



AUTOMATES À ÉTATS FINIS (AEF)

5

Dans ce chapitre, nous étudions les langages réguliers à travers les automates à états finis (AEF) appelés également automates finis (AF). Les AEFs sont les plus simples parmi les différentes machines d'analyse de langages, notamment car ils ne comportent pas de mémoire (l'AEF se souvient que de son état dans lequel il se trouve). L'absence de mémoire signifie que ces automates sont adaptés pour reconnaître (accepter) des langages réguliers, les plus simples parmi les quatre classes de Chomsky. Cette simplicité est liée aux restrictions grammaticales qui génèrent ces langages et permettent ainsi la conception d'algorithmes efficaces pour optimiser les AEFs.

Les automates à états finis peuvent être utilisés pour modéliser et résoudre divers problèmes dont la solution n'est pas évidente. Par exemple dans :

- L'analyse lexicale : Dans la compilation de programmes informatiques, en découplant le code source en "tokens" ou unités lexicales comme les mots-clés, les identificateurs, les opérateurs, etc., pour les analyser
- La vérification de protocoles de communication et la détection d'erreurs dans les systèmes informatiques.
- Le traitement sur les chaînes de caractères : La recherche des mots-clés dans un document ou la validation de numéros de téléphone.
- La recherche de motifs génétiques pour identifier la présence d'un motif dans un échantillon d'ADN.

5.1 Représentations formelles

5.1.1 Représentation mathématique

➊ Définition 5.1.1

La définition abstraite d'un AEF (ou AF) se donne par le quintuplet (X, Q, s_0, F, δ) tel que :

- X : C'est l'alphabet (l'ensemble des symboles) sur lequel sont définis les entrées (mots à analyser).
- Q : Un ensemble fini états.
- s_0 : C'est l'état initial $s_0 \in Q$.
- F : L'ensemble des états finaux ou les états d'acceptation ($F \subseteq Q$).
- δ : La fonction de transition d'un état vers un autre selon le symbole couramment lu tel que :
$$\delta : Q \times (X \cup \{\varepsilon\}) \mapsto 2^Q$$

➊ Définition 5.1.2 Une configuration dans AEF

Une configuration dans un AEF défini par le couple (s_i, a) signifie que l'automate se trouve dans un état donné $s_i \in Q$ et lit un symbole donné $a \in X$.

➋ Définition 5.1.3 Une transition dans un AEF

Une transition $\delta(s_i, a)$ permet de lire un symbole a et de changer ou non l'état de l'AEF. Lors du changement d'état, il peut y avoir un ou plusieurs états possibles de Q . On écrit : $\delta(s_i, a) = \{s_{j_1}, s_{j_2}, \dots, s_{j_k}\}$. Dans le cas où $\delta(s_i, a) = \emptyset$, cela signifie que la transition n'existe pas.

💡 Déduction 5.1.1 $w \in L(A)$ d'un point de vue AEF

- $w \in L(A)$ si w est accepté par un l'AEF A .
- Un mot w est accepté par un AEF A si, après avoir lu tout le mot, l'AEF se trouve dans un état final $s_f \in F$. Autrement dit, l'AEF se trouve dans la configuration (s_f, ε) . Ici, ε marque la fin du mot (la fin d'une séquence d'analyse).
- ▶ L'ensemble de tous les mots acceptés par A est représenté par le langage $L(A)$. On dit que L est accepté par A .
- Le reste des mots $X^* - L(A)$ sont rejetés par A . On dit que l'AEF A joue le rôle d'un classifieur qui distingue les mots $L(A)$ de $X^* - L(A)$

💬 Remarque 5.1.1 Le rejet d'un mot par un AEF

Un mot w est rejeté par un AEF A dans deux cas :

- Lorsque l'AEF se trouve dans un état $s \in Q$ et le symbole en cours de lecture $a \in X$, mais la transition $\delta(s, a)$ n'existe pas.
- Le mot w est entièrement lu par A , ce qui veut dire que A se trouve dans une configuration (s, ε) mais $s \notin F$ (l'automate n'est pas dans un état final).

📋 Exemple de définition d'un AEF et séquence d'analyse d'un mot

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

$$\begin{array}{ll} \delta(0, a) = \{0\} & \delta(0, b) = \{1\} \\ \delta(1, a) = \emptyset & \delta(1, b) = \{1\} \end{array}$$

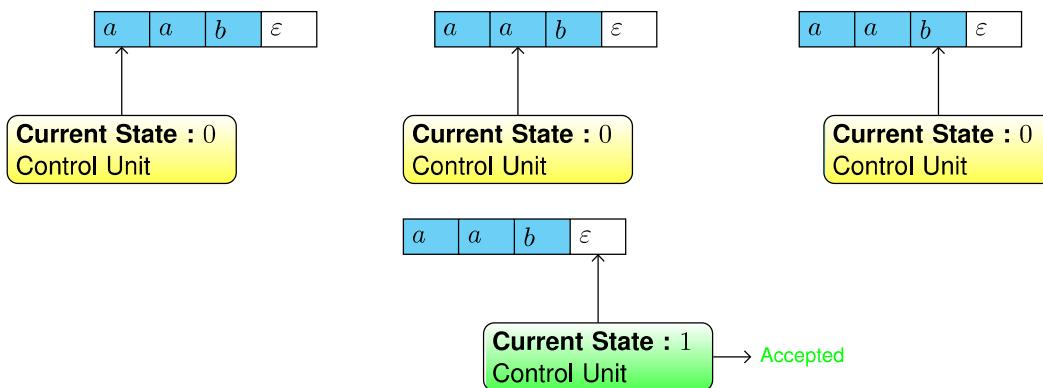
Voici 3 mots à analyser :

Séquence d'analyse

L'analyse de chaque mot consiste à donner la (les) séquences de configurations. Chacune se termine par une réponse (mot accepté ou rejeté).

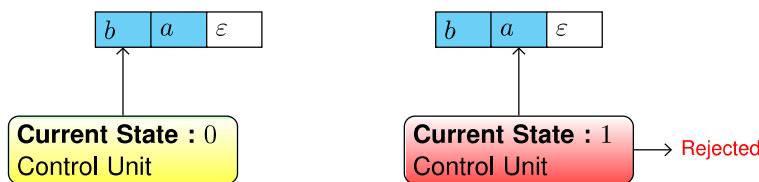
1 Le mot : aab : $(0, a) \rightarrow (0, a) \rightarrow (0, b) \rightarrow (1, \varepsilon)$. Mot accepté.

Car l'état de sortie 1 qui est final

FIGURE 5.1 – Le fonctionnement de l'AEF 'A' pendant l'analyse d'un mot : aab 

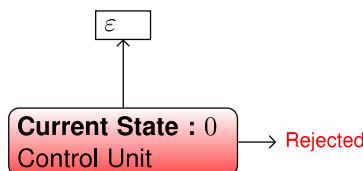
- 2** Le mot : ba : $(0, b) \rightarrow (1, a)$. **Mot rejeté.**

Car le mot n'a pas été entièrement lu, parce qu'on n'a pas de transition $(1, a)$.

FIGURE 5.2 – Le fonctionnement de l'AEF 'A' pendant l'analyse d'un mot : ba 

- 3** Le mot : ε : $(0, \varepsilon)$. **Mot rejeté.**

Car l'AEF n'est pas dans un état final même si le mot a été entièrement lu en une seule configuration.

FIGURE 5.3 – Le fonctionnement de l'AEF 'A' pendant l'analyse d'un mot : ε 

5.1.2 Représentation tabulaire

La représentation tabulaire consiste à définir un AEF par une table à deux dimensions $n \times m$, tel que $n = \text{card}(Q)$ correspond au nombre d'états et m correspond au nombre de symboles dans X . Autrement dit, chaque ligne i correspond à un état (s_i) et chaque colonne j correspond à un symbole. Une case $[i, j] = \delta(i, j)$ est le résultat de la transition qui part de l'état s_i en lisant le symbole j

Remarque 5.1.2

La définition par table n'est pas suffisante pour définir entièrement l'AEF, car elle ne donne ni l'état initial ni les états finaux.

Exemple Rprésentation tabulaire de l'AEF précédent

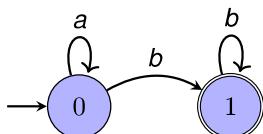
État	a	b
0	0	1
1	-	1

5.1.3 Représentation graphique

La représentation graphique consiste à représenter un AEF par un graphe orienté $G(N, E)$, tel que :

- 1 Les nœuds $N = Q$ représentent les états, schématisé par un rond. L'état initial est désigné par une flèche entrante au nœud correspondant tandis que les états finaux sont marqués par un double rond.
- 2 E c'est l'ensemble d'arcs étiquetés qui représentent les transitions. Si l'AEF contient une transition $\delta(s_i, a) = s_j$ alors on ajoute un arc étiqueté par le symbole a allant du nœud s_i vers le nœud s_j . Lorsqu'il y a plusieurs symboles a_1, \dots, a_k tels que $(s_i, a_l) = \{s_j\}$, ($l = 1..k$) alors l'arc (s_i, s_j) peut être décoré par l'ensemble $\{a_1, \dots, a_k\}$.

Exemple Représentation graphique de l'AEF précédent



Peut-on déduire le langage accepté $L(A)$?

$$L(A) = \{a^p b^q | p \geq 0, q > 0\}.$$



Déduction 5.1.2 L'acceptation d'un mot par un AEF d'un point de vue graphique

La définition d'un AEF $A = (X, Q, s_0, F, \delta)$ par un graphe permet de définir autrement l'acceptation d'un mot $w \in L(A)$.

Un mot w est accepté par A s'il existe un chemin dans le graphe entre deux nœuds, le premier représente état initial s_0 et le dernier est un état final.

Def 5.1.4 Un chemin correspond à un mot w

Un chemin dans le graph de A est toute séquence d'états $s_0, s_1, s_2, \dots, s_n$ tels que $\forall i : s_i \in Q$ et $\exists a_i \in X : s_{i+1} \in \delta(s_i, a_i)$. Le mot w qui correspond à ce chemin : $a_0 a_1 a_2 \dots a_{n-1}$.



Exemple Quelques chemins selon l'AEF précédent

- Le chemin $0, 0, 0, 0$ correspond au mot aaa mais ce dernier est rejeté car le chemin ne se termine pas par un état final.
- Le chemin $0, 0, 0, 1$ correspond au mot aab qui est accepté.
- Le chemin $0, 1, 1, 1, 1, 1$ correspond au mot $abbbb$ qui est accepté.
- Le chemin $1, 1, 1, 1$ correspondant au mot bbb qui est rejeté car le chemin ne commence pas par l'état initial

5.2

Le déterminisme des automates

Le déterminisme d'un programme informatique (un automate) est une caractéristique importante permettant de simplifier le traitement (l'analyse). En général, un programme informatique est déterministe. Autrement dit son comportement est clairement défini pour toutes les situations envisageables. Cependant, certaines conditions d'exécution peuvent rendre difficile la prédiction de ce comportement, notamment lorsque l'on doit interagir avec des machines distantes ou des environnements variables.

► Donc, il est préférable de manipuler une version déterministe d'un programme (d'un automate) qu'une version non-déterministe car cette dernière n'est pas optimale et risque d'augmenter la complexité d'analyse de maintenance.

5.2.1 AEF déterministe Vs. non-déterministe

❶ Définition 5.2.1 Automate déterministe (AFD)

Un AEF $A = (X, Q, s_0, F, \delta)$ est dit déterministe (AFD ou DFA en anglais) s'il vérifie les deux conditions suivantes :

- ❶ $\forall s_i \in Q, \forall a \in X$, il existe au plus un état s_j tel que $\delta(s_i, a) = s_j$.
- ❷ L'automate ne comporte pas de transitions avec ε -transitions).

❷ Remarque 5.2.1

Selon cette définition un AEF est dit déterministe lorsqu'il sait déterminer son prochain état peu importe sa configuration (s_i, a) pendant l'analyse d'un mot donné, ça veut dire que son prochain comportement est connu avec certitude.

❸ Définition 5.2.2 Automate non-déterministe (AFN)

Un AEF non-déterministe (AFN ou NFA en anglais) est un AEF qui ne vérifie pas l'une des deux conditions précédentes :

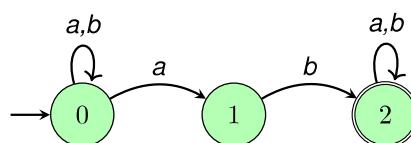
- ❶ Si la condition (1) n'est pas vérifiée, cela veut dire qu'il existe au moins un cas de non-déterminisme $\delta(s_i, a) = \{s_0, s_1, \dots, s_n\}$. C'est une configuration dans laquelle il existe plusieurs transitions envisageables.
 - Dans une telle situation l'AEF effectue une transition vers l'un de ces états s_0, s_1, \dots, s_n en allant jusqu'au bout de la séquence de l'analyse. Si l'entrée n'est pas acceptée, l'AEF doit revenir en arrière pour prendre un autre chemin d'analyse.
- ❷ Si la condition (2) n'est pas vérifiée cela veut dire que l'AEF comporte des ε -transitions (ε -NFA)
 - L'existence d'une ε -transition appelée aussi transition spontanée $\delta(s_i, \varepsilon) = s_j$ signifie que l'AEF peut passer de s_i à s_j sans avoir besoin de lire un symbole (C'est le mot vide : ε)

❹ Exemple Un AEF non-déterministe

Nous donnons deux exemples d'un AEF non-déterministe, l'un ne vérifie pas la condition (1) et l'autre ne vérifie pas la condition (2). Les deux automates acceptent le même langage, celui des mots définis sur $\{a, b\}$ et qui contiennent le facteur ab .

- A_{nondet}^1 ne vérifiant pas la condition (1) :

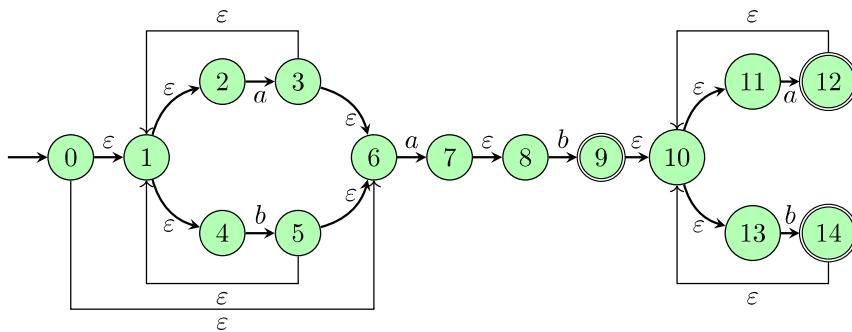
— Représentation graphique :



— Représentation tabulaire : état initial 0, état final 2

Etat / Symboles	a	b
0	0,1	0
1	-	2
2	2	2

- A_{nondet}^2 ne vérifiant pas la condition (2) :



Remarque 5.2.2

- Un AEF non-déterministe peut ne pas vérifier les deux conditions en même temps.
- Il est plus facile de construire un AEF non-déterministe que de construire un AEF déterministe
(On en verra l'illustration dans le chapitre suivant)

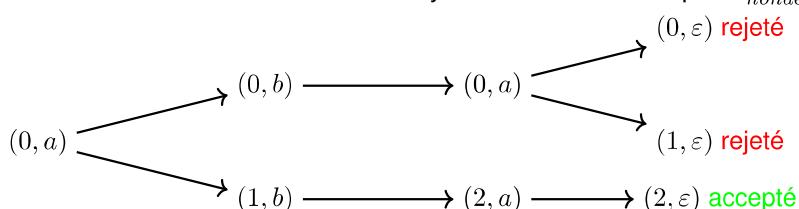
Question. Peut-on rendre un AEF non-déterministe, déterministe ?

► En répondant à cette question, on verra pourquoi et comment déterminiser un AEF.

5.2.2 Pourquoi déterminiser un AEF

Multiples séquences d'analyse

Pour répondre à cette question et montrer l'intérêt de la déterminisation, nous étudions le comportement d'un AEF non-déterministe à travers l'analyse d'un mot $w = aba$ par A_{nondet}^1 (vu dans l'exemple précédent) :



L'analyse part de la première configuration $(0, a)$. À ce stade il y a deux possibilités $\{0, 1\}$. Donc A_{nondet}^1 est incapable de déterminer avec certitude quel est son prochain état qui pourrait aboutir par la suite à l'acceptation. Dans une telle situation et tant que la tête L/E se déplace dans un seul sens, l'AEF choisit arbitrairement un état, parmi $\{0, 1\}$ et continue l'analyse. Si ce chemin mène à l'acceptation du mot, alors on dira que le choix était bon. Sinon, tant que le mot est rejeté, l'AEF revient au dernier point de non-déterminisme afin de choisir un autre état et poursuivre l'analyse.

Remarque 5.2.3

La question d'analyse d'un mot w évoque souvent l'exploration de tous les chemins (les séquences) d'analyse qui mènent à l'acceptation ou au rejet de w .

Question. Comment peut-on analyser le même mot avec A_{nondet}^2 ?

Problème de complexité

On sait que la complexité d'analyse d'un langage régulier par un AEF est théoriquement linéaire (Chapitre précédent). Or, l'exemple précédent montre que l'analyse d'un mot de longueur n ($n = |aba| = 3$), n'engendre pas autant de configurations. Mais il y en a plus (8 configurations). La complexité d'analyse pourrait atteindre ainsi au pire des cas $f(n) = t^n$ ($t > 1$) afin d'arriver à explorer toutes les séquences d'analyse et trouver

celle(s) qui mène(ent) éventuellement à l'acceptation. Cet exemple illustre d'une certaine manière le problème que pose les automates non-déterministes.

Exemple

Prenons un mot w tel que $|w| = 100$, pour $t = 2$, le nombre de configurations dont on a besoin pour analyser w peut atteindre l'ordre de $2^{100} = 10^{30}$. Cela veut dire que l'analyse pourrait durer des milliers de siècles même sur l'ordinateur le plus puissant.

Déduction 5.2.1 L'intérêt de la déterminisation

L'intérêt majeur de la déterminisation d'un AEF consiste à réduire la complexité d'analyse, car une version déterministe d'un AEF offre une complexité linéaire : Un nombre de configurations $f(n) = n+1$.

Bien que les automates non-déterministes puissent générer des coûts élevés lors de l'analyse, ils sont souvent plus faciles à concevoir que les automates déterministes. On en verra l'illustration dans le chapitre suivant (Construire des AEFs non-déterministe avec ε -transitions depuis des expressions régulières).

Théorème 5.2.1 Rabin et Scott

Tout langage accepté par un AEF non-déterministe peut également être accepté par un AEF déterministe.

Ce théorème (sans démonstration) établit l'équivalence entre les automates déterministes et non-déterministes, ce qui simplifie l'analyse des langages réguliers.

Déduction 5.2.2 Tout AEF A_{nondet} peut être transformé en un AEF A_{det}

Pour chercher l'AEF acceptant un langage régulier donné, nous pouvons commencer par chercher un AEF non-déterministe (qui est plus facile à construire) et qui peut être transformé en un AEF déterministe.

Voyons maintenant comment rendre par exemple A_{nondet}^1 , A_{nondet}^2 déterministe...

5.2.3 Déterminisation d'un AEF sans ε -transitions

Nous étudions la déterminisation d'un AEF non-déterministe à travers un algorithme général de déterminisation qui s'applique en réalité sur tous les AEFs non-déterministes que ce soient sur ceux ne vérifiant pas la première condition ou ceux ne vérifiant pas la deuxième condition (comportant des ε -transitions) ou les deux. Mais pour des raisons de simplicité, nous allons appliquer cet algorithme sur les AEFs non-déterministes qui ne vérifient pas la première condition (sans ε -transitions).

Algorithmme

Algorithm 1 Déterminisation d'un AEF sans ε -transitions

Require: $A_{nondet}(X, Q, s_0, F, \delta)$
Ensure: A_{det}

- 1: Créer un nouvel état initial $E^{(0)} = \{s_0\}$
- 2: Construire un nouvel ensemble (nouvel état dans A_{det}) : $E^{(1)}$ à partir de $E^{(0)}$ par rapport à un symbole $a \in X$: $E^{(1)} = \bigcup_{s \in E^{(0)}} \delta(s, a)$
- 3: **while** il y a un nouvel ensemble $E^{(i)}$ **do**
- 4: **for** chaque symbole $a \in X$ **do**
- 5: Calculer $E^{(i+1)} = \bigcup_{s \in E^{(i)}} \delta(s, a)$
- 6: **end for**
- 7: **end while**
- 8: Chaque ensemble $E^{(i)}$ représente un état dans A_{det} . (Renumeroter les ensembles en tant qu'états simples)
- 9: L'ensemble $E^{(0)}$ représente l'état initial dans A_{det} .
- 10: Chaque ensemble $E^{(i)}$ contenant au moins un état final de A_{nondet} représente un état final dans A_{det}
- 11: **return** A_{det}

Remarque 5.2.4

L'idée de base de cet algorithme consiste à regrouper les états dans des ensembles qui seront considérés plus tard comme des états dans l'AEF déterministe. Comme ça, toutes les transitions (les états) envisageables au niveau d'une situation de non-déterminisme, seront agrégées dans un seul état.

Exemple Déterminisation de A_{nondet}^1

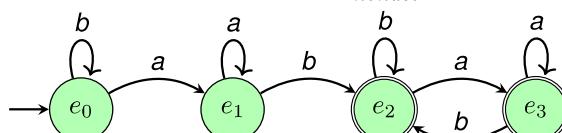
Nous allons appliquer l'algorithme 1 sur l'AEF non-déterministe A_{nondet}^1 vu précédemment.

État / Symboles	a	b
$e_0 = \{0\}$	$\{0, 1\}$	$\{0\}$
$e_1 = \{0, 1\}$	$\{0, 1\}$	$\{0, 2\}$
$e_2 = \{0, 2\}$	$\{0, 1, 2\}$	$\{0, 2\}$
$e_3 = \{0, 1, 2\}$	$\{0, 1, 2\}$	$\{0, 2\}$

Table de déterminisation de A_{nondet}^1

État / Symboles	a	b
e_0	e_1	e_0
e_1	e_1	e_2
e_2	e_3	e_2
e_3	e_3	e_2

Représentation tabulaire de A_{det}^1



En résultat nous avons obtenu l'AEF déterministe : $A_{det}^1(\underbrace{\{a, b\}}_X, \underbrace{\{e_0, e_1, e_2, e_3\}}_Q, e_0, \underbrace{\{e_2, e_3\}}_F, \delta)$

Question. Nous avons vu comment analyser le mot $w = aba$ par A_{nondet}^1 . Maintenant, que se passera-t-il si on analyse w avec sa version déterministe A_{det}^1 ?

$(e_0, a) \longrightarrow (e_1, b) \longrightarrow (e_2, a) \xrightarrow{(e_3, \varepsilon)} \text{accepté}$

Remarque 5.2.5 $L(A_{nondet}^1) = L(A_{det}^1)$

- On remarque que le mot $w = aba$ est également accepté par A_{det}^1 . Si le mot est rejeté par A_{det}^1 il sera également rejeté par A_{nondet}^1 .

► Par conséquent : $L(A_{nondet}^1) = L(A_{det}^1)$

- On remarque aussi que l'analyse de w par A_{det}^1 nécessite un nombre plus limité de configurations. Donc une analyse moins complexe.

5.2.4 Déterminisation d'un AEF avec les ε -transitions

Si un AEF non-déterministe contient au moins une ε -transition, nous ferons appel à un deuxième algorithme de déterminisation. L'idée de cet algorithme ressemble à celle de l'algorithme précédent mais tout en ajoutant une nouvelle notion appelée : ε -fermeture (ε -closure) d'un ensemble d'états.

⌚ Définition 5.2.3 ε -fermeture : $\varepsilon_f()$

L' ε -fermeture d'un ensemble d'états E , noté $\varepsilon_f(E)$ est défini par l'ensemble des états de E en ajoutant tous les états accessibles depuis les états de E par un chemin dont les transitions sont étiquetées que par ε .

🔗 Algorithme

Algorithm 2 ε_f d'un ensemble E

Require: E

Ensure: $\varepsilon_f(E)$

```

1:  $\varepsilon_f = E$ 
2: repeat
3:   for chaque nouvel état  $s \in \varepsilon_f$  do
4:      $\varepsilon_f = \varepsilon_f \cup \delta(s, \varepsilon)$ 
5:   end for
6: until l'ensemble  $\varepsilon_f$  ne change plus
7: return  $\varepsilon_f$ 
```

📋 Exemple Calcul de ε_f

Nous allons calculer ε_f de quelques ensembles d'états à partir de l'AEF A_{nondet}^2 :

- $\varepsilon_f(\{0\}) = \{0, 1, 2, 4, 6\}$
- $\varepsilon_f(\{1, 2\}) = \{1, 2, 4\}$
- $\varepsilon_f(\{3\}) = \{1, 2, 3, 4, 6\}$

Nous allons pouvoir maintenant définir l'algorithme de déterminisation des AEFs non-déterministes contenant des ε -transitions :

 Algorithme
Algorithm 3 Déterminisation d'un AEF avec ε -transitions**Require:** $A_{nondet}(X, Q, s_0, F, \delta)$ **Ensure:** A_{det}

- 1: Créer un nouvel état initial $E^{(0)} = \varepsilon_f(\{s_0\})$
- 2: Construire un nouvel ensemble (nouvel état dans A_{det}) : $E^{(1)} = \varepsilon_f(\bigcup_{s \in E^{(0)}} \delta(s, a))$ à partir de $E^{(0)}$ par rapport à un symbole $a \in X$
- 3: **while** il y a un nouvel ensemble $E^{(i)}$ **do**
- 4: **for** chaque symbole $a \in X$ **do**
- 5: Calculer $E^{(i+1)} = \varepsilon_f(\bigcup_{s \in E^{(i)}} \delta(s, a))$
- 6: **end for**
- 7: **end while**
- 8: Chaque ensemble $E^{(i)}$ représente un état dans A_{det} . (Renumeroter les ensembles en tant qu'états simples)
- 9: l'ensemble $E^{(0)}$ représente l'état initial dans A_{det} .
- 10: Chaque ensemble $E^{(i)}$ contenant au moins un état final de A_{nondet} représente un état final dans A_{det}
- 11: **return** A_{det}

 **Exemple** Déterminisation de A_{nondet}^2
Nous allons appliquer l'algorithme 3 sur l'AEF non-déterministe A_{nondet}^2 vu précédemment.

État	a	b
$e_0 = \varepsilon_f(\{0\}) = \{0, 1, 2, 4, 6\}$	$\underbrace{\{1, 2, 3, 4, 6, 7, 8\}}_{\varepsilon_f(\{3,7\})}$	$\underbrace{\{1, 2, 4, 5, 6\}}_{\varepsilon_f(\{5\})}$
$e_1 = \{1, 2, 3, 4, 6, 7, 8\}$	$\underbrace{\{1, 2, 3, 4, 6, 7, 8\}}_{\varepsilon_f(\{3,7\})}$	$\underbrace{\{1, 2, 4, 5, 6, 9, 10, 11, 13\}}_{\varepsilon_f(\{5,9\})}$
$e_2 = \{1, 2, 4, 5, 6\}$	$\underbrace{\{1, 2, 3, 4, 6, 7, 8\}}_{\varepsilon_f(\{3,7\})}$	$\underbrace{\{1, 2, 4, 5, 6\}}_{\varepsilon_f(\{5\})}$
$e_3 = \{1, 2, 4, 5, 6, 9, 10, 11, 13\}$	$\underbrace{\{1, 2, 3, 4, 6, 7, 8, 10, 11, 12, 13\}}_{\varepsilon_f(\{3,7,12\})}$	$\underbrace{\{1, 2, 4, 5, 6, 10, 11, 13, 14\}}_{\varepsilon_f(\{5,14\})}$
$e_4 = \{1, 2, 3, 4, 6, 7, 8, 10, 11, 12, 13\}$	$\underbrace{\{1, 2, 3, 4, 6, 7, 8, 10, 11, 12, 13\}}_{\varepsilon_f(\{3,7,12\})}$	$\underbrace{\{1, 2, 4, 5, 6, 9, 10, 11, 13, 14\}}_{\varepsilon_f(\{5,9,14\})}$
$e_5 = \{1, 2, 4, 5, 6, 10, 11, 13, 14\}$	$\underbrace{\{1, 2, 3, 4, 6, 7, 8, 10, 11, 12, 13\}}_{\varepsilon_f(\{3,7,12\})}$	$\underbrace{\{1, 2, 4, 5, 6, 10, 11, 13, 14\}}_{\varepsilon_f(\{5,9,14\})}$
$e_6 = \{1, 2, 4, 5, 6, 9, 10, 11, 13, 14\}$	$\underbrace{\{1, 2, 3, 4, 6, 7, 8, 10, 11, 12, 13\}}_{\varepsilon_f(\{3,7,12\})}$	$\underbrace{\{1, 2, 4, 5, 6, 10, 11, 13, 14\}}_{\varepsilon_f(\{5,14\})}$

Après la simplification de cette table de déterminisation, nous obtenons :

État	a	b
e_0	e_1	e_2
e_1	e_1	e_3
e_2	e_1	e_2
e_3	e_4	e_5
e_4	e_4	e_6
e_5	e_4	e_5
e_6	e_4	e_5

La déterminisation de A_{nondet}^2 , nous permet de définir :

$$A_{det}^2(\underbrace{\{a, b\}}_X, \underbrace{\{e_0, e_1, e_2, e_3, e_4, e_5, e_6\}}_Q, e_0, \underbrace{\{e_3, e_4, e_5, e_6\}}_F, \delta)$$

5.3**Minimisation d'un AEF déterministe**

Après avoir introduit les AEFs déterministes et non-déterministes et présenté les algorithmes permettant de déterminiser un AEF, nous avons vu selon les exemples précédents qu'un langage peut être accepté par plusieurs AEFs. (non-déterministes / déterministes). Cela nous conduit vers la définition suivante :

 **Définition 5.3.1 Automates équivalents**

Deux automates finis A_1 et A_2 sont équivalents s'ils reconnaissent le même langage L . $L(A_1) = L(A_2)$

? **Question.** Mais, comment-il possible de simplifier encore (minimiser) un AEF déterministe afin de réduire la complexité d'analyse (le nombre de configurations nécessaires pour analyser un mot) ?

5.3.1 Pourquoi minimiser un AEF ?

Nous traitons la question de minimisation sur des AEFs déterministes dont la complexité d'analyse ne dépend pas en réalité 'seulement' de la longueur de l'entrée.

? **Question.** Le nombre d'états d'un automate peut-t-il avoir un impact sur le coût d'analyse d'un mot ?

En réalité, on se pose souvent cette question après la déterminisation qui pourrait augmenter le nombre d'états dans un (A_{det}) par rapport à sa version non-déterministe (A_{nondet}). Dans certains cas, il est tout à fait possible que A_{det} ait exactement le même nombre d'états que celui que l'on trouve dans A_{nondet} .

► Mais en général, si un AEF non-déterministe contient x états, le nombre d'états de sa version déterministe pourrait atteindre $2^x - 1$. Certains ne sont pas toujours indispensables.

Il faut savoir que si x est très élevé, la représentation en mémoire d'un AEF déterministe, malgré sa complexité linéaire pourrait coûter plusieurs pages (espaces) de mémoire. Cela peut augmenter non seulement le temps d'analyse mais aussi la consommation électrique requise pour stocker les données (Par exemple dans les systèmes embarqués)

 **Déduction 5.3.1 Accepter le même mot en passant par un nombre inférieur d'états :**
 $L(A_{det}) = L(A_{Min})$

Afin d'améliorer la reconnaissance des chaînes et d'optimiser l'espace mémoire utilisée, on pourrait également réduire (**minimiser**) le nombre d'états d'un AEF déterministe, en lisant et acceptant le même langage.

 **Déduction 5.3.2 Un AEF minimal unique**

Si on a une multitude d'AEFs déterministes acceptant le même langage, il y aura qu'un seul AEF déterministe et minimal (ayant un nombre minimal d'états) acceptant le même langage.

► Il n'est pas évident et même possible de comparer deux AEFs afin de vérifier s'ils sont équivalents. En revanche, il suffit, de calculer l'automate déterministe minimal de chacun et de les comparer.

5.3.2 Comment minimiser un AEF ?

Il existe de nombreuses stratégies pour minimiser un AEF. Nous étudions la minimisation en se basant sur la caractérisation des états (des classes) et qui nécessite de supprimer ceux qui sont inutiles. Cela se fait en deux principales étapes :

- 1 Nettoyage de l'AEF : qui consiste à éliminer les états inaccessibles.
- 2 Regroupement des états congruents (des états β -équivalents) dans des classes de congruence (des classes d'équivalence).

Nettoyage de l'AEF

La première étape de la minimisation consiste à éliminer les états inaccessibles (improductifs).

❶ Définition 5.3.2 état inaccessible

Un état est inaccessible (inatteignable) s'il n'existe aucun chemin permettant de l'atteindre en partant de l'état initial.

💡 Remarque 5.3.1

Tout état inaccessible est improductif, mais un état improductif peut ne pas être inaccessible. Même s'il est accessible, un état improductif ne se trouve jamais sur le chemin d'acceptation d'un mot.

Pour détecter les états inaccessibles, il suffit d'appliquer un simple algorithme de marquage (marquage des nœuds accessibles dans un graphe). Mais comment ?

Création des classes de congruence regroupant des états β -indistinguables

Dans cette deuxième étape, nous allons regrouper les états selon un certain critère (des états indistinguables, ou des états β -équivalents) dans des classes qui seront considérées plus tard comme les états de l'AEF minimal.

❶ Définition 5.3.3 Les états distinguables

Dans un AEF déterministe, deux états s_i, s_j sont distinguables s'il existe un mot w tel que :

- Partant de l'état s_i : $(s_i, a) \xrightarrow{*} (s_f, \varepsilon)$ (a le premier symbole de w) on termine dans un état d'acceptation $s_f \in F$.
- Partant de l'état s_j : $(s_j, a) \xrightarrow{*} (s, \varepsilon)$ on termine dans un état de non acceptation $s \notin F$ (ou vice versa).

❶ Définition 5.3.4 Les états indistinguables (équivalents)

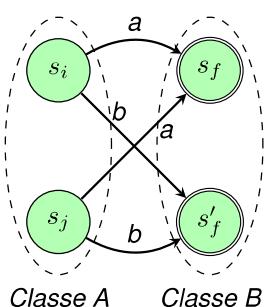
Dans un AEF déterministe deux états s_i, s_j sont indistinguables ou équivalents, s'ils ne sont pas distinguables. Autrement dit ils permettent d'atteindre des états finaux en lisant la même séquences de symboles (le même mot w).

❶ Définition 5.3.5 Les états β -indistinguables (β -équivalents)

Dans un AEF déterministe deux états s_i, s_j sont β -indistinguables ou β -équivalents (on écrit $s_i \beta s_j$) si en partant de l'état s_i ou s_j et après avoir lu les β prochains symboles d'entrée du mot w ($\beta \leq |w|$) on se retrouvera dans une configuration finale.

► Donc s_i et s_j sont indistinguables veut dire que $\forall \beta \geq 0, s_i \beta s_j$.

💡 Exemple Deux états β -équivalents



Partant de l'état s_i ou de l'état s_j , on peut atteindre les états finaux en lisant le mot a . C'est la même chose lorsqu'on lit le mot b .

💡 Remarque 5.3.2

- L'idée de la minimisation consiste à chercher ces états β -équivalents et les fusionner en une seule classe d'équivalence (Classe A : = $\{s_i, s_j\}$) qui représentera un seul état.
- Par transitivité, la relation d'indistinguabilité (de β -équivalence) est une relation d'équivalence appelée une relation de congruence. Si s_i et s_j sont β -équivalents, alors $\forall a \in X, \delta(s_i, a)$ et $\delta(s_j, a)$ sont également β -équivalents (c'est le cas par exemple de s_f et s'_f que l'on peut fusionner dans la même classe B).

⌚ Définition 5.3.6 Une Classe d'équivalence

Une classe d'équivalence ou de congruence est un ensemble qui regroupe des états β -équivalents.

L'algorithme de minimisation consiste à chercher itérativement tous les groupes d'états qui peuvent être distingués par une chaîne d'entrée. Chaque groupe d'états indistinguables est alors fusionné en un état unique.

💡 Algorithme

Algorithm 4 Création des classes de congruences fusionnant des états β -équivalents

Require: $A_{det}(X, Q, s_0, F, \delta)$ après nettoyage

Ensure: Une partition en classes de congruences

- 1: Créer 2 classes initiales $A = F, B = Q - F$
- 2: **repeat**
- 3: **for** chaque classe C **do**
- 4: Vérifier la cohérence de C
- 5: **for** chaque symbole $a \in X$ **do**
- 6: **if** $\forall s_i \in C, \delta(s_i, a) = s_j$ tel que $s_j \in C'$ ($C' = C$ ou une classe différente) **then**
- 7: on dit que la classe est cohérente par rapport à a (pas d'éclatement).
- 8: **else**
- 9: Si $\exists a \in X$ et deux états $s_i, s_j \in C$ tel que $\delta(s_i, a) \in C_1$ et $\delta(s_j, a) \in C_2$ et $C_1 \neq C_2$, alors créer une nouvelle classe pour séparer s_j et s_j (On ne laisse dans la même classe que les états β -équivalents états).
- 10: {Sortir de la boucle (ligne 3) pour faire une nouvelle itération de la boucle repeat ligne 2}
- 11: **end if**
- 12: **end for**
- 13: **end for**
- 14: **until** Les classes ne changent plus (Tant qu'il n'y a plus d'états à séparer)
- 15: **return** La partition en classes de congruence.

Initialement, l'algorithme forme une partition initiale (deux classes ou groupes) : états d'acceptation et le reste. Dans chaque itération, on étudie les transitions qui partent d'un groupe d'états sur un symbole. Si ces transitions conduisent à des états appartenant à au moins deux groupes différents, alors on doit diviser ce groupe. La division fait en sorte que les transitions depuis chaque groupe soient confinées à un seul groupe de la partition courante.

💡 Déduction 5.3.3 Comment définir l'AEF minimal

Cet algorithme itératif permet de raffiner à chaque itération la partition des états en groupes.

L'algorithme renvoie à la fin une partition dont le nombre de classes d'équivalence obtenu est égal

au nombre d'états de l'automate minimal tout en sachant que les état fusionnés dans une classe d'équivalence acceptent le même langage.

► Donc on définit l'AEF minimal $A_{Min}(X, Q, s_0, F, \delta)$ comme suivant :

- X : C'est le même vocabulaire de l'ancien AEF. (A_{Min} accepte le même langage).
- Q : Chaque classe d'équivalence représente un état dans A_{Min} .
- s_0 : La classe qui contient l'ancien état initial représente l'état initial de A_{Min} .
- Toute classe contenant au moins un état final de l'ancien AEF devient un état final dans A_{Min} .
- δ : Pour définir les transitions de A_{Min} , il faut regarder le résultat de la dernière itération de l'algorithme 4. Pour chaque classe A , $\forall s \in A, \forall a \in X$, on aura forcément $\delta(s, a) \in B$ (soit $B \neq A$ ou $B = A$) ce qui nous permet de déduire la transition $\delta(A, a) = B$ dans A_{Min}

Exemple Minimisation d'un AEF

Supposons que nous avons l'AEF déterministe représenté par la table suivante (sachant que 1 : est l'état initial, 1, 2 représentent les états finaux)

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

Pour minimiser cet AEF, il faut passer par deux étapes :

1. Nettoyage : On va éliminer les états inaccessibles en marquant les états accessibles à partir de l'état initial.

Soit $States_{acc} = \{1\}$, l'ensemble initial des états accessibles.

Partant de l'état 1, on peut accéder à 2 et 5 (en lisant a et b respectivement). Donc $States_{acc} = \{1, 2, 5\}$

Depuis 2 on peut accéder à 4. $States_{acc} = \{1, 2, 4, 5\}$.

Depuis 4 on peut accéder à 3. $States_{acc} = \{1, 2, 3, 4, 5\}$.

Depuis 5 on peut accéder à 6. $States_{acc} = \{1, 2, 3, 4, 5, 6\}$.

En revanche, on remarque que l'état 7 est inaccessible, alors il sera éliminé.

2. Crédation des classes de congruences : en appliquant l'algorithme 4 :

Selon la phase d'initialisation (ligne 1 de l'algorithme 4), on va créer une partition initiale en deux classes :

$$A = F = \{1, 2\} \text{ et } B = Q - F = \{3, 4, 5, 6\}$$

— Itération 01 :

Classe A	a	b
1	$2 \in A$	$5 \in B$
2	$2 \in A$	$4 \in B$

► On dit que la classe A est cohérente par rapport aux 2 symboles a et b.

Classe B	a	b
3	$3 \in B$	$2 \in A$
4	$5 \in B$	$3 \in B$
5	$4 \in B$	$6 \in B$
6	$6 \in B$	$1 \in A$

► La classe B est cohérente par rapport au symbole a mais elle est incohérente par rapport à b. Donc il faut séparer les états qui causent cette incohérence. Par exemple : $B = \{5, 4\}$ et on crée une nouvelle classe C = {3, 6}.

— Itération 02 : $A = \{1, 2\}$, $B = \{5, 4\}$, $C = \{3, 6\}$.

Classe A	a	b
1	$2 \in A$	$5 \in B$
2	$2 \in A$	$4 \in B$

► La classe A reste cohérente sans aucune modification.

Classe B	a	b
4	$5 \in B$	$3 \in C$
5	$4 \in B$	$6 \in C$

► La classe B est cohérente.

Classe C	a	b
3	$3 \in C$	$2 \in A$
6	$6 \in C$	$1 \in A$

► La classe C est cohérente.

Au bout de la deuxième itération, on remarque que toutes les classes sont cohérentes. On arrête alors l'algorithme car il n'y a plus de modifications possibles.

Donc on définit l'AEF minimal par $AEF_{Min}(\{a, b\}, \{A, B, C\}, A, \{A\}, \delta)$:

État	a	b
A	A	B
B	B	C
C	C	A

Remarque 5.3.3 Cas particuliers de la minimisation

- Si après la minimisation on obtient le même AEF, alors l'AEF est déjà minimal.
- Si $L = L'$, (L, L' étant réguliers), alors ils ont le même AEF déterministe et minimal (On peut être amené à éliminer certains autres états non nécessaires néanmoins).
- Si tous les états de l'AEF à minimiser sont finaux, alors l'algorithme 4 commence avec une seule classe ($A = F$). Il y aura deux cas de figure :
 - Si l'AEF est complet (voir plus loin), l'AEF minimal se réduit à un seul état.
 - Sinon, s'il y a une transition qui manque, tel que $\exists s \in Q, \exists a \in X, \delta(s, a) = \emptyset$, alors l'ensemble vide est différent de tout autre ensemble, et donc une séparation sera nécessaire.
- Si l'AEF ne contient aucun état final, sa version minimale est réduite à un seul état sans aucune transition

5.4

Opérations sur les AEFs

Dans cette section, nous allons aborder diverses opérations applicables aux AEFs. Cinq opérations fondamentales seront étudiées, en commençant par la complémentation d'un AEF. Les quatre opérations qui se poursuivent vont traduire certaines opérations sur les langages formels (vue précédemment) et nous permettront de démontrer plus tard que les langages réguliers sont clos (fermés) par ces opérations. Autrement dit, nous allons construire des AEFs acceptant : le langage complémentaire, celui de l'entrelacement, le produit des AEFs acceptant l'intersection entre deux langages et celui du langage miroir.

Remarque 5.4.1

Comme c'est le cas avec les différents types de langages, ces opérations ne concernent pas exclusivement les AEFs, elles peuvent plus ou moins être étendues aux types d'automates. Cependant, la complexité de ces opérations augmente inversement avec le type de l'automate.

5.4.1 Compléter un AEF

➊ Définition 5.4.1 AEF complet

Un AEF déterministe est complet si et seulement si $\forall(s, a) \in Q \times X, \exists s' \in Q$, tel que $\delta(s, a) = s'$.

? **Question.** Si l'AEF déterministe ne vérifie pas cette définition (avec des transitions manquantes), comment peut-on le rendre complet ?

► La construction d'une version complète d'un AEF (qui doit être déterministe) passe par les étapes suivantes :

1. Rajouter un nouvel état N ($Q \cup \{N\}$) C'est un état puits absorbant les transitions manquantes de l'automate original, sans pour autant altérer le langage reconnu.
2. $\forall s \in Q, \forall a \in X$, ajouter toutes les transitions manquantes $\delta(s, a)$ vers N : $\delta(s, a) = N$
3. En ce qui concerne l'état N , toutes les transition sortantes de N revient au même état : $\forall a \in X, \delta(N, a) = N$.

💡 Remarque 5.4.2

Cette notion peut être abordée aussi lorsque l'AEF est non-déterministe. Mais en ce concerne ce cours, l'AEF doit être déterministe pour vérifier s'il est complet ou non.

💡 Déduction 5.4.1 Automate canonique

Si l'AEF A acceptant un langage L est déterministe, complet et minimal, alors A est appelé l'automate canonique de L (A est unique).

5.4.2 Le complément d'un AEF

➊ Définition 5.4.2

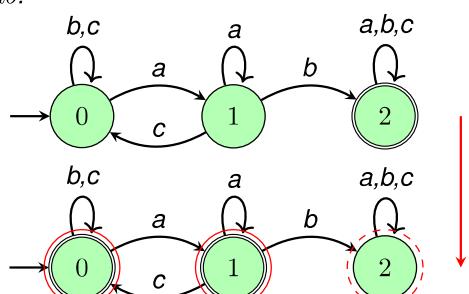
Le complément d'un AEF $A(X, Q, s_0, F, \delta)$ déterministe et complet, acceptant un langage L est un AEF acceptant le langage complémentaire (langage inverse) $\bar{L} = X^* - L$ et qui est défini par $A(X, Q, s_0, Q - F, \delta)$.

► Pour obtenir le complément de A :

- Il faut que A soit déterministe (sinon il faut le déterminiser)
- Il faut que A soit complet (sinon il faut le compléter).
- Il faut changer les états finaux en états non finaux et vice-versa.

📋 Exemple Le complément d'un AEF (déterministe et complet)

Soit $A(\{a, b, c\}, \{0, 1, 2\}, 0, \{2\}, \delta)$ l'AEF acceptant le langage ($L(A)$) des mots contenant le facteur ab .



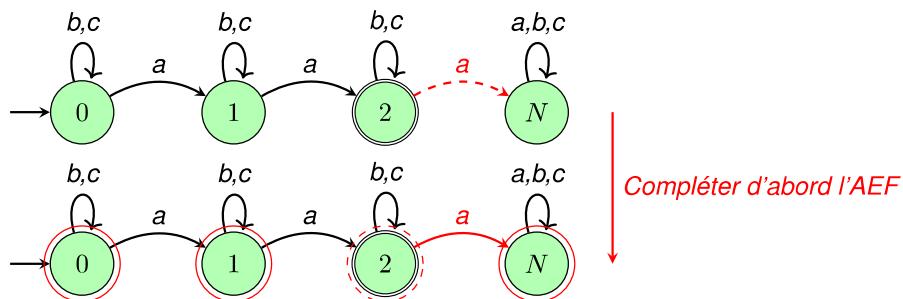
Les états finaux deviennent non finaux (et vice-versa)

A est déterministe et complet. Alors, nous appliquons la définition précédente pour trouver l'AEF ac-

ceptant \bar{L} : les mots qui ne contiennent pas le facteur ab (Le langage complémentaire). Le nouvel AEF est défini par $A(\{a, b, c\}, \{0, 1, 2\}, 0, \{0, 1\}, \delta)$.

Exemple Le complément d'un AEF déterministe, mais pas complet

L'AEF suivant accepte le langage L celui des mots définis sur $\{a, b, c\}$ contenant exactement deux a



Mais cet AEF n'est pas complet !

Donc avant d'appliquer la définition précédente, il faut d'abord compléter l'AEF (le transformer en un AEF complet) en ajoutant un état non final N , appelé aussi un état d'erreur (état puits). Car une fois qu'on lit le troisième a , on ne peut pas reconnaître le mot.

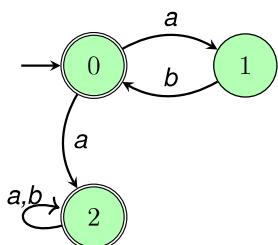
On voit bien que le résultat final donne un AEF qui accepte tous les mots sur $\{a, b, c\}$ ne contenant pas 2 a .

Compléter d'abord l'AEF

Remarque 5.4.3

Si on n'avait pas complété l'AEF donné dans l'exemple précédent, on aurait obtenu un AEF acceptant un langage différent (les mots contenant au plus deux a) de celui du langage complémentaire cherché. Par exemple on n'aurait pas accepté un mot ayant trois a . Or, ce mot fait partie du langage complémentaire.

Exemple Le complément d'un AEF non-déterministe ?



On remarque ici que cet AEF n'est pas déterministe. !

On ne peut donc pas appliquer la transformation pour trouver l'AEF du langage complémentaire. Il faut d'abord procéder à sa déterminisation (??).

5.4.3 L'AEF de l'entrelacement

DEFINITION 5.4.3 AEF acceptant l'entrelacement d'un langage avec un symbole

Soit L un langage accepté par un AEF $A(X, Q, s_0, F, \delta)$. L'entrelacement $L(A) \sqcup \alpha$ (α est un symbole) donne un langage qui est accepté par l'AEF A'' que l'on peut définir selon les étapes suivantes :

- 1 Créer une copie de A notée $A'(X, Q', s'_0, F', \delta')$
 - Q' est une copie de l'ensemble des états Q .
 - s'_0 est une copie de s_0 .
 - $F' \subset Q'$ une copie des états finaux de F .

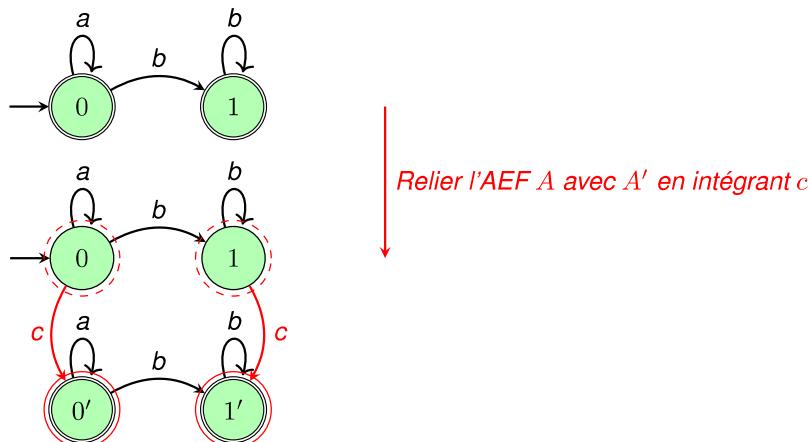
- La fonction de transition δ' est définie par :
S'il existe une transition $\delta(s_i, a) = s_j$ dans A , alors on ajoute sa copie $\delta'(s'_i, a) = s'_j$ dans A' .

2 Définir l'AEF acceptant $A''(X \cup \{\alpha\}, Q'', s_0, F', \delta'')$ à partir de A et A' tel que

- L'alphabet sera $X \cup \{\alpha\}$.
- $Q'' = Q \cup Q'$.
- l'état initial sera s_0 (l'état initial de A).
- F' : Les états finaux sont les états finaux de la copie A' .
- δ'' : Les transitions de A'' vont inclure :
 - Les transitions de A tel que $\delta''(s_i, a) = \delta(s_i, a), s_i \in Q$.
 - Les transitions de A' tel que $\delta''(s'_i, a) = \delta'(s'_i, a), s'_i \in Q'$.
 - Depuis chaque état s_i dans A , on ajoute une transition décorée par α vers son homologue s'_i dans $A' : \delta''(s_i, \alpha) = s'_i$

💡 Exemple Construire l'AEF de l'entrelacement

L'exemple suivant montre comment construire l'AEF acceptant $L \sqcup c$.



Après avoir créé une copie de l'AEF acceptant $L = \{a^n b^m \mid n, m \geq 0\}$, nous avons ajouté des arcs en rouge pour relier chaque état du premier automate avec son homologue du deuxième automate. L'état initial est celui du premier automate (0) et les états finaux sont ceux du deuxième automates $\{0', 1'\}$

💡 Remarque 5.4.4

- Nous avons étudié un cas simple de l'entrelacement d'un langage avec un seul symbole.
- En élargissant l'étendue de cette opération, la question qui se pose est la suivante :

? **Question.** Comment peut-on construire l'AEF acceptant $L(A) \sqcup w$ tel que $w = \alpha_1 \alpha_2 \dots \alpha_n$?

- Si $\alpha_1 = \alpha_2 = \dots = \alpha_n$. C'est-à-dire $w \in \{\alpha\}^*$, donc $w = \alpha^n, n \geq 0$, alors il s'agit d'un entrelacement entre un langage et un mot formé par le même symbole. La réponse ici est basée sur la même idée précédente, mais au lieu de créer une seule copie, on crée n copies (A_1, A_2, \dots, A_n) de A . L'état initial est celui de A_1 et les états finaux sont ceux de A_n . Les nouvelles transitions ajoutées (décorées par α) vont relier chaque état de A_i et son homologue dans A_{i+1} .
- S'il y a au moins un symbole différent dans le mot w cette construction ne fonctionnera pas car elle ne respecte pas l'ordre des symboles $\alpha_1, \alpha_2, \dots, \alpha_n$

5.4.4 Produit de deux AEFs

❶ Définition 5.4.4

Le produit de deux AEFs $A(X, Q, s_0 F, \delta)$ et $A'(X', Q', s'_0 F', \delta')$ acceptant respectivement les deux langages L et L' donne un AEF noté $A''(X'', Q'', s''_0 F'', \delta'')$ et qui accepte $L(A) \cap L'(A')$. Pour définir A'' :

- $X'' = X \cap X'$
- $Q'' = Q \times Q'$: Les états de A'' sont des paires (s, s') , $s \in Q, s' \in Q'$.
- $s''_0 = (s_0, s'_0)$
- $F'' = F \times F'$: Un état final est toute paire dont la première et la deuxième composante sont des états finaux dans A et A' respectivement.
- $\delta''((s, s'), a) = \delta(s, a) \times \delta'(s', a)$ Sachant que $\delta(s, a)$ et $\delta'(s', a)$ peuvent produire chacun un ensemble d'états (Si A ou A' est non-déterministe).

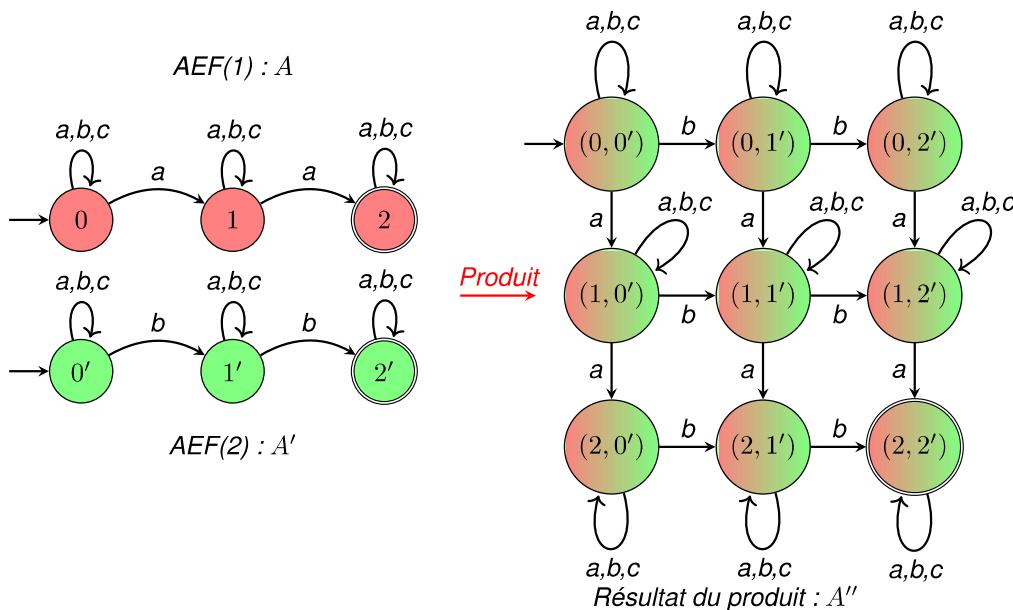
❷ Déduction 5.4.2

$(L \cap L')(A'')$ signifie que A'' accepte $L \cap L'$. On sait que si $w \in L(A) \cap L'(A')$ alors :

- $w \in L(A)$ et donc il est accepté par A .
- $w \in L'(A')$ et donc il est accepté par A' .

► Donc on peut déduire que si w est accepté par A'' , alors il est accepté à la fois par A et A'

❸ Exemple de Produit de 2 AEFs



Dans cet exemple :

- L'AEF $A(\{a, b, c\}, \{0, 1, 2\}, 0, \{2\}, \delta)$ accepte le langage L des mots contenant au moins deux a (A est non-déterministe).
- L'AEF $A'(\{a, b, c\}, \{0', 1', 2'\}, 0', \{2'\}, \delta')$ accepte le langage L' des mots contenant au moins deux b (A' est non-déterministe).
- En appliquant le produit $A \times A'$, on obtient un AEF A'' qui accepte les mots contenant au moins deux a et au moins deux b et cela correspond exactement au langage $L'' = L \cap L'$.

► On remarque que tout chemin dans A'' qui part de l'état initial vers l'état final (chemin d'acceptation) doit passer forcément par deux a et deux b (l'ordre n'est pas important).

5.4.5 L'AEF acceptant le langage miroir

❶ Définition 5.4.5 L'AEF acceptant L^R

Si l'AEF $A(X, Q, s_0, F, \delta)$ accepte le langage L , alors l'AEF qui accepte le langage miroir L^R est défini par $A^R(X, Q, s_0^R, F^R, \delta^R)$ tel que :

- X reste le même.
- Q : On garde les mêmes états (avec la possibilité d'ajouter un nouvel état initial, pour en savoir plus regardez les éléments suivants).
- L'état initial s_0^R :
 - Si $F = \{s_f\}$ contient un seul état final alors ce dernier représentera l'état initial de A^R ($s_0^R = s_f$).
 - Sinon, s'il y a plusieurs états finaux dans A cela ne devrait pas donner lieu à des multiples états initiaux dans A^R !
Dans une telle situation, il suffit d'ajouter un nouvel état initial s_0^R et de le raccorder aux anciens états finaux par des ε -transitions. Ce qui signifie que A^R est dans ce cas non-déterministe.
- $F^R = \{s_0\}$ l'état initial de A devient l'état final de A^R .
- δ^R : Les transitions de A^R sont déduites celles de A mais inversées, tel que : si $\delta(s, a) = s'$ alors $\delta^R(s', a) = s$ (Il suffit juste d'inverser les sens des arcs de A .)



Déduction 5.4.3

Si A est déterministe, la construction de A^R n'aboutit pas nécessairement à un automate déterministe.

❷ Remarque 5.4.5 Deuxième méthode de minimisation basée sur A^R

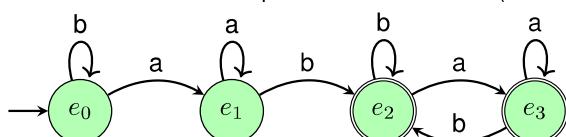
La construction de l'AEF A^R acceptant le langage miroir de $L(A)$ peut servir aussi à minimiser A mais selon les étapes suivantes :

1. Construire A^R acceptant $(L(A))^R$.
2. Si A^R est non-déterministe alors :
 - (a) Déterminiser A^R , ce qui donne A_{det}^R .
 - (b) Construire le miroir de A_{det}^R , ce qui donne A' ($A' = (A_{det}^R)^R$).
 - (c) Déterminiser A' . Le résultat A'_{det} représente l'AEF déterministe et minimal acceptant le langage $L(A)$.
3. Sinon (C'est le cas où A n'a qu'un seul état final), cette méthode ne fonctionne pas.

5.5

Simulation des AEFs

Les automates finis sont des modèles abstraits puissants dont la simulation est très importante pour mieux comprendre leur comportements, ainsi que dans de nombreuses applications pratiques. La simulation d'un AEF consiste à implémenter un programme (en Python) simulant les composants et le fonctionnement de cet automate. Il existe diverses approches pour implementer des automates finis. Nous allons étudier quelques unes en utilisant le langage de programmation Python, en s'appuyant sur un exemple d'AEF déterministe (pas forcément minimal). Reprenons l'exemple d'un automate A acceptant les mots contenant le facteur ab et que nous avons obtenu après déterminisation (dans la partie déterminisation).



5.5.1 Simulation classique

Simuler les composants

Il est possible de simuler les composants de l'automate, avec une fonction qui fait déplacer la tête de lecture sur le ruban :

- Un ruban : représenté par une chaîne (tableau) de caractères (word).
- Une tête de lecture représentée par un entier (head), initialisé à 0 (le premier symbole de word).
- Déplacement : La fonction next() renvoie à chaque étape un caractère et passe à la case suivante. À la fin du mot, elle renvoie 0.

💡 Simulation des composants (en Python)

```

1 head = 0 # Tete de lecture positionnee sur le premier symbole.
2 def next(): # Fonction qui simule le deplacement de la tete.
3     global head
4     if head < len(word):
5         result = word[head]
6         head += 1
7         return result
8     else:
9         return 0

```

En se basant sur cette fonction, nous allons implémenter les transitions définissant le comportement de l'automate :

💡 Fonctionnement d'un AEF basé sur la fonction next()

```

1 current_state = "e0" # Partant de l'état initial
2 F = ["e2", "e3"] # les états finaux
3 char = next()
4 go = 1 # il y a une transition
5 while go and char:
6     if current_state == "e0": # Les transitions en partant de e0
7         if char == 'a':
8             current_state = "e1" # Faire une transition
9         elif char == 'b':
10            ....
11        else: # si on lit un autre symbole
12            go = 0 # Il n'y a plus de transition
13
14    elif current_state == "e1": # Les transitions en partant de e1
15        if char == 'a':
16            current_state = "e1"
17        ....
18        else:
19            go = 0 # Il n'y a plus de transition
20
21    #... completer le reste des transitions
22
23    if go: # Tant qu'il y a une transition , on va lire le symbole suivant
24        char = next()
25
26    # Si les conditions d'acceptation sont vérifiées
27    if char == 0 and current_state in F:
28        print("Accepted")
29    else:
30        print("Rejected")

```

Cette approche est plus modulaire. Avec une fonction distincte, on peut obtenir le prochain caractère, ce qui peut rendre le code plus lisible et parfaitement en adéquation avec la définition de l'acceptation d'un mot.

Simulation directe

Cette approche est similaire dans le sens où on va parcourir l'entrée itérativement et faire des transitions en fonction du caractère actuel. Sauf que l'implémentation est plus directe et intuitive

💡 Simulation directe d'un AEF en Python

```

1 def simulate_fsa(word):
2     current_state = "e0" # Partant de l'état initial
3     F = ["e2", "e3"]
4     for char in word:
5         if current_state == "e0":
6             if char == "a":
7                 current_state = "e1"
8             elif char == 'b':
9                 #.....
10            #.....
11            else:
12                return False # Rejected
13            elif current_state == "e1"
14                if char == 'a':
15                    current_state = "e1"
16                #.....
17                else:
18                    return False # Rejected
19                #....
20            # En sortant de la boucle veut dire que le mot est entièrement lu
21            if current_state in F:
22                return True # Accepted
23            else:
24                return False # Rejected

```

Il est vrai que ce type de simulations est plus intuitive, mais cela peut devenir complexe et difficile à gérer à mesure que l'automate devient plus grand.

💡 Remarque 5.5.1 Simulation récursive

- Il faut noter que les exemples d'implémentation présentés ci-dessus permettent de parcourir le mot itérativement. Une autre alternative consiste à simuler le fonctionnement d'un AEF récursivement, d'autant plus que le fonctionnement des AEFs est intrinsèquement récursive.
- L'idée classique derrière la simulation récursive d'un AEF consiste à diviser le problème d'analyse d'un mot en sous-problèmes, en effectuant une transition à chaque étape de la récursion, en lisant un symbole de l'entrée. La condition d'arrêt est définie par la condition d'acceptation (ou de rejet) d'un mot par AEF (à aborder en TP).
- Il existe par ailleurs d'autres implémentations récursives qui peuvent par exemple simplifier la simulation des AEFs non-déterministes : Il est possible par exemple d'associer à chaque état une fonction, et d'appeler ces fonctions (d'une manière récursive) suivant les transitions de l'AEF

5.5.2 Simulation avec les dictionnaires

L'une des approches les plus efficaces et lisibles est celle basée sur les dictionnaires (en Python). Il ne s'agit pas ici de simuler explicitement les composants d'un AEF. On va utiliser plutôt les dictionnaires comme une structure de données pour définir exactement l'AEF selon sa définition formelle vue précédemment (en quintuplet)

💡 Définition d'un AEF avec un dictionnaire (en Python)

```

1 AEF = {
2     "alphabet": [ "a", "b" ],
3

```

```

4   "states": ["e0", "e1", "e2", "e3"],
5   "initial_state": "e0",
6   "final_states": ["e2", "e3"],
7   "transitions": { "e0": { "a": "e1", "b": "e0" },
8                   "e1": { "a": "e1", "b": "e2" },
9                   "e2": { "a": "e3", "b": "e2" },
10                  "e3": { "a": "e3", "b": "e2" }
11                 }
12 }
```

On voit bien que les dictionnaires (emboîtés) permettent une meilleur structuration des transitions. Par ailleurs, il suffit d'ajouter des entrées au dictionnaire pour une éventuelle extension de l'automate (en ajoutant de nouveaux états ou de nouvelles transitions). Cependant, pour simuler le fonctionnement de l'automate, un programme (une fonction) sera nécessaire pour exploiter cette structure.

💡 Simulation d'un AEF basé sur les dictionnaires

```

1 def simulate_fsa_with_dict(dfa, word):
2     # Recuperer l'état initial de l'AEF
3     current_state = dfa["initial_state"]
4     # Parcourir le mot
5     for char in word:
6         # Verifie s'il existe une transition (current_state, char)
7         if char not in dfa["transitions"][current_state]:
8             return False # Rejected
9         current_state = dfa["transitions"][current_state][char]
10
11    # Vérifier si on a atteint l'état final
12    if current_state in dfa["final_states"]:
13        return True # Accepted
14    else:
15        return False # Rejected
```

💡 Remarque 5.5.2

- Il est également possible de simuler récursivement le fonctionnement d'un AEF défini avec des dictionnaires.
- On peut penser aussi à une implémentation orientée objet où il est possible de représenter un automate fini sous forme d'une classe. Cette dernière permet d'encapsuler à la fois toutes les données de l'automate : alphabet, états, ..., transitions (avec des dictionnaires) et des méthodes associées pour simuler son fonctionnement d'analyse.
- Il existe aussi des bibliothèques python comme :
 - 'Automata' qui propose des structures et des algorithmes permettant de manipuler des automates finis (déterministes et non-déterministes), des automates à pile et des machines de Turing. Ses algorithmes ont été optimisés pour traiter des entrées importantes.
 - 'Visual Automata' est une bibliothèque Python 3 conçue comme une surcouche (wrapper) de la bibliothèque 'Automata', ajoutant des fonctionnalités de visualisation supplémentaires.

TABLE 5.1 – Types de simulation qui nous intéressent dans ce cours

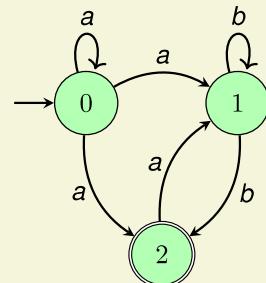
Traitement	Simulation	Classique (Sans dict)	Avec dict
Itératif	Simuler les composants	✓	✗
	Directe	✓	✓
Récursif	Directe	✓	✓
	Modulaires (fonction pour chaque état)	✓	✗

5.6

Série d'exercices de TD N°3

**Exercice 1 : Déterminisation, Complexité d'analyse et Minimisation**

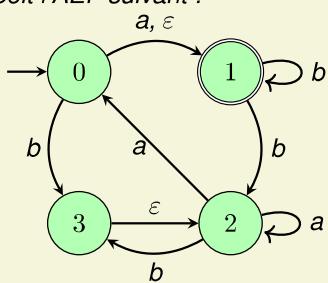
Soit l'AEF suivant qui modélise un petit jeu dont les étapes sont $\{0, 1, 2\}$. Un joueur doit effectuer une série de mouvements (une stratégie) définie sur $\{a, b\}$ afin d'atteindre l'objectif : 2 (gagner le jeu)



1. Analysez les mots (stratégies) : abb, baa , en explorant toutes les séquences de configurations.
2. Pourquoi cet AEF est-il non-déterministe ? Déterminez-le et réanalyser les mots précédents.
3. Minimisez l'AEF déterministe obtenu.

**Exercice 2 : Déterminisation (avec les ε -transitions), Complexité d'analyse et Minimisation**

Soit l'AEF suivant :



1. Analysez le mot : baa , en explorant toutes les séquences de configurations possibles.
2. Pourquoi cet AEF est-il non-déterministe ? Déterminez-le et réanalyser le mot précédent.
3. Minimisez l'AEF déterministe obtenu.

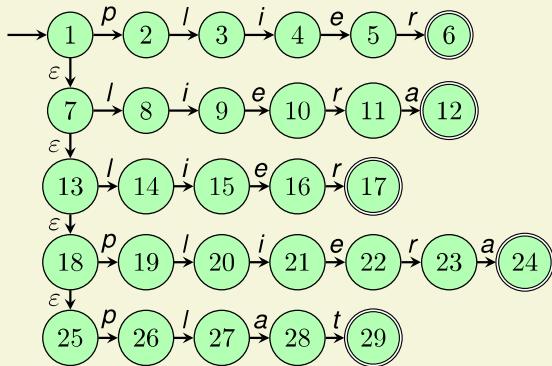
**Exercice 3 : Minimisation**

Minimiser l'AEF suivant $A(\{a, b, c\}, \{1, 2, 3, 4, 5, 6, 7, 8, 9\}, 1, \{3, 6, 8\}, \delta)$ tel que δ est défini par la table suivante :

Etat	a	b	c
1	2	3	4
2	1	5	6
3	1	5	6
4	2	6	1
5	4	7	8
6	4	5	3
7	4	5	3
8	9	3	6
9	7	3	9

**Exercice Application de 1,2,3 : L'automate d'un dictionnaire de mots**

L'AEF suivant est conçu pour chercher des mots dans un petit dictionnaire (Une liste finie de mots).



1. Donnez le langage accepté par cet AEF et déterminisez-le.
2. Identifiez une paire d'états distinguables et des paires d'états indistinguables. Minimisez l'AEF déterministe.

Exercice 4 : Construire des AEFs, en cherchant des facteurs, en appliquant des opérations sur les AEFs

Donnez les AEFs qui acceptent les langages suivants :

1. Tous les mots sur $\{a, b, c\}$.
2. Tous les mots sur $\{a, b, c\}$ qui se terminent avec un symbole différent de celui du début.
3. L'ensemble des mots $\{w = uabv \mid u, v \in \{a, b, c\}^*\}$
4. Tous les mots sur $\{a, b, c\}$ qui ne contiennent pas le facteur aba.
5. Tous les mots sur $\{a, b, c\}$ qui ne contiennent pas le facteur aac.
6. Tous les mots sur $\{a, b, c\}$ qui ne contiennent ni le facteur aba ni le facteur aac.
7. Tous les mots w définis sur $\{a, b, c\}$ tels que toutes les occurrences possibles de c dans w précèdent celles de b tout en ayant deux a.

Exercice 5 : Constructions des AEFs,

1. Donnez un AEF déterministe qui accepte tous les mots sur $\{a, b\}$ qui contiennent un nombre pair de a. Ensuite, donnez la séquence d'analyse de chacun des mots suivants : suivants : bb, aa, baa, baba, ab, aba avec l'AEF proposé.
2. Donnez l'AEF déterministe qui accepte le langage des mots $\{w \in \{a, b\}^* \mid |w|_a = 2n \wedge |w|_b = 2m, n, m \geq 0\}$. Ensuite dite si les mots suivants : aabb, abab, baba, baa appartiennent ou non à ce langage. Justifiez.

Exercice 6 : Divisibilité d'un entier

1. Donnez deux versions d'un AEF acceptant les représentations binaires des entiers multiples de 2 et montrez la différence entre les deux.
2. Déduisez l'AEF capable de reconnaître les nombres en base 10 divisibles par 2.
3. Donnez l'AEF qui accepte les représentations binaires des entiers divisibles par 4 et dites s'il est minimal ou non ?
4. Selon les constructions précédentes, montrez que le nombre minimum d'états d'un AEF acceptant les nombres binaires divisibles par 2^m , $m > 0$ n'est que $m + 1$.

Exercice 7 : Divisibilité d'un entier, simulation de lecture des chiffres sur les états

En simulant la lectures des 0 et 1 sur les états :

1. Donnez l'AEF qui accepte les représentations binaires des entiers multiples de 3.
2. Construisez l'AEF qui accepte les nombres binaires multiples de 6. Y a-t-il une autre méthode permettant de construire le même AEF ?

Exercice 8 : Nombres réels en langage C

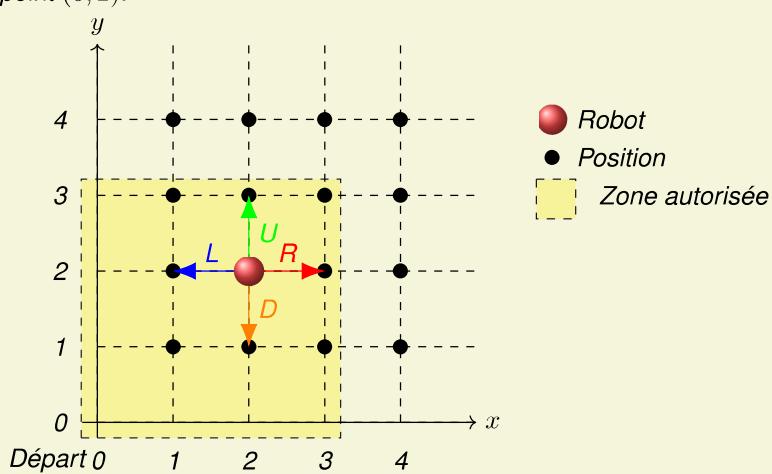
Donnez un automate déterministe acceptant la notation décimale et scientifique des nombres réels en langage C (et non pas des simples entiers).

Exercice 9 : Format d'une date

Proposez un AEF capable de reconnaître la forme suivante d'une date (jour/mois) pendant l'année courante : dd/mm, et rejeter les dates invalides comme par exemple 30/02 (la date 29/02 sera acceptée)

Exercice 10 : Les déplacements d'un Robot de tri

Supposons que nous avons un robot de tri automatisé d'Amazon qui se déplace dans un entrepôt représenté par une grille limitée sous forme de matrice : $\{0, 1, 2, 3\} \times \{0, 1, 2, 3\}$. Le point de départ c'est le point $(0, 0)$. Pour se déplacer, le robot reçoit des instructions de déplacement sous forme de mots (strings) composés de lettres $\{L, R, U, D\}$. Chacune de ses lettres représente un mouvement (d'une seule unité) que le robot peut effectuer sur la grille : L (gauche), R (droite), U (haut), D (bas). Par exemple, en lisant le mot RUUL, le robot qui se trouve sur le point $(0, 0)$ va se déplacer vers le point $(0, 2)$.



Selon la tâche que l'on veut effectuer, nous voulons définir un ensemble de déplacements autorisés qui représente un langage de déplacement. Donnez si c'est possible l'AEF capable d'accepter chacun des langages de déplacement suivants :

1. Tout chemin de déplacement assurant que le robot reste dans la zone autorisée.
2. Tout chemin de déplacement qui ne fait pas entrer le robot dans la zone $\{1, 2\} \times \{1, 2\}$ (une zone de stockage de produits triés, impénétrable par le robot)
3. Les déplacements qui font revenir le robot vers le point de départ à la fin d'une tâche accomplie

5.7**Série de TP N°3**

L'objectif de cette série d'exercices de travaux pratiques est de définir et simuler le fonctionnement des automates à états finis (AEFs). Nous aurons besoins de :

 **Prérequis**

- Des notions sur la définition, le fonctionnement, la simulation des AEFs vus dans ce chapitre (cours/TD)
- Des rappels sur les définitions et la manipulation des dictionnaires en Python.

 Quelques notions sur: Les dictionnaires en Python

- Un dictionnaire est une structure de données qui stocke des valeurs associées à des clés sous forme de paires clé : valeur. Les types des objets clés sont des types non-mutables (un entier, une chaîne de caractères, un tuple,... mais pas une liste).
- Création d'un dictionnaire :

```
1| my_dict = { "key1": "value1", "key2": "value2", "key3": "value3"}
```

On peut créer aussi des dictionnaires vides :

```
1| my_dict2= {} # Ne pas confondre avec la declaration des sets
2| my_dict3 = dict()
```

- Accès aux éléments :

Les dictionnaires offrent un accès rapide aux valeurs. L'accès à une valeur se fait en utilisant la clé correspondante :

```
1| get_value = my_dict[ "key1" ] # --> cela donne: value1
```

Par contre, si on utilise une clé qui n'existe pas, on obtient l'erreur de type `KeyError`.

- Manipulation des dictionnaires :

Modification d'une valeur associée à une clé :

```
1| my_dict[ "key1" ] = "new_value"
```

Ajout d'une nouvelle paire clé-valeur :

```
1| my_dict[ "key4" ] = "value4"
```

Suppression d'une paire clé-valeur :

```
1| del my_dict[ "key4" ]
```

- Programmer avec les dictionnaires :

Vérification de l'existence d'une clé (Mais cela ne permet pas de tester l'existence d'une valeur)

```
1| if "key1" in my_dict # --> True
```

Parcourir les clés et les valeurs d'un dictionnaire :

```
1| for k, v in my_dict.items():
2|     print(k, ":", v)
```

Parcourir seulement les clés d'un dictionnaire (en accédant aux valeurs) :

```
1| for k in my_dict:
2|     print(my_dict[k])
```

Parcourir seulement les valeurs d'un dictionnaire :

```
1| for v in my_dict.values():
2|     print(v)
```