Théorie des Langages 1

Cours 6 : Preuves, implémentation et applications

L. Rieg

Grenoble INP - Ensimag, 1re année

Année 2022-2023

Rappels sur l'élimination des ε-transistions

1. Calculer $\mathrm{Acc}_\varepsilon(p)$, les états accessibles par ε-transistions
   $\rightsquigarrow$ par itération (cf. cours 1)

2. Construire un automate B équivalent sans ε-transition

$$\underbrace{p \xrightarrow{a} q}_{\boxed{B}} \iff \exists\, r,\ \underbrace{p \overset{\varepsilon}{\rightsquigarrow} r}_{\boxed{A}} \underbrace{\xrightarrow{a} q}_{\boxed{A}}$$

- $(p, a, q) \in \delta' \iff a \neq \varepsilon$ et $\exists r \in \mathrm{Acc}_\varepsilon(p)$ tel que $(r, a, q) \in \delta$
- $F' = \{p \in Q \mid \mathrm{Acc}_\varepsilon(p) \cap F \neq \emptyset\}$

Remarques

- *Même Q, V et I, seuls δ et F changent*
- *Par construction, B est sans ε-transition*

Correction de l'élimination des ε-transistions

Théorème

$\forall$ *automate* A*, l'automate* B *défini précédemment est équivalent à* A*.*

Correction de l'élimination des ε-transistions

Théorème

∀ *automate* A*, l'automate* B *défini précédemment est équivalent à* A*.*

Lemme intermédiaire : caractérisation des chemins

L'automate B vérifie la propriété suivante :

Il existe un chemin de p à q de trace w dans A
si et seulement si
il existe $r \in Q$ tel qu'il existe un chemin de p à r de trace w dans B
et un chemin de r à q de trace ε dans A.

Correction de l'élimination des ε-transistions

Théorème

$\forall$ *automate* A, *l'automate* B *défini précédemment est équivalent à* A.

Lemme intermédiaire : caractérisation des chemins

L'automate B vérifie la propriété suivante :

Il existe un chemin de p à q de trace w dans A
si et seulement si
il existe $r \in Q$ tel qu'il existe un chemin de p à r de trace w dans B
et un chemin de r à q de trace ε dans A.

Graphiquement : $\underset{\boxed{A}}{p \overset{w}{\rightsquigarrow} q} \iff \exists r, \underset{\boxed{B}}{p \overset{w}{\rightsquigarrow} r}$ et $\underset{\boxed{A}}{r \overset{\varepsilon}{\rightsquigarrow} q}$

Correction de l'élimination des ε-transistions

Théorème

$\forall$ *automate* A, *l'automate* B *défini précédemment est équivalent à* A.

Lemme intermédiaire : caractérisation des chemins

L'automate B vérifie la propriété suivante :

Il existe un chemin de p à q de trace w dans A
si et seulement si
il existe $r \in Q$ tel qu'il existe un chemin de p à r de trace w dans B
et un chemin de r à q de trace ε dans A.

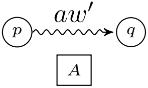
Graphiquement : $\underset{\boxed{A}}{p \overset{w}{\rightsquigarrow} q} \iff \exists r, \underset{\boxed{B}}{p \overset{w}{\rightsquigarrow} r}$ et $\underset{\boxed{A}}{r \overset{\varepsilon}{\rightsquigarrow} q}$

Preuve par induction sur w.

- Base : $w = \varepsilon$. Il suffit de prendre $r \overset{\text{def}}{=} p$.

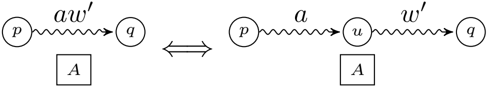
Preuve par induction, suite

- Induction : $w = aw' \quad (a \in V)$



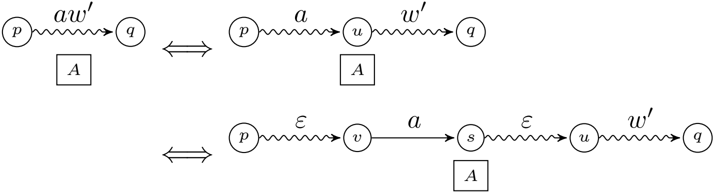
Preuve par induction, suite

- Induction : $w = aw' \quad (a \in V)$



Preuve par induction, suite

- Induction : $w = aw' \quad (a \in V)$



Preuve par induction, suite

- Induction : $w = aw' \quad (a \in V)$

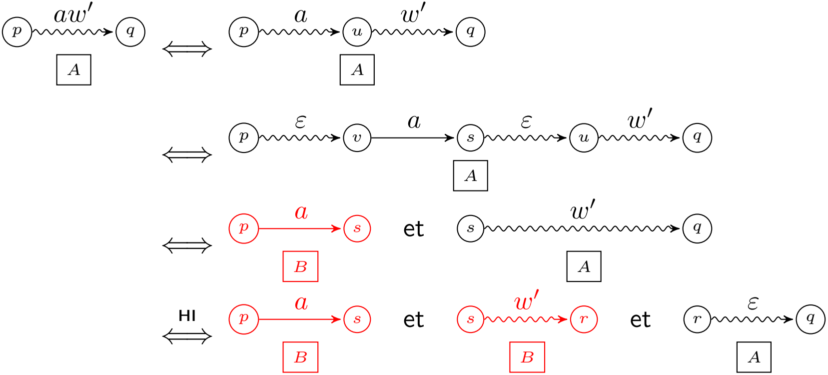
$$\underset{\boxed{A}}{p \overset{aw'}{\rightsquigarrow} q} \iff \underset{\boxed{A}}{p \overset{a}{\rightsquigarrow} u \overset{w'}{\rightsquigarrow} q}$$

$$\iff \underset{\boxed{A}}{p \overset{\varepsilon}{\rightsquigarrow} v \xrightarrow{a} s \overset{\varepsilon}{\rightsquigarrow} u \overset{w'}{\rightsquigarrow} q}$$

$$\iff \underset{\boxed{B}}{p \xrightarrow{a} s} \quad \text{et} \quad \underset{\boxed{A}}{s \overset{w'}{\rightsquigarrow} q}$$

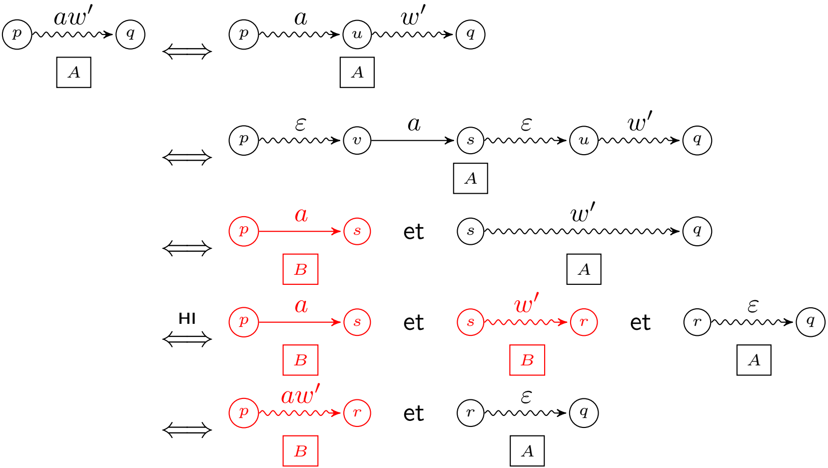
Preuve par induction, suite

- Induction : $w = aw' \quad (a \in V)$



Preuve par induction, suite

- Induction : $w = aw' \quad (a \in V)$



Preuve du théorème

$w \in \mathcal{L}(A) \iff \exists$ un chemin de $q_0 \in I$ à $q_f \in F$ de trace w dans A

Preuve du théorème

$$
\begin{aligned}
w \in \mathcal{L}(A) \quad & \iff \quad \exists \text{ un chemin de } q_0 \in I \text{ à } q_f \in F \text{ de trace } w \text{ dans } A \\
& \iff \quad \exists r \in Q, \exists \text{ un chemin de } q_0 \in I \text{ à } r \text{ de trace } w \text{ dans } B \\
& \qquad\qquad \text{et } \exists \text{ un chemin de } r \text{ à } q_f \in F \text{ de trace } \varepsilon \text{ dans } A
\end{aligned}
$$

Preuve du théorème

$$
\begin{aligned}
w \in \mathcal{L}(A) \quad &\iff \quad \exists \text{ un chemin de } q_0 \in I \text{ à } q_f \in F \text{ de trace } w \text{ dans } A \\
&\iff \quad \exists r \in Q, \exists \text{ un chemin de } q_0 \in I \text{ à } r \text{ de trace } w \text{ dans } B \\
&\qquad\quad \text{et } \exists \text{ un chemin de } r \text{ à } q_f \in F \text{ de trace } \varepsilon \text{ dans } A
\end{aligned}
$$

Preuve du théorème

$$
\begin{aligned}
w \in \mathcal{L}(A) \quad &\iff \quad \exists \text{ un chemin de } q_0 \in I \text{ à } q_f \in F \text{ de trace } w \text{ dans } A \\
&\iff \quad \exists r \in Q, \exists \text{ un chemin de } q_0 \in I \text{ à } r \text{ de trace } w \text{ dans } B \\
&\qquad\quad \text{et } \exists \text{ un chemin de } r \text{ à } q_f \in F \text{ de trace } \varepsilon \text{ dans } A \\
&\iff \quad \exists r \in Q, \exists \text{ un chemin de } q_0 \in I \text{ à } r \text{ de trace } w \text{ dans } B \\
&\qquad\quad \text{et } r \in F'
\end{aligned}
$$

Preuve du théorème

$w \in \mathcal{L}(A) \iff$ $\exists$ un chemin de $q_0 \in I$ à $q_f \in F$ de trace w dans A

$\iff$ $\exists r \in Q$, $\exists$ un chemin de $q_0 \in I$ à r de trace w dans B
et $\exists$ un chemin de r à $q_f \in F$ de trace ε dans A

$\iff$ $\exists r \in Q$, $\exists$ un chemin de $q_0 \in I$ à r de trace w dans B
et $r \in F'$

Preuve du théorème

$$
\begin{aligned}
w \in \mathcal{L}(A) \quad & \iff \quad \exists \text{ un chemin de } q_0 \in I \text{ à } q_f \in F \text{ de trace } w \text{ dans } A \\
& \iff \quad \exists r \in Q, \exists \text{ un chemin de } q_0 \in I \text{ à } r \text{ de trace } w \text{ dans } B \\
& \qquad\quad \text{et } \exists \text{ un chemin de } r \text{ à } q_f \in F \text{ de trace } \varepsilon \text{ dans } A \\
& \iff \quad \exists r \in Q, \exists \text{ un chemin de } q_0 \in I \text{ à } r \text{ de trace } w \text{ dans } B \\
& \qquad\quad \text{et } r \in F' \\
& \iff \quad w \in \mathcal{L}(B)
\end{aligned}
$$

Preuve du théorème

$$
\begin{aligned}
w \in \mathcal{L}(A) \quad & \iff \quad \exists \text{ un chemin de } q_0 \in I \text{ à } q_f \in F \text{ de trace } w \text{ dans } A \\
& \iff \quad \exists r \in Q, \exists \text{ un chemin de } q_0 \in I \text{ à } r \text{ de trace } w \text{ dans } B \\
& \qquad\quad \text{et } \exists \text{ un chemin de } r \text{ à } q_f \in F \text{ de trace } \varepsilon \text{ dans } A \\
& \iff \quad \exists r \in Q, \exists \text{ un chemin de } q_0 \in I \text{ à } r \text{ de trace } w \text{ dans } B \\
& \qquad\quad \text{et } r \in F' \\
& \iff \quad w \in \mathcal{L}(B)
\end{aligned}
$$

Les automates A et B sont bien équivalents.

Rappels sur la déterminisation

Idée : suivre tous les chemins en parallèle

- Entrée : un automate $A = \langle Q, V, \delta_A, I, F_A \rangle$ sans ε-transistions
- Sortie : un automate B déterministe complet équivalent à A

Définition (Automate des parties)

Etant donné un automate $A = \langle Q, V, \delta_A, I, F_A \rangle$ sans ε-transition, on construit l'automate $B = \langle \mathcal{P}(Q), V, \delta_B, \{I\}, F_B \rangle$, où :

- δ_B est défini par

$$\forall P \subseteq Q, \forall a \in V, \ \delta_B(P, a) \ = \ \{q \in Q \mid \exists p \in P : (p, a, q) \in \delta_A\}$$

- $F_B = \{P \subseteq Q \mid P \cap F_A \neq \emptyset\}$

Propriété (caractérisation des chemins)

Proposition

Pour tout $w \in V^*$ *et pour tout* $P \subseteq Q$*, on a*
$\delta_B^*(P, w) = \{q \in Q \mid \exists p \in P, \exists$ *un chemin dans* A *de* p *à* q *de trace* $w\}$.

Propriété (caractérisation des chemins)

Proposition

Pour tout $w \in V^*$ *et pour tout* $P \subseteq Q$*, on a*
$\delta^*_B(P, w) = \{q \in Q \mid \exists p \in P, \exists$ *un chemin dans* A *de* p *à* q *de trace* $w\}$.

Preuve : par induction sur w

Propriété (caractérisation des chemins)

Proposition

Pour tout $w \in V^*$ *et pour tout* $P \subseteq Q$*, on a*
$\delta_B^*(P, w) = \{q \in Q \mid \exists p \in P, \exists$ *un chemin dans* A *de* p *à* q *de trace* $w\}$.

Preuve : par induction sur w

- $w = \varepsilon$:

Propriété (caractérisation des chemins)

Proposition

Pour tout $w \in V^*$ *et pour tout* $P \subseteq Q$, *on a*
$\delta_B^*(P, w) = \{q \in Q \mid \exists p \in P, \exists$ *un chemin dans* A *de* p *à* q *de trace* $w\}$.

Preuve : par induction sur w

- $w = \varepsilon$:
  - $\delta_B^*(P, \varepsilon) = P$

Propriété (caractérisation des chemins)

Proposition

Pour tout $w \in V^*$ et pour tout $P \subseteq Q$, on a
$\delta_B^*(P, w) = \{q \in Q \mid \exists p \in P, \exists$ *un chemin dans* A *de* p *à* q *de trace* $w\}$.

Preuve : par induction sur w

- $w = \varepsilon$:
  - $\delta_B^*(P, \varepsilon) = P$
  - $\{q \in Q \mid \exists p \in P, \exists$ un chemin dans A de p à q de trace $\varepsilon\} \; = \; P$
    (car A est un automate sans ε-transition)

Propriété (caractérisation des chemins)

Proposition

Pour tout $w \in V^*$ *et pour tout* $P \subseteq Q$*, on a*
$\delta^*_B(P, w) = \{q \in Q \mid \exists p \in P, \exists$ *un chemin dans* A *de* p *à* q *de trace* $w\}$.

Preuve : par induction sur w

Propriété (caractérisation des chemins)

Proposition

Pour tout $w \in V^*$ *et pour tout* $P \subseteq Q$*, on a*
$\delta_B^*(P, w) = \{q \in Q \mid \exists p \in P, \exists$ *un chemin dans* A *de* p *à* q *de trace* $w\}$.

Preuve : par induction sur w

- $w = aw'$:

Propriété (caractérisation des chemins)

Proposition

Pour tout $w \in V^*$ et pour tout $P \subseteq Q$, on a
$\delta^*_B(P, w) = \{q \in Q \mid \exists p \in P, \exists$ *un chemin dans* A *de* p *à* q *de trace* $w\}$.

Preuve : par induction sur w

- $w = aw'$:
  - Posons $P' \stackrel{\text{def}}{=} \delta_B(P, a) = \{q \in Q \mid \exists p \in P, (p, a, q) \in \delta_A\}$

Propriété (caractérisation des chemins)

Proposition

Pour tout $w \in V^*$ *et pour tout* $P \subseteq Q$*, on a*
$\delta_B^*(P, w) = \{q \in Q \mid \exists p \in P, \exists$ *un chemin dans* A *de* p *à* q *de trace* $w\}$.

Preuve : par induction sur w

- $w = aw'$:
  - Posons $P' \stackrel{\text{def}}{=} \delta_B(P, a) = \{q \in Q \mid \exists p \in P, (p, a, q) \in \delta_A\}$
    $\delta_B^*(P, aw') = \delta_B^*(\delta_B(P, a), w') = \delta_B^*(P', w')$

Propriété (caractérisation des chemins)

Proposition

Pour tout $w \in V^*$ et pour tout $P \subseteq Q$, on a
$\delta_B^*(P, w) = \{q \in Q \mid \exists p \in P, \exists$ *un chemin dans A de p à q de trace w*$\}$.

Preuve : par induction sur w

- $w = aw'$:
  - Posons $P' \stackrel{\text{def}}{=} \delta_B(P, a) = \{q \in Q \mid \exists p \in P, (p, a, q) \in \delta_A\}$
    $\delta_B^*(P, aw') = \delta_B^*(\delta_B(P, a), w') = \delta_B^*(P', w')$
    $\stackrel{\text{HI}}{=} \{q' \in Q \mid \exists p' \in P', \exists$ un chemin de p' à q' de trace $w'\}$

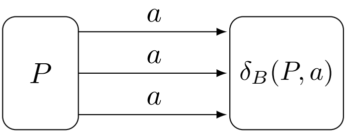
Propriété (caractérisation des chemins)

Proposition

Pour tout $w \in V^*$ *et pour tout* $P \subseteq Q$*, on a*
$\delta_B^*(P, w) = \{q \in Q \mid \exists p \in P, \exists$ *un chemin dans* A *de* p *à* q *de trace* $w\}$.

Preuve : par induction sur w

- $w = aw'$:
  - Posons $P' \overset{\text{def}}{=} \delta_B(P, a) = \{q \in Q \mid \exists p \in P, (p, a, q) \in \delta_A\}$
    $\delta_B^*(P, aw') = \delta_B^*(\delta_B(P, a), w') = \delta_B^*(P', w')$
    $\overset{\text{HI}}{=} \{q' \in Q \mid \exists p' \in P', \exists$ un chemin de p' à q' de trace $w'\}$



Propriété (caractérisation des chemins)

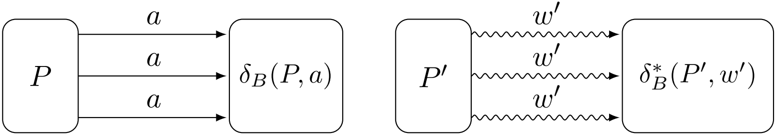
Proposition

Pour tout $w \in V^*$ et pour tout $P \subseteq Q$, on a
$\delta_B^*(P, w) = \{q \in Q \mid \exists p \in P, \exists$ *un chemin dans* A *de* p *à* q *de trace* $w\}$.

Preuve : par induction sur w

- $w = aw'$:
  - Posons $P' \overset{\text{def}}{=} \delta_B(P, a) = \{q \in Q \mid \exists p \in P, (p, a, q) \in \delta_A\}$

$$\begin{aligned} \delta_B^*(P, aw') &= \delta_B^*(\delta_B(P, a), w') = \delta_B^*(P', w') \\ &\overset{\text{HI}}{=} \{q' \in Q \mid \exists p' \in P', \exists \text{ un chemin de } p' \text{ à } q' \text{ de trace } w'\} \end{aligned}$$



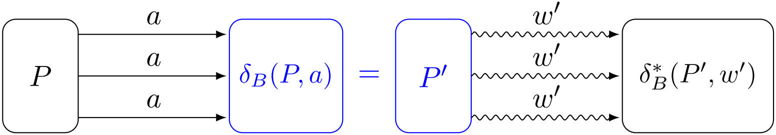
Propriété (caractérisation des chemins)

Proposition

Pour tout $w \in V^*$ *et pour tout* $P \subseteq Q$*, on a*
$\delta_B^*(P, w) = \{q \in Q \mid \exists p \in P, \exists$ *un chemin dans* A *de* p *à* q *de trace* $w\}$.

Preuve : par induction sur w

- $w = aw'$:
  - Posons $P' \overset{\text{def}}{=} \delta_B(P, a) = \{q \in Q \mid \exists p \in P, (p, a, q) \in \delta_A\}$
    $\delta_B^*(P, aw') = \delta_B^*(\delta_B(P, a), w') = \delta_B^*(P', w')$
    $\overset{\text{HI}}{=} \{q' \in Q \mid \exists p' \in P', \exists$ un chemin de p' à q' de trace $w'\}$



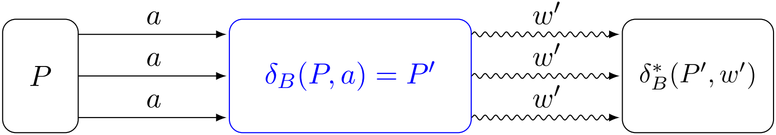
Propriété (caractérisation des chemins)

Proposition

Pour tout $w \in V^*$ et pour tout $P \subseteq Q$, on a
$\delta_B^*(P, w) = \{q \in Q \mid \exists p \in P, \exists$ *un chemin dans* A *de* p *à* q *de trace* $w\}$.

Preuve : par induction sur w

- $w = aw'$:
  - Posons $P' \stackrel{\text{def}}{=} \delta_B(P, a) = \{q \in Q \mid \exists p \in P, (p, a, q) \in \delta_A\}$
    $\delta_B^*(P, aw') = \delta_B^*(\delta_B(P, a), w') = \delta_B^*(P', w')$
    $\stackrel{\text{HI}}{=} \{q' \in Q \mid \exists p' \in P', \exists$ un chemin de p' à q' de trace $w'\}$



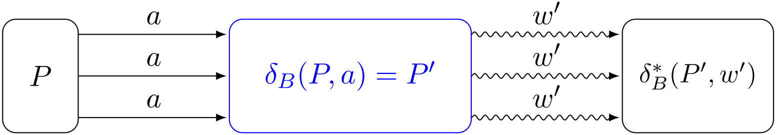
Propriété (caractérisation des chemins)

Proposition

Pour tout $w \in V^*$ *et pour tout* $P \subseteq Q$*, on a*
$\delta_B^*(P, w) = \{q \in Q \mid \exists p \in P, \exists$ *un chemin dans* A *de* p *à* q *de trace* $w\}$.

Preuve : par induction sur w

- $w = aw'$:
  - Posons $P' \overset{\text{def}}{=} \delta_B(P, a) = \{q \in Q \mid \exists p \in P, (p, a, q) \in \delta_A\}$
    $\delta_B^*(P, aw') = \delta_B^*(\delta_B(P, a), w') = \delta_B^*(P', w')$
    $\overset{\text{HI}}{=} \{q' \in Q \mid \exists p' \in P', \exists$ un chemin de p' à q' de trace $w'\}$
    $= \{q \in Q \mid \exists p \in P, \exists$ un chemin de p à q de trace $aw'\}$



Correction de la déterminisation

Théorème

L'automate B *est équivalent à* A*.*

Correction de la déterminisation

Théorème

L'automate B est équivalent à A.

Preuve :

$$w \in \mathcal{L}(A) \quad \Longleftrightarrow \quad \exists p \in I, \exists q \in F_A, \exists \text{ un chemin de } p \text{ à } q \text{ de trace } w$$

Correction de la déterminisation

Théorème

L'automate B est équivalent à A.

Preuve :

$$\begin{aligned} w \in \mathcal{L}(A) &\iff \exists p \in I,\ \exists q \in F_A,\ \exists \text{ un chemin de } p \text{ à } q \text{ de trace } w \\ &\iff \delta_B^*(I, w) \cap F_A \neq \emptyset \end{aligned}$$

Correction de la déterminisation

Théorème

L'automate B est équivalent à A.

Preuve :

$$\begin{aligned} w \in \mathcal{L}(A) & \iff \exists p \in I,\ \exists q \in F_A,\ \exists \text{ un chemin de } p \text{ à } q \text{ de trace } w \\ & \iff \delta_B^*(I, w) \cap F_A \neq \emptyset \\ & \iff \delta_B^*(I, w) \in F_B \end{aligned}$$

Correction de la déterminisation

Théorème

L'automate B est équivalent à A.

Preuve :

$$
\begin{array}{rcl}
w \in \mathcal{L}(A) & \iff & \exists p \in I, \exists q \in F_A, \exists \text{ un chemin de } p \text{ à } q \text{ de trace } w \\
& \iff & \delta_B^*(I, w) \cap F_A \neq \emptyset \\
& \iff & \delta_B^*(I, w) \in F_B \\
& \iff & w \in \mathcal{L}(B)
\end{array}
$$

Correction de la déterminisation

Théorème

L'automate B est équivalent à A.

Preuve :

$$
\begin{array}{rcl}
w \in \mathcal{L}(A) & \iff & \exists p \in I, \exists q \in F_A, \exists \text{ un chemin de } p \text{ à } q \text{ de trace } w \\
 & \iff & \delta_B^*(I, w) \cap F_A \neq \emptyset \\
 & \iff & \delta_B^*(I, w) \in F_B \\
 & \iff & w \in \mathcal{L}(B)
\end{array}
$$

Les automates A et B sont bien équivalents.

Implémentation des automates

Pour les AFD complets

AFD = cas facile

- jamais de choix à faire (déterminisme)
- toujours défini (complétude)

$\rightsquigarrow$ existence + unicité du chemin
$\rightsquigarrow$ important pour définir l'état d'un système (cf cours d'architecture)

Pour les AFD complets

AFD = cas facile

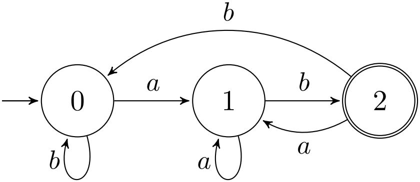
- jamais de choix à faire (déterminisme)
- toujours défini (complétude)

$\rightsquigarrow$ existence + unicité du chemin
$\rightsquigarrow$ important pour définir l'état d'un système (cf cours d'architecture)

3 méthodes :

- par table
- par tests
- par fonctions

Exemple filé : $L = V^*.\{ab\}$



Interface

Comment est représenté le mot ?

- par une chaîne de caractères (tout est connu d'un coup)
- par une source de caractères (arrivée au compte-goutte)

```
# initialisation de la source
init_input(...)

# acces au caractere suivant
next_char()
```

Exemples

```
import sys
in_stream = sys.stderr

def init_input():
    global in_stream = sys.stdin

def next_char():
    global in_stream
    return in_stream.read(1)
```

```
word = ""
index = 0

def init_input(w):
    global word = w
    global index = -1

def next_char():
    global index+=1
    return global word[index]
```

Automates et fin de mot

```
def exec():
    state = i
    init_input()
    ch = next_char()
    while ???:
        state = step(state, ch)
        ch = next_char()
    return state ∈ accepting
```

avec :

- i l'état initial
- step la fonction de transition
- accepting les états acceptants

Automates et fin de mot

```
def exec():
    state = i
    init_input()
    ch = next_char()
    while ???:
        state = step(state, ch)
        ch = next_char()
    return state ∈ accepting
```

avec :

- i l'état initial
- step la fonction de transition
- accepting les états acceptants

Comment reconnaître la fin du mot ?

Automates et fin de mot

```
def exec():
    state = i
    init_input()
    ch = next_char()
    while ???:
        state = step(state, ch)
        ch = next_char()
    return state ∈ accepting
```

avec :

- `i` l'état initial
- `step` la fonction de transition
- `accepting` les états acceptants

Comment reconnaître la fin du mot ?

- Connaître la taille du mot (exemple : chaînes Python, OCaml)
  possible uniquement si toute l'entrée est disponible à la fois
- Ajouter un caractère spécial $\notin V$ de fin de mot (`'\0'`, `EOL`/`EOF`)
  marche dans tous les cas (réseau)

Automates et fin de mot

```
def exec():
    state = i
    init_input()
    ch = next_char()
    while ch != '$':
        state = step(state, ch)
        ch = next_char()
    return state ∈ accepting
```

avec :

- i l'état initial
- step la fonction de transition
- accepting les états acceptants
- $ le caractère de fin

Comment reconnaître la fin du mot ?

- Connaître la taille du mot (exemple : chaînes Python, OCaml)
  possible uniquement si toute l'entrée est disponible à la fois
- Ajouter un caractère spécial $\notin V$ de fin de mot (`'\0'`, `EOL/EOF`)
  marche dans tous les cas (réseau)

Ici, on choisit d'ajouter un caractère spécial $ (ou `END`)

Implémentation par table ou par tests

Par table

Idée : la fonction de transition est une matrice

```
def A = {0: {'a': 1, 'b': 0},
         1: {'a': 1, 'b': 2},
         2: {'a': 1, 'b': 0}]

def step(state, ch):
    return A.transitions[state][ch]
```

Implémentation par table ou par tests

Par table

Idée : la fonction de transition est une matrice

```
def A = {0: {'a': 1, 'b': 0},
         1: {'a': 1, 'b': 2},
         2: {'a': 1, 'b': 0}]

def step(state, ch):
    return A.transitions[state][ch]
```

Par tests

Idée : la fonction de transition est un bout de code

```
def step(state, ch):
    if state == 0:
        if ch == 'a':
            return 1
        elif ch == 'b':
            return 0
    elif state == 1:
        ⋮
```

Implémentation par table ou par tests

Par table

Idée : la fonction de transition est une matrice

```
def A = {0: {'a': 1, 'b': 0},
         1: {'a': 1, 'b': 2},
         2: {'a': 1, 'b': 0}]

def step(state, ch):
    return A.transitions[state][ch]
```

$\leadsto$ l'automate est une donnée

Par tests

Idée : la fonction de transition est un bout de code

```
def step(state, ch):
    if state == 0:
        if ch == 'a':
            return 1
        elif ch == 'b':
            return 0
    elif state == 1:
        ⋮
```

$\leadsto$ l'automate est un programme

Implémentation par table ou par tests

Par table

Idée : la fonction de transition est une matrice

```
def A = {0: {'a': 1, 'b': 0},
         1: {'a': 1, 'b': 2},
         2: {'a': 1, 'b': 0}]

def step(state, ch):
    return A.transitions[state][ch]
```

$\rightsquigarrow$ l'automate est une donnée

Coût = lecture mémoire

Par tests

Idée : la fonction de transition est un bout de code

```
def step(state, ch):
    if state == 0:
        if ch == 'a':
            return 1
        elif ch == 'b':
            return 0
    elif state == 1:
        ⋮
```

$\rightsquigarrow$ l'automate est un programme

Coût = tests + sauts

Implémentation par fonctions

Idée : chaque état est une fonction

$\rightsquigarrow$ pas de boucle **while** ni de fonction step

```
def state0():
    ch = next_char()
    if ch == 'a':
        return state1()
    elif ch == 'b':
        return state0()
    elif ch == '$':
        return False
```

```
def state1():
    ch = next_char()
    if ch == 'a':
        return state1()
    elif ch == 'b':
        return state2()
    elif ch == '$':
        return False
```

```
def state2():
    ch = next_char()
    if ch == 'a':
        return state1()
    elif ch == 'b':
        return state0()
    elif ch == '$':
        return True
```

Implémentation par fonctions

Idée : chaque état est une fonction

$\rightsquigarrow$ pas de boucle **while** ni de fonction step

```
def state0():
    ch = next_char()
    if ch == 'a':
        return state1()
    elif ch == 'b':
        return state0()
    elif ch == '$':
        return False
```

```
def state1():
    ch = next_char()
    if ch == 'a':
        return state1()
    elif ch == 'b':
        return state2()
    elif ch == '$':
        return False
```

```
def state2():
    ch = next_char()
    if ch == 'a':
        return state1()
    elif ch == 'b':
        return state0()
    elif ch == '$':
        return True
```

- plus modulaire que la boucle while
- permet de faire du calcul
- Coût = appel de fonction

automate	=	ensemble de fonctions
lecture	=	appeler l'état initial
état courant	=	la fonction qui s'exécute

```
def exec_v2():
    init_input()
    return state0()
```

Implémentation par fonctions

Idée : chaque état est une fonction

$\rightsquigarrow$ pas de boucle **while** ni de fonction step

```
def state0():
    ch = next_char()
    if ch == 'a':
        return state1()
    elif ch == 'b':
        return state0()
    elif ch == '$':
        return False
```

```
def state1():
    ch = next_char()
    if ch == 'a':
        return state1()
    elif ch == 'b':
        return state2()
    elif ch == '$':
        return False
```

```
def state2():
    ch = next_char()
    if ch == 'a':
        return state1()
    elif ch == 'b':
        return state0()
    elif ch == '$':
        return True
```

- plus modulaire que la boucle while
- permet de faire du calcul
- Coût = appel de fonction réduit par des sauts (appel terminal)

automate	=	ensemble de fonctions
lecture	=	appeler l'état initial
état courant	=	la fonction qui s'exécute

```
def exec_v2():
    init_input()
    return state0()
```

Quelle méthode pour les AFND ?

- Par fonction
  les fonctions renvoient un seul état...
  Exponentiel avec plusieurs états ! (nb de chemins)

Quelle méthode pour les AFND ?

- Par fonction
  les fonctions renvoient un seul état...
  Exponentiel avec plusieurs états ! (nb de chemins)

- Par matrice ou tests
  variable `state` = ensemble d'états
  + itération sur `state` pour calculer l'état suivant
  $\leadsto$ deux variables : `state`, `new_state`
  $\leadsto$ boucles sur `state` paralléliser les tests / lectures mémoire

Quelle méthode pour les AFND ?

- Par fonction
  les fonctions renvoient un seul état...
  Exponentiel avec plusieurs états ! (nb de chemins)

- Par matrice ou tests
  variable state = ensemble d'états
  + itération sur state pour calculer l'état suivant
  $\leadsto$ deux variables : state, new_state
  $\leadsto$ boucles sur state paralléliser les tests / lectures mémoire

```
def step(A, state, ch):
    new_state = set.empty()
    for q in state:
        new_state.add(
            A.transitions[
                state, ch])
    return new_state
```

```
def exec(A):
    state = A.init
    init_input()
    ch = next_char()
    while (ch != '$'):
        state = step(A, state, ch)
        ch = next_char()
    return A.accepting.inter(state)
```

Comparaison AFD/AFND

AFD

- exécution très rapide $O(|w|)$ (jamais de choix à faire)
- plus gros que AFND (exponentiellement !)

Cas d'utilisation

- Vitesse exigée
- Beaucoup d'utilisation
- Construction de l'automate à l'avance

$\rightsquigarrow$ ex : compilateur

AFND

- exécution plus lente $O(|w| \cdot |Q|)$ (ensembles d'états)
- plus compacts que AFD

Cas d'utilisation

- Contraintes d'espace
- Utilisation unique (ou faible)
- Construction de l'automate à l'utilisation
- Facile dans les circuits

$\rightsquigarrow$ ex : expressions régulières

Comparaison AFD/AFND

AFD

- exécution très rapide $O(|w|)$ (jamais de choix à faire)
- plus gros que AFND (exponentiellement !)

Cas d'utilisation

- Vitesse exigée
- Beaucoup d'utilisation
- Construction de l'automate à l'avance

$\leadsto$ ex : compilateur

AFND

- exécution plus lente $O(|w| \cdot |Q|)$ (ensembles d'états)
- plus compacts que AFD

Cas d'utilisation

- Contraintes d'espace
- Utilisation unique (ou faible)
- Construction de l'automate à l'utilisation
- Facile dans les circuits

$\leadsto$ ex : expressions régulières

Choix AFD/AFND = compromis espace/temps

Exemple d'utilisation : analyse lexicale

Première partie d'un compilateur : reconnaître les programmes corrects

Étapes :

1. Décrire les programmes corrects expression régulière / grammaire
2. Construire un AFND
3. Le déterminiser
4. En faire une implémentation par `goto`

Exemple d'utilisation : analyse lexicale

Première partie d'un compilateur : reconnaître les programmes corrects

Étapes :

1. Décrire les programmes corrects — expression régulière / grammaire
2. Construire un AFND
3. Le déterminiser
4. En faire une implémentation par `goto`

En plus

- faire du calcul vs. OUI/NON — voir TP/projet
- gestion des erreurs (cas `else` des `if`)

Exemple d'utilisation : analyse lexicale

Première partie d'un compilateur : reconnaître les programmes corrects

Étapes :

1. Décrire les programmes corrects — expression régulière / grammaire
2. Construire un AFND
3. Le déterminiser
4. En faire une implémentation par `goto`

En plus

- faire du calcul vs. OUI/NON — voir TP/projet
- gestion des erreurs (cas `else` des `if`)

En TP : reconnaissance des constantes flottantes en Python

Application des automates

Application 1 : algo KMP

Contexte : recherche d'un motif m dans un texte t

Contraintes :

- construction de l'automate linéaire (en $|m|$)
- parcours du texte en ligne = caractère par caractère
  pas de retour en arrière possible
  Complexité linéaire en $|t|$

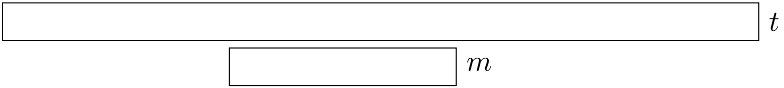
Application 1 : algo KMP

Contexte : recherche d'un motif m dans un texte t

Contraintes :

- construction de l'automate linéaire (en $|m|$)
- parcours du texte en ligne = caractère par caractère
  pas de retour en arrière possible
  Complexité linéaire en $|t|$

Idée : décaler le motif de « juste ce qu'il faut » en cas d'erreur
sans rater d'occurence de m



Application 1 : algo KMP

Contexte : recherche d'un motif m dans un texte t

Contraintes :

- construction de l'automate linéaire (en $|m|$)
- parcours du texte en ligne = caractère par caractère
  pas de retour en arrière possible
  Complexité linéaire en $|t|$

Idée : décaler le motif de « juste ce qu'il faut » en cas d'erreur
sans rater d'occurence de m

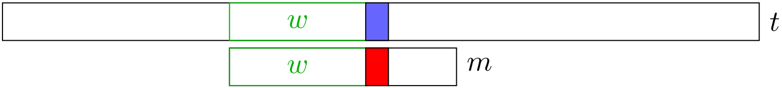
Application 1 : algo KMP

Contexte : recherche d'un motif m dans un texte t

Contraintes :

- construction de l'automate linéaire (en $|m|$)
- parcours du texte en ligne = caractère par caractère
  pas de retour en arrière possible
  Complexité linéaire en $|t|$

Idée : décaler le motif de « juste ce qu'il faut » en cas d'erreur
sans rater d'occurence de m



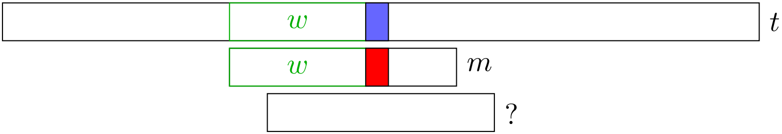
Application 1 : algo KMP

Contexte : recherche d'un motif m dans un texte t

Contraintes :

- construction de l'automate linéaire (en $|m|$)
- parcours du texte en ligne = caractère par caractère
  pas de retour en arrière possible
  Complexité linéaire en $|t|$

Idée : décaler le motif de « juste ce qu'il faut » en cas d'erreur
sans rater d'occurence de m



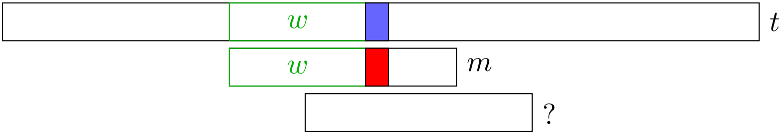
Application 1 : algo KMP

Contexte : recherche d'un motif m dans un texte t

Contraintes :

- construction de l'automate linéaire (en $|m|$)
- parcours du texte en ligne = caractère par caractère
  pas de retour en arrière possible
  Complexité linéaire en $|t|$

Idée : décaler le motif de « juste ce qu'il faut » en cas d'erreur
sans rater d'occurence de m



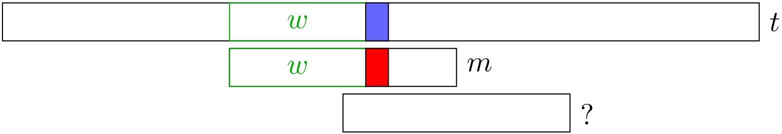
Application 1 : algo KMP

Contexte : recherche d'un motif m dans un texte t

Contraintes :

- construction de l'automate linéaire (en $|m|$)
- parcours du texte en ligne = caractère par caractère
  pas de retour en arrière possible
  Complexité linéaire en $|t|$

Idée : décaler le motif de « juste ce qu'il faut » en cas d'erreur
sans rater d'occurence de m



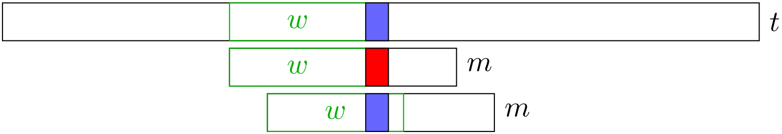
Application 1 : algo KMP

Contexte : recherche d'un motif m dans un texte t

Contraintes :

- construction de l'automate linéaire (en $|m|$)
- parcours du texte en ligne = caractère par caractère
  pas de retour en arrière possible
  Complexité linéaire en $|t|$

Idée : décaler le motif de « juste ce qu'il faut » en cas d'erreur
sans rater d'occurence de m



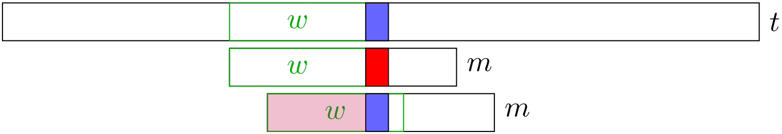
Application 1 : algo KMP

Contexte : recherche d'un motif m dans un texte t

Contraintes :

- construction de l'automate linéaire (en $|m|$)
- parcours du texte en ligne = caractère par caractère
  pas de retour en arrière possible
  Complexité linéaire en $|t|$

Idée : décaler le motif de « juste ce qu'il faut » en cas d'erreur
sans rater d'occurence de m



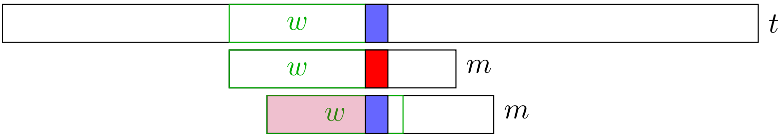
Application 1 : algo KMP

Contexte : recherche d'un motif m dans un texte t

Contraintes :

- construction de l'automate linéaire (en $|m|$)
- parcours du texte en ligne = caractère par caractère
  pas de retour en arrière possible
  Complexité linéaire en $|t|$

Idée : décaler le motif de « juste ce qu'il faut » en cas d'erreur
sans rater d'occurence de m



Que peut-on dire de par rapport à w ?

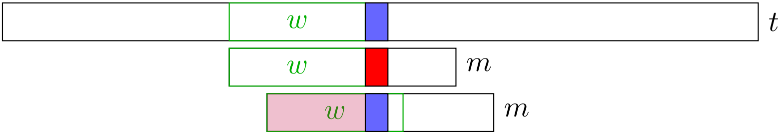
Application 1 : algo KMP

Contexte : recherche d'un motif m dans un texte t

Contraintes :

- construction de l'automate linéaire (en $|m|$)
- parcours du texte en ligne = caractère par caractère
  pas de retour en arrière possible
  Complexité linéaire en $|t|$

Idée : décaler le motif de « juste ce qu'il faut » en cas d'erreur
sans rater d'occurence de m



Que peut-on dire de par rapport à w ?
C'est un préfixe (strict) et un suffixe de w !

Calcul des bords d'un mot

Définition (Bord)

Le bord d'un mot $w \neq \varepsilon$ est son plus grand préfixe strict qui en est également un suffixe. On le note $\varphi(w)$.

Calcul des bords d'un mot

Définition (Bord)

Le bord d'un mot $w \neq \varepsilon$ est son plus grand préfixe strict qui en est également un suffixe. On le note $\varphi(w)$.

Pourquoi strict ?

Calcul des bords d'un mot

Définition (Bord)

Le bord d'un mot $w \neq \varepsilon$ est son plus grand préfixe strict qui en est également un suffixe. On le note $\varphi(w)$.

Pourquoi strict ?

Calcul des bords d'un mot

Définition (Bord)

Le bord d'un mot $w \neq \varepsilon$ est son plus grand préfixe strict qui en est également un suffixe. On le note $\varphi(w)$.

Pourquoi strict ?

Calcul de $\varphi(w)$

- pour $a \in V$, $\varphi(a) = ?$

Calcul des bords d'un mot

Définition (Bord)

Le bord d'un mot $w \neq \varepsilon$ est son plus grand préfixe strict qui en est également un suffixe. On le note $\varphi(w)$.

Pourquoi strict ?

Calcul de $\varphi(w)$

- pour $a \in V$, $\varphi(a) = \varepsilon$

Calcul des bords d'un mot

Définition (Bord)

Le bord d'un mot $w \neq \varepsilon$ est son plus grand préfixe strict qui en est également un suffixe. On le note $\varphi(w)$.

Pourquoi strict ?

Calcul de $\varphi(w)$

- pour $a \in V$, $\varphi(a) = \varepsilon$
- pour $w \in V^+, a \in V$, $\varphi(wa) = ?$

Calcul des bords d'un mot

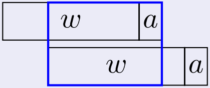
Définition (Bord)

Le bord d'un mot $w \neq \varepsilon$ est son plus grand préfixe strict qui en est également un suffixe. On le note $\varphi(w)$.

Pourquoi strict ?

Calcul de $\varphi(w)$

- pour $a \in V$, $\varphi(a) = \varepsilon$
- pour $w \in V^+, a \in V$, $\varphi(wa) =$



Calcul des bords d'un mot

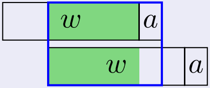
Définition (Bord)

Le bord d'un mot $w \neq \varepsilon$ est son plus grand préfixe strict qui en est également un suffixe. On le note $\varphi(w)$.

Pourquoi strict ?

Calcul de $\varphi(w)$

- pour $a \in V$, $\varphi(a) = \varepsilon$
- pour $w \in V^+, a \in V$, $\varphi(wa) =$



Calcul des bords d'un mot

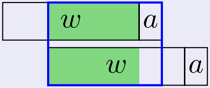
Définition (Bord)

Le bord d'un mot $w \neq \varepsilon$ est son plus grand préfixe strict qui en est également un suffixe. On le note $\varphi(w)$.

Pourquoi strict ?

Calcul de $\varphi(w)$

- pour $a \in V$, $\varphi(a) = \varepsilon$
- pour $w \in V^+, a \in V$, $\varphi(wa) = \begin{cases} \varphi(w)a & \text{si } \varphi(w)a \text{ préfixe de } w \\ & \end{cases}$



Calcul des bords d'un mot

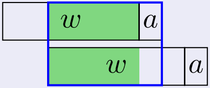
Définition (Bord)

Le bord d'un mot $w \neq \varepsilon$ est son plus grand préfixe strict qui en est également un suffixe. On le note $\varphi(w)$.

Pourquoi strict ?

Calcul de $\varphi(w)$

- pour $a \in V$, $\varphi(a) = \varepsilon$
- pour $w \in V^+, a \in V$, $\varphi(wa) = \begin{cases} \varphi(w)a & \text{si } \varphi(w)a \text{ préfixe de } w \\ \varphi(\varphi(w)a) & \text{sinon} \end{cases}$



Calcul des bords d'un mot

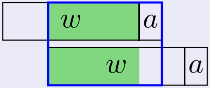
Définition (Bord)

Le bord d'un mot $w \neq \varepsilon$ est son plus grand préfixe strict qui en est également un suffixe. On le note $\varphi(w)$.

Pourquoi strict ?

Calcul de $\varphi(w)$

- pour $a \in V$, $\varphi(a) = \varepsilon$
- pour $w \in V^+, a \in V$, $\varphi(wa) = \begin{cases} \varphi(w)a & \text{si } \varphi(w)a \text{ préfixe de } w \\ \varphi(\varphi(w)a) & \text{sinon} \end{cases}$



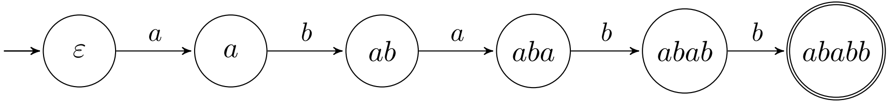
Idée : si problème après w, réessayer avec $\varphi(w)$!

Exemple : recherche de $ababb$

Recherche de $m = ababb$ dans $t = abaababb$.

Exemple : recherche de $ababb$

Recherche de $m = ababb$ dans $t = abaababb$.



Exemple : recherche de $ababb$

Recherche de $m = ababb$ dans $t = abaababb$.

$$\rightarrow (\varepsilon) \xrightarrow{a} (a) \xrightarrow{b} (ab) \xrightarrow{a} (aba) \xrightarrow{b} (abab) \xrightarrow{b} ((ababb))$$

Calcul de $\varphi(w)$ pour w préfixe de m :

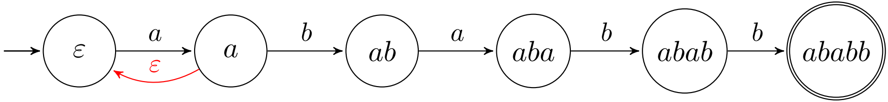
$\varphi(a) =$ $\qquad$ $\varphi(ab) =$

$\varphi(aba) =$ $\qquad$ $\varphi(abab) =$

$\varphi(ababb) =$

Exemple : recherche de $ababb$

Recherche de $m = ababb$ dans $t = abaababb$.



Calcul de $\varphi(w)$ pour w préfixe de m :

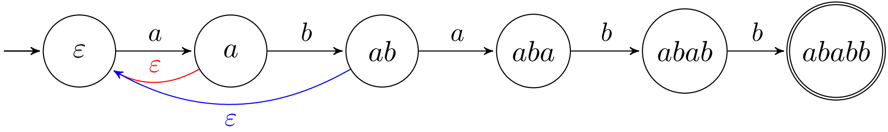
$\varphi(a) = \varepsilon$ $\qquad$ $\varphi(ab) =$

$\varphi(aba) =$ $\qquad$ $\varphi(abab) =$

$\varphi(ababb) =$

Exemple : recherche de $ababb$

Recherche de $m = ababb$ dans $t = abaababb$.



Calcul de $\varphi(w)$ pour w préfixe de m :

$\varphi(a) = \varepsilon$

$\varphi(ab) = \varphi(\varphi(a)b) = \varphi(\varepsilon b) = \varepsilon$

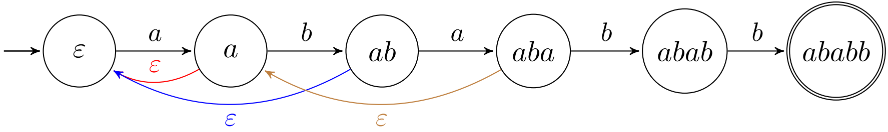
$\varphi(aba) =$

$\varphi(abab) =$

$\varphi(ababb) =$

Exemple : recherche de $ababb$

Recherche de $m = ababb$ dans $t = abaababb$.



Calcul de $\varphi(w)$ pour w préfixe de m :

$\varphi(a) = \varepsilon$

$\varphi(ab) = \varphi(\varphi(a)b) = \varphi(\varepsilon b) = \varepsilon$

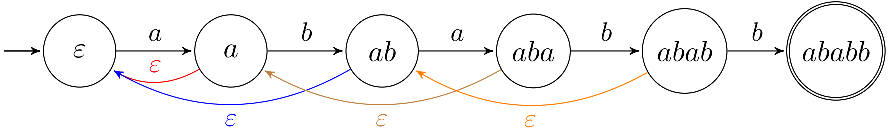
$\varphi(aba) = \varphi(ab)a = a$

$\varphi(abab) =$

$\varphi(ababb) =$

Exemple : recherche de $ababb$

Recherche de $m = ababb$ dans $t = abaababb$.



Calcul de $\varphi(w)$ pour w préfixe de m :

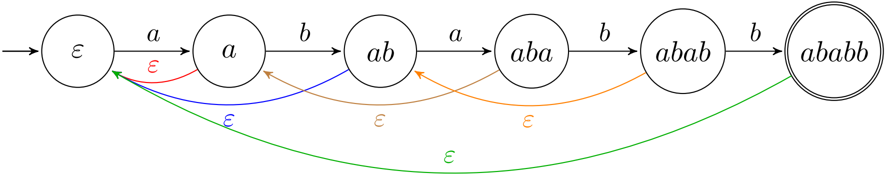
$\varphi(a) = \varepsilon$ $\qquad$ $\varphi(ab) = \varphi(\varphi(a)b) = \varphi(\varepsilon b) = \varepsilon$

$\varphi(aba) = \varphi(ab)a = a$ $\qquad$ $\varphi(abab) = \varphi(aba)b = ab$

$\varphi(ababb) =$

Exemple : recherche de $ababb$

Recherche de $m = ababb$ dans $t = abaababb$.



Calcul de $\varphi(w)$ pour w préfixe de m :

$\varphi(a) = \varepsilon$

$\varphi(ab) = \varphi(\varphi(a)b) = \varphi(\varepsilon b) = \varepsilon$

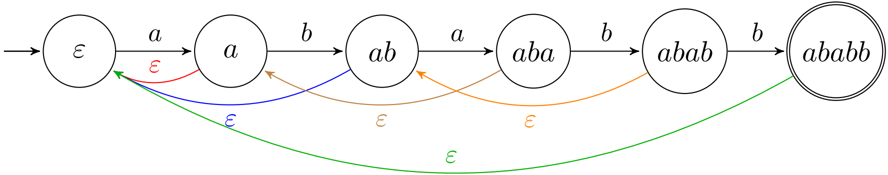
$\varphi(aba) = \varphi(ab)a = a$

$\varphi(abab) = \varphi(aba)b = ab$

$\varphi(ababb) = \varphi(\varphi(abab)b) = \varphi(abb) = \varphi(\varphi(ab)b) = \varphi(b) = \varepsilon$

Exemple : recherche de $ababb$

Recherche de $m = ababb$ dans $t = abaababb$.



Calcul de $\varphi(w)$ pour w préfixe de m :

$\varphi(a) = \varepsilon$ $\quad$ $\varphi(ab) = \varphi(\varphi(a)b) = \varphi(\varepsilon b) = \varepsilon$

$\varphi(aba) = \varphi(ab)a = a$ $\quad$ $\varphi(abab) = \varphi(aba)b = ab$

$\varphi(ababb) = \varphi(\varphi(abab)b) = \varphi(abb) = \varphi(\varphi(ab)b) = \varphi(b) = \varepsilon$

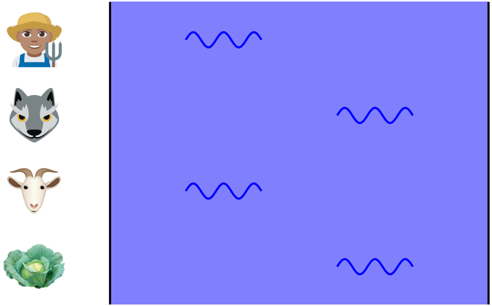
Complexité de la construction : $O|m|$ en temps et en espace

Taille de l'AFD complet : $O(|m| \times |V|)$ $\qquad$ $|V|$ transitions par état

Complexité de la lecture : $O(|t|)$ $\qquad$ analyse amortie

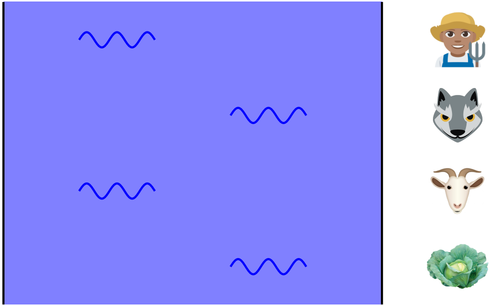
Application 2 : statégie dans un jeu

Jeu du fermier (cf. exercice 15 du recueil de TD)



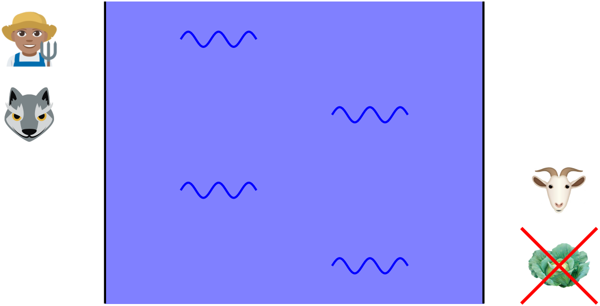
Application 2 : statégie dans un jeu

Jeu du fermier (cf. exercice 15 du recueil de TD)



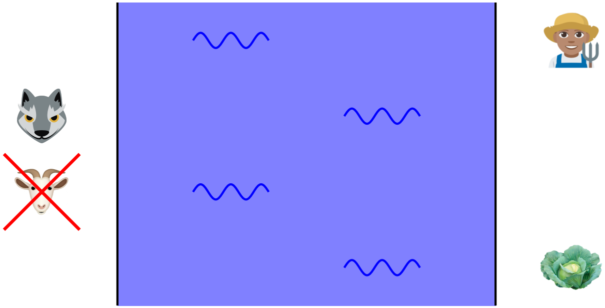
Application 2 : statégie dans un jeu

Jeu du fermier (cf. exercice 15 du recueil de TD)



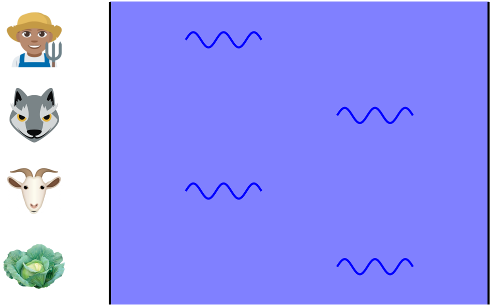
Application 2 : statégie dans un jeu

Jeu du fermier (cf. exercice 15 du recueil de TD)



Application 2 : statégie dans un jeu

Jeu du fermier (cf. exercice 15 du recueil de TD)



Exercice

1. Représenter le problème par un automate, en précisant le vocabulaire choisi.
2. Comment déterminer une stratégie à partir de l'automate ? Quelles sont les stratégies optimales ?

Solution

- États : positions (quelle rive) de 👨‍🌾, 🐺, 🐐, 🥬
  $\rightarrow$ 16 états

Solution

- États : positions (quelle rive) de 🧑‍🌾, 🐺, 🐐, 🥬
  $\rightarrow$ 16 états
- Vocabulaire = mouvements possibles :
  - F : le fermier traverse seul
  - L : le fermier traverse avec le loup
  - C : le fermier traverse avec la chèvre
  - P : le fermier traverse avec le chou

Solution

- États : positions (quelle rive) de 👨🏽‍🌾, 🐺, 🐐, 🥬
  $\rightarrow$ 16 états
- Vocabulaire = mouvements possibles :
  - F : le fermier traverse seul
  - L : le fermier traverse avec le loup
  - C : le fermier traverse avec la chèvre
  - P : le fermier traverse avec le chou
- États acceptants = tous sont sur l'autre rive

Solution

- États : positions (quelle rive) de 🧑‍🌾, 🐺, 🐐, 🥬
  $\rightarrow$ 16 états
- Vocabulaire = mouvements possibles :
  - F : le fermier traverse seul
  - L : le fermier traverse avec le loup
  - C : le fermier traverse avec la chèvre
  - P : le fermier traverse avec le chou
- États acceptants = tous sont sur l'autre rive
- Sans passer par les états perdants

Solution

- États : positions (quelle rive) de 👨🏽‍🌾, 🐺, 🐐, 🥬
  $\rightarrow$ 16 états
- Vocabulaire = mouvements possibles :
  - F : le fermier traverse seul
  - L : le fermier traverse avec le loup
  - C : le fermier traverse avec la chèvre
  - P : le fermier traverse avec le chou
- États acceptants = tous sont sur l'autre rive
- Sans passer par les états perdants
  $\rightsquigarrow$ États perdants = états puits

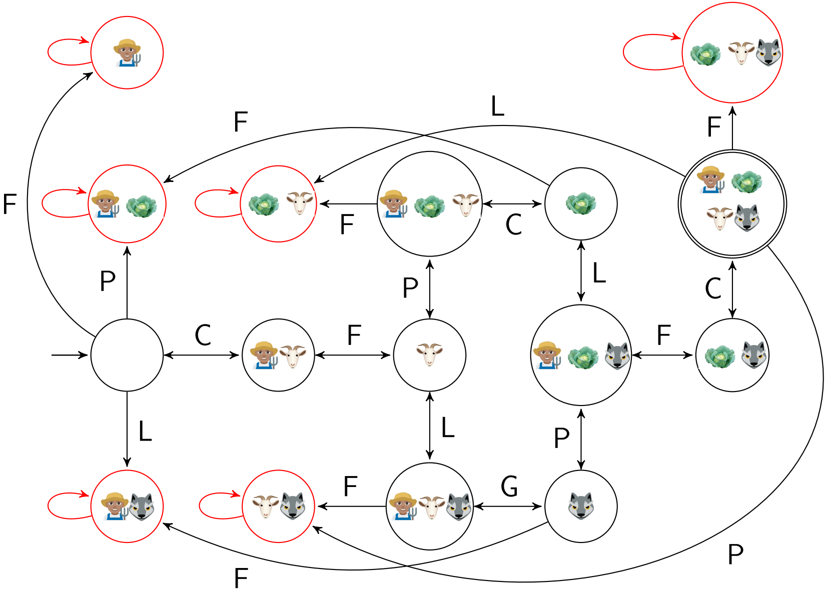
Solution

- États : positions (quelle rive) de 🧑‍🌾, 🐺, 🐐, 🥬
  $\rightarrow$ 16 états
- Vocabulaire = mouvements possibles :
  - F : le fermier traverse seul
  - L : le fermier traverse avec le loup
  - C : le fermier traverse avec la chèvre
  - P : le fermier traverse avec le chou
- États acceptants = tous sont sur l'autre rive
- Sans passer par les états perdants
  $\rightsquigarrow$ États perdants = états puits
- Stratégie gagnante = mot accepté par cet automate

Solution

- États : positions (quelle rive) de 👨🏽‍🌾, 🐺, 🐐, 🥬
  $\rightarrow$ 16 états
- Vocabulaire = mouvements possibles :
  - F : le fermier traverse seul
  - L : le fermier traverse avec le loup
  - C : le fermier traverse avec la chèvre
  - P : le fermier traverse avec le chou
- États acceptants = tous sont sur l'autre rive
- Sans passer par les états perdants
  $\rightsquigarrow$ États perdants = états puits
- Stratégie gagnante = mot accepté par cet automate
  Optimum = 7 coups

Automate



Automate