent le même langage). La relation β-équivalence est dite une relation de congruence.

Remarque : Le nombre de classes d'équivalence de la relation β-équivalence est égal au nombre des états de l'automate minimal car les état de chaque classe d'équivalence acceptent le même langage (ils seront fusionnés).

3..3 Minimiser un AEF

La méthode de réduction d'un AEF est la suivante :

1. *Nettoyer* l'automate en éliminant les états inaccessibles (cette étape n'est pas nécessaire si l'automate à minimiser a été obtenu par la procédure de déterminisation) ;
2. Regrouper les états congruents (appartenant à la même classe d'équivalence).

Dans l'algorithme suivant, chaque classe d'équivalence représentera un état dans l'automate minimal.

Algorithme de fusion des états β-équivalents d'un AEF $A = (X, Q, q_0, F, \delta)$

1 Créer deux classes d'états : $A = F$, $B = Q - F$

2 S'il existe un symbole a et deux états q_i et q_j d'une même classe tel que $\delta(q_i, a)$ et $\delta(q_j, a)$ n'appartiennent pas à la même classe, alors créer une nouvelle classe et séparer q_i et q_j. On laisse dans la même classe tous les états qui donnent des états appartenant à la même classe

3 Recommencer l'étape 2 jusqu'à ce qu'il n'y ait plus d'états à séparer

Les paramètres de l'automate minimal sont, alors, les suivants :

— Chaque classe d'équivalence est un état de l'automate minimal ;
— La classe qui contient l'ancien état initial représente l'état initial de l'automate minimal ;
— Toute classe contenant un état final devient un état final ;
— La fonction de transition est définie comme suit : soient A une classe d'équivalence obtenue, a un symbole de l'alphabet et q_i un état $q_i \in A$ tel que $\delta(q_i, a)$ est définie. La transition $\delta(A, a)$ est égale à la classe B qui contient l'état q_j tel que $\delta(q_i, a) = q_j$.

Exemple 2.8 : étapes pratiques de la minimisation

Soit à minimiser l'automate suivant (les états finaux sont les états 1 et 2 tandis que l'état 1 est initial) :

État	a	b
1	2	5
2	2	4
3	3	2
4	5	3
5	4	6
6	6	1
7	5	7

La première étape consiste à éliminer les états inaccessibles, il s'agit juste de l'état 7.

Pour le regroupement des états congruents, nous appliquons l'algorithme suivant :

1. Définir deux ensembles d'états : A : ensemble des états finaux, B : ensemble des états non finaux (si tous les états sont finaux, alors commencer avec un seul ensemble).
2. Procéder par itérations en vérifiant tous les ensembles d'états contenant plus qu'un état comme suit :
   (a) Pour un ensemble d'états E, pour chaque symbole a de l'alphabet, vérifier qu'il n'existe pas deux états $q, q' \in E$ tel que $\delta(q, a) \in J$, $\delta(q', a) \in J'$ et $J \neq J'$.
   (b) Si un tel symbole existe (on dit que l'ensemble n'est pas cohérent), procéder à la découpe de l'ensemble E comme dans l'algorithme précédent.
   (c) Sinon on passe au prochain symbole.
3. Si aucune modification n'a eu lieu au cours d'une itération, alors arrêter l'algorithme, sinon refaire une nouvelle itération.

Pour l'automate précédent, les étapes de détermination des classes d'équivalence sont les suivantes :

1. On crée d'abord les deux ensembles : $A = \{1, 2\}, B = \{3, 4, 5, 6\}$.
2. Itération 1 :
   (a) Pour l'ensemble A et le symbole a : $\delta(1, a) = 2 \in A$ et $\delta(2, a) = 2 \in A$, OK.
   (b) Pour l'ensemble A et le symbole b : $\delta(1, b) = 5 \in B$ et $\delta(2, b) = 4 \in B$, OK. L'ensemble A est cohérent pour le moment.
   (c) Pour l'ensemble B et le symbole a : $\delta(3, a) = 3 \in B$, $\delta(4, a) = 5 \in B$, $\delta(5, a) = 4 \in B$ et $\delta(6, a) = 6 \in B$. OK.
   (d) Pour l'ensemble B et le symbole b : $\delta(3, b) = 2 \in A$, $\delta(4, b) = 3 \in B$, $\delta(5, b) = 6 \in B$ et $\delta(6, b) = 1 \in A$. La classe doit être éclatée, alors on crée un nouvel ensemble $C = \{3, 6\}$ ce qui laisse l'ensemble $B = \{4, 5\}$.
3. Itération 2 (comme il y a eu des modifications au cours de la première itération).
   (a) On constate que l'ensemble A est cohérent (à vous de faire les vérifications).
   (b) On constate que l'ensemble B est cohérent (à vous de faire les vérifications).
   (c) On constate que l'ensemble C est cohérent (à vous de faire les vérifications).
   (d) On arrête alors l'algorithme car il n'y a plus de modifications possibles.

Le nouvel automate est donc le suivant (l'état initial est A) :

État	a	b
A	A	C
B	B	A
C	C	B

Remarque : L'automate obtenu est minimal et est unique, il ne peut plus être réduit. Si après la réduction d'un automate on obtient le même, alors cela signifie qu'il est déjà minimal. Par ailleurs, l'automate déterministe et minimal d'un AEF caractérise le langage accepté. En d'autres mots, si $L = L'$ (tous les deux étant réguliers), alors ils ont le même automate déterministe et minimal (on peut être amené à éliminer certains autres états non nécessaires néanmoins).

4. Opérations sur les automates

Notons d'abord que les opérations suivantes ne sont pas spécifiques aux AEF, elles peuvent plus ou moins être étendues à tout automate. Cependant, la complexité de ces opérations augmente inversement avec le type du langage.

4..1 Le complément

Soit A un automate <u>déterministe</u> défini par le quintuplet (X, Q, q_0, F, δ) acceptant le langage L. L'automate acceptant le langage inverse (c'est-à-dire $X^* - L$) est défini par le quintuplet $(X, Q, q_0, Q - F, \delta)$ (en d'autres termes, il suffit de changer les états finaux en états non finaux et vice-versa).

Cependant, cette propriété ne fonctionne que si l'automate est *complet* : la fonction δ est complètement définie pour toute paire $(q, a) \in Q \times X$ comme le montre les exemples suivants.

Exemple 2.9 : automate du langage complémentaire

Soit le langage des mots définis sur l'alphabet $\{a, b, c\}$ contenant le facteur ab. Il s'agit ici d'un automate complet, on peut, donc, lui appliquer la propriété précédente pour trouver l'automate des mots qui ne contiennent pas le facteur ab (notons que l'état 2 du deuxième automate correspond à une sorte d'état d'erreur auquel l'automate se branche lorsqu'il détecte le facteur ab. On dit que l'état 2 est *improductif*, étant donné qu'il ne peut pas générer de mots).

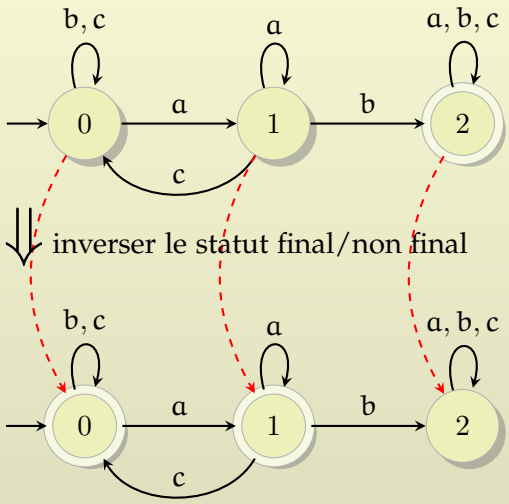

FIGURE 2.9 – Comment obtenir l'automate du langage complémentaire (d'un automate complet). Attention, les flèches rouges ne font pas partie de l'automate, elles illustrent seulement le changement de statut.

Soit maintenant le langage des mots définis sur $\{a, b, c\}$ contenant exactement deux a (figure 2.10). L'automate n'est pas complet, donc, on ne peut pas appliquer la propriété précédente à moins de le transformer en un automate complet. Pour le faire, il suffit de rajouter un état non final E (on le désignera comme un état d'erreur) tel que $\delta(2, a) = E$.

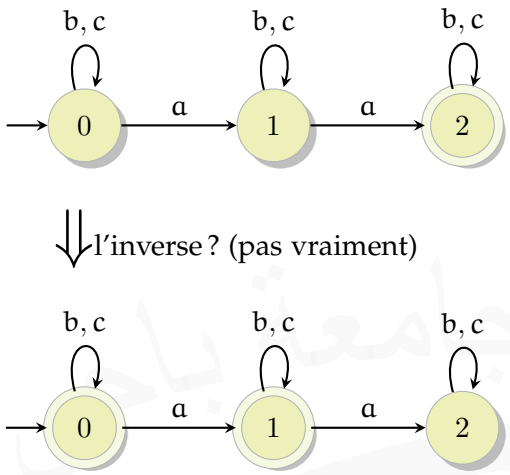

FIGURE 2.10 – Si l'automate n'est pas complet, on ne peut pas obtenir l'automate du langage inverse. L'automate obtenu accepte les mots contenant au plus 1 a.

Considérons à la fin l'automate de la figure 2.11. Les deux automates acceptant le mot a, ce qui signifie que les deux automates n'acceptent pas des langages complémentaires.

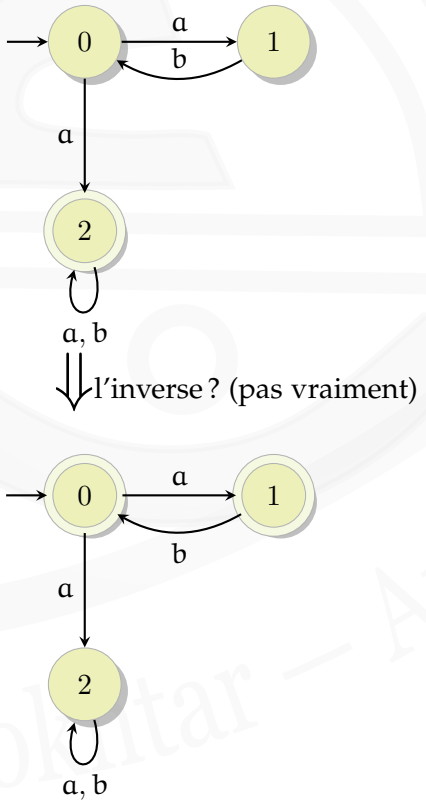

FIGURE 2.11 – Si l'automate n'est pas déterministe, on ne peut pas trouver l'automate du langage complémentaire.

Remarque : Le fait qu'un AEF ne soit pas déterministe ne représente pas un obstacle pour trouver l'automate du langage inverse. En effet, on pourra toujours le construire en procédant à une transformation. *Laquelle ?*

La construction d'une version complète d'un AEF (qui doit être déterministe) se fait comme suit :

1. Rajouter un nouvel état N ;
2. Les transitions manquantes mènent dorénavant vers N ;
3. Toute transition sortante de N revient au même état.

4..2 L'opération d'entrelacement

Nous verrons ici le cas simple de l'entrelacement d'un langage avec un symbole (voir la figure 2.12). Soit L un langage accepté par un AEF $A = (X, Q, q_0, F, \delta)$. On note par $A' = (X, Q', q'_0, F', \delta')$ une copie de l'automate A avec : Q' l'ensemble des états de Q auxquels on rajoute prime ($'$), par q'_0 l'état initial de A auquel on rajoute $'$, par F' l'ensemble des états de F auxquels on rajoute $'$ et par δ' la fonction de transition $\delta(q'_i, a) = q'_j$ si $\delta(q_i, a) = q_j$. L'automate acceptant le langage $L \sqcup \alpha$ (α est un seul symbole) est donné par $A'' = (X + \{\alpha\}, Q \cup Q', q_0, F', \delta'')$ tel que :

— $\delta''(q_i, x) = \delta(q_i, x)$, $q_i \in Q$;
— $\delta''(q'_i, x) = \delta'(q'_i, x)$, $q'_i \in Q'$;
— $\delta''(q_i, \alpha) = q'_i$, $q_i \in Q$.

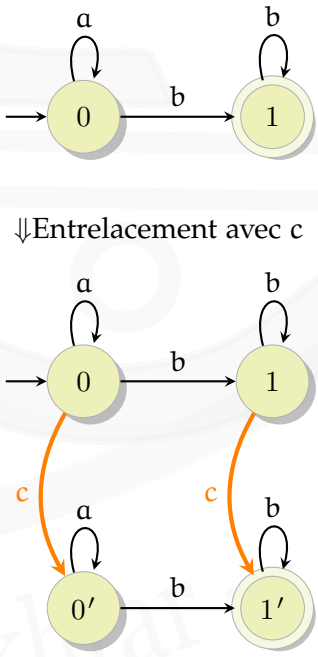

FIGURE 2.12 – Exemple d'entrelacement (ici c'est l'entrelacement avec le symbole c). Les arcs oranges sont les arcs (ayant été rajoutés) reliant les états du premier automate avec leurs correspondants du deuxième automates.

4..3 Produit d'automates

Définition 2.8 : Soient $A = (X, Q, q_0, F, \delta)$ et $A' = (X', Q', q'_0, F', \delta')$ deux automates à états finis. On appelle produit des deux automates A et A' l'automate $A'' = (X'', Q'', q''_0, F'', \delta'')$ défini comme suit :

- $X'' = X \cap X'$;
- $Q'' = Q \times Q'$;
- $q''_0 = (q_0, q'_0)$;
- $F'' = F \times F'$;
- $\delta''((q, q'), a) = \delta(q, a) \times \delta'(q', a)$

Cette définition permet de synchroniser l'analyse d'un mot par les deux automates. On pourra facilement démontrer que si w est un mot accepté par A'' alors il est également accepté par A et A'. Ainsi, si $L(A)$ est le langage accepté par A et $L(A')$ est le langage accepté par le langage A' alors l'automate A'' accepte l'intersection des deux langages : $L(A) \cap L(A')$.

Exemple 2.10 : produit d'automates

Considérons la figure 2.13.

- L'automate (1) accepte les mots sur $\{a, b, c\}$ contenant au moins deux a.
- L'automate (2) accepte les mots sur $\{a, b, c\}$ contenant au moins deux b.
- L'automate (3) représente le produit des deux automates.

Remarquons que dans l'automate (3), tout chemin qui part de l'état initial vers l'état final passe forcément par deux a et deux b (tout ordre est possible). Or, ceci est exactement le langage résultant de l'intersection des deux premiers langages.

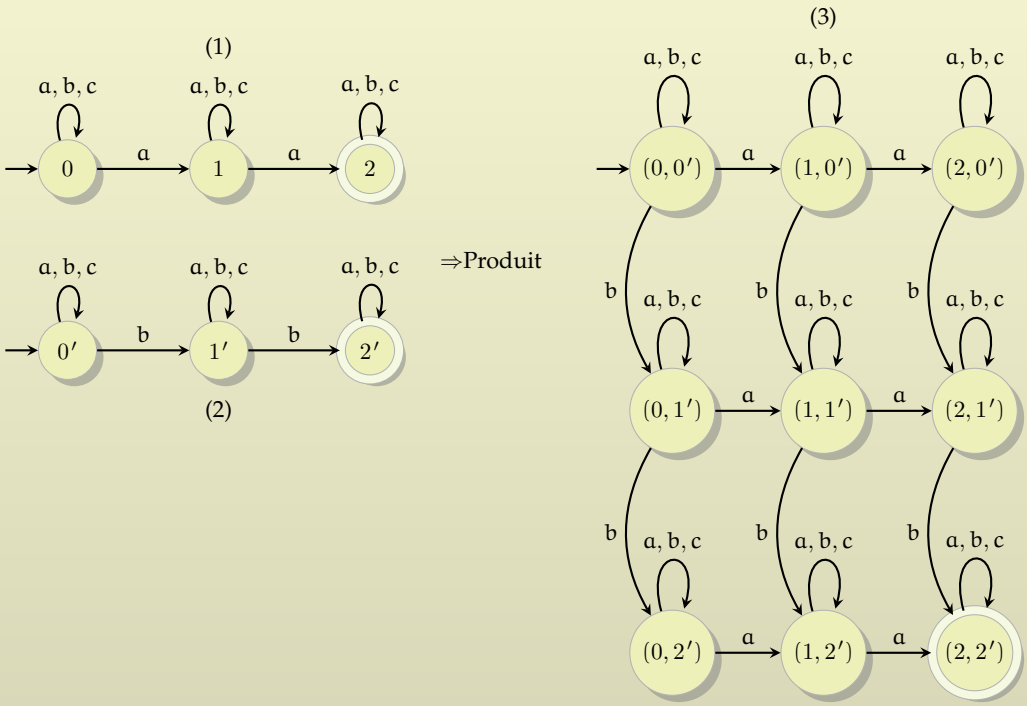

FIGURE 2.13 – Exemple de produit d'automates

4..4 Le langage miroir

Soit $A = (X, Q, q_0, F, \delta)$ un automate acceptant le langage $L(A)$. L'automate qui reconnaît le langage $(L(A))^R$ est accepté par l'automate $A^R = (X, Q, F, \{q_0\}, \delta^R)$ tel que : $\delta^R(q', a) = q$ si $\delta(q, a) = q'$.

En d'autres termes, il suffit juste d'inverser les sens des arcs de l'automate et le statut initial/final des états initiaux et finaux. Notons, par ailleurs, que l'automate A^R peut contenir plusieurs états initiaux ce qui signifie qu'il est le plus souvent non-déterministe. Pour éliminer le problème des multiples états initiaux, il suffit de rajouter un nouvel état initial et de le raccorder aux anciens états finaux par des ε-transitions.

Deuxième méthode de minimisation

La construction du miroir a une application surprenante, qui permet de minimiser un AEF quelconque. Soit A un AEF acceptant le langage $L(A)$. L'algorithme de minimisation est le suivant :

1. Construire l'AEF de $(L(A))^R$, appelons cet AEF A^R ;
2. Déterminiser A^R, appelons l'AEF résultat A^R_{det} ;
3. Construie l'AEF du miroir à partir de A^R_{det}, soit A' l'AEF résultant ;
4. Déterminiser A', le résultat représente l'AEF déterministe et minimal acceptant le langage $L(A)$.

Notons, enfin, que cet algorithme requiert que l'on accepte que l'AEF ait plusieurs états initiaux. Autrement, il ne fonctionne pas.

5. Simulation des AEF

Dans cette section, on s'intéresse à la construction d'un programme simulant le fonctionnement d'un AEF. On suppose alors que l'on dispose d'un AEF déterministe (pas forcément minimal) et que l'on veuille produire un programme en langage C qui le simule.

Tout d'abord, nous avons besoin de simuler certains éléments de la machine de Turing comme le ruban, la lecture, le déplacement de la tête, etc. Pour ce faire, on utilise les hypothèses suivantes :

— Le ruban est représenté par un tableau de caractères *word*. La tête de lecture est représentée par un entier `head` (initialisé à 0), la longueur du mot est stockée dans la variable entière `len`.

— Le déplacement de la tête se fait par la fonction `next()` qui renvoie le prochain caractère de la lecture. À la fin du mot, il renvoie 0.

Le code de la fonction peut être le suivant :

```
int next()
{
```

```
 if (head<len)
 return word[head++];
 else
 return 0;
}
```

La transcription d'un AEF $A = (X, Q, q_0, F, \delta)$ en un code C++ (car cela simplifie l'écriture seulement) se fait comme suit (le code peut bien sûr être amélioré par la suite) :

```
int state=q0;
char c=next();
int go=1;
while(go&&c)
{
 switch(state)
 {
  Pour chaque état q
  Mettre : case q :
  Mettre switch(c)
  //s'il y a des transitions sortantes, sinon ne rien mettre
  {
   Pour chaque symbole a tel que δ(q,a)=q'
   mettre :  case a:state=q';break;
   default:go=0;
  }
 }
 if (go) c=next();
}
if (c==0&&state|q ∈ F)
printf("%s","Accepted");|
else
printf("%s","Rejected");
```

A titre d'exemple, prenons l'automate de la figure 2.7, le code à écrire est alors le suivant :

```
int state=0;
char c=next();
int go=1;
while(go&&c)
{
    switch(state)
    {
        case 0 :
        switch(c)
        {
            case 'a' :
            state=1;
            break;
            case 'b' :
```

```
            state=0;
            break;
            default:
            go=0;
        }
        break;

        case 1 :
        switch(c)
        {
            case 'a' :
            state=1;
            break;
            case 'b' :
            state=2;
            break;
            default:
            go=0;
        }
        break;

        case 2 :
        switch(c)
        {
            case 'a' :
            state=3;
            break;
            case 'b' :
            state=2;
            break;
            default:
            go=0;
        }
        break;

        case 3 :
        switch(c)
        {
            case 'a' :
            state=3;
            break;
            case 'b' :
            state=2;
            break;
            default:
            go=0;
        }
        break;
    }
```

```
    if (go) c=next();
}
if (c==0&&(state==2||state==3))
printf("%s","Accepted");
else
printf("%s","Rejected");
```