

Filières MP/MPI - ENS de Paris-Saclay, Lyon, Rennes et Paris - Session 2025
 Page de garde du rapport de TIPE

Attention: Le rapport doit être réalisé par l'étudiant(e).

Si le rapport résulte d'une collaboration, elle doit être clairement annoncée.

NOM : HENNET	Prénoms : Audrey Lise
Classe : MPI	
Lycée : Lycée du Parc	Numéro de candidat : 31 286
Ville : Lyon	

Concours auxquels vous êtes admissible (les indiquer par une croix) :

Banque MP

ENS Paris-Saclay MP - Option MP
 MP - Option MI

ENS de Lyon Math - Option M-MP
 Math - Option M-MI
 Informatique

ENS Rennes MP - Option MP
 MP - Option MI

ENS Paris MP - Option P
 MP - Option I

Banque MPI

ENS Paris-Saclay Option Math
 Option Info

X

ENS de Lyon Math MPI
 Info MPI

X

ENS Rennes Info MPI

--

ENS Paris Option Math
 Option Info

Matière dominante du TIPE (la sélectionner d'une croix inscrite dans la case correspondante) :

Informatique	X	Mathématiques		Physique	
--------------	---	---------------	--	----------	--

Titre du TIPE :

Apprendre un automate fini déterministe par l'algorithme L*

Nombre de pages (à indiquer dans les cases ci-dessous) :

Texte	5	Illustration	2	Bibliographie	1
-------	---	--------------	---	---------------	---

Attention, les illustrations doivent figurer dans le corps du texte et non en fin du document !

Résumé ou descriptif succinct du TIPE (6 lignes, maximum) :

L'objectif est d'étudier l'algorithme L* développé par Dana Angluin (1980) pour apprendre un automate fini déterministe. Une étude du fonctionnement de la correction, de la terminaison et de la complexité est menée ainsi que de la minimalité de l'automate appris.
 Je propose une implémentation en OCaml.

À Lyon
 Le 05/06/2025
 Signature du (de la) candidat(e)

La signature du professeur responsable et le tampon de l'établissement ne sont pas indispensables pour les candidats libres (hors CPGE).

Signature du professeur responsable de
 la classe préparatoire dans la discipline

Cachet de l'établissement

LYCEE DU PARC

1, Boulevard Anatole France

69006 LYON

Tél. 04 37 51 15 51 - INE : 0690026d

Apprendre un automate fini déterministe par l'algorithme L^*

Audrey Hennet

Juin 2025

1 Introduction

J'ai mené ce travail de recherche dans l'optique d'étudier et d'implémenter un algorithme permettant d'apprendre un automate fini déterministe. L'algorithme étudié est l'algorithme L^* développé par Dana Angluin en 1987 [1]. J'en expliquerai le principe puis proposerai une implémentation avec des exemples d'applications. Je montrerai par la même occasion que l'automate appris grâce à cet algorithme est minimal.

Un enseignant connaît un langage régulier \mathcal{L}_m sur un alphabet Σ sous la forme d'un automate fini déterministe A_m . Un élève souhaite apprendre le langage en interrogeant l'enseignant (ou oracle) afin de construire un automate minimal reconnaissant \mathcal{L}_m .

2 Principe

2.1 Enseignant / Élève

Un automate complet déterministe A_m reconnaissant \mathcal{L}_m est connu par l'enseignant `teacher`. L'exemple utilisé sera le langage $\mathcal{L}_m = \{(b|a)b^*\}$ décrit par l'automate Figure 1.

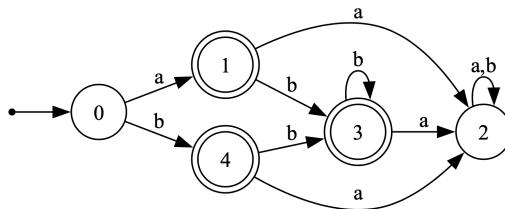


FIGURE 1 – Automate exemple A_m reconnaissant $\mathcal{L}_m = \{(b|a)b^*\}$

Pour apprendre le langage \mathcal{L}_m , l'élève `learner` pose deux types de questions à l'enseignant.

Lorsqu'il appelle `member` sur un mot `word` l'enseignant lui indique si le mot appartient ou pas à \mathcal{L}_m . Nous implémentons cela en faisant un simple test d'appartenance dans l'automate A_m .

Lorsque l'élève appelle la fonction `counterexample` sur un automate fini déterministe complet `automaton`, l'enseignant calcule la différence symétrique des deux langages et lui renvoie le mot le plus court accepté par ce nouvel automate, si un tel mot existe (sinon il renvoie `None`).

On nommera un *enseignant minimalement compétent*, si l'enseignant peut répondre à deux types de questions :

- Est-ce que le mot `word` appartient au langage ?
- Est-ce que l'automate fini déterministe `automaton` reconnaît le langage ? Si ce n'est pas le cas, il renvoie un contre-exemple.

La réponse de l'enseignant à la deuxième requête suppose que l'enseignant connaisse parfaitement l'automate qu'il veut enseigner. Si ce n'est pas le cas, nous verrons en Section 4, une manière de contourner ce problème.

2.2 Table d'observation

Pour apprendre le langage \mathcal{L}_m l'élève **learner** met en place une table d'observation.

Définition 2.1 Une table d'observation $T = (S, E, f)$ associée à un langage \mathcal{L} sur l'alphabet Σ est composée d'un ensemble S de mots clôturés par préfixe, d'un ensemble E de mots clôturés par suffixe et d'une fonction f :

$$f : \Sigma^* \longrightarrow \{\text{true}, \text{false}\}$$

f indique si un mot (quelconque) appartient au langage \mathcal{L} . \diamond

Pour représenter les ensembles S et E on utilise la structure de trie (arbre préfixe).

On utilise des tables de hachage pour associer à chaque couple $(s, e) \in (S \cup S\Sigma) \times E$ la valeur de $f(s \cdot e)$, obtenue grâce à un test d'appartenance ($S\Sigma$ signifie l'ensemble des mots de S suivis d'une lettre de l'alphabet Σ). On utilise aussi une table de hachage pour associer à chaque mot $s \in S \cup S\Sigma$ la ligne à laquelle il correspond.

Définition 2.2 Une fonction f et un langage \mathcal{L} sont compatibles si pour tout $(s, e) \in (S \cup S\Sigma) \times E$, on a :

$$f(s \cdot e) = \text{true} \iff se \in \mathcal{L}$$

Une fonction f et un automate A sont compatibles si f est compatible avec $\mathcal{L}(A)$. \diamond

Définition 2.3 Une table d'observation (S, E, f) est dite close si pour tout $s \in S\Sigma$ il existe $s' \in S$ tel que $\text{ligne}(s) = \text{ligne}(s')$. \diamond

Si une table d'observation T n'est pas close alors il existe un mot $s \in S\Sigma$ tel que $\text{ligne}(s)$ soit différente de toutes les lignes $\text{ligne}(s')$; $s' \in S$ alors on ajoute à S le mot s .

Exemple : D'après la Table 1, $S = \{\varepsilon\}$. Elle n'est donc pas close car $\text{ligne}(b) = 1 \neq \text{ligne}(\varepsilon) = 0$. Le mot "b" est donc ajouté à l'ensemble S de la table d'observation (voir Table 2). La clôture par préfixe est conservée. Dans ces tables, 1 correspond à *true* et 0 à *false*.

$(S \cup S\Sigma)/E$	ε	n° ligne
ε	0	0
b	1	1
a	1	1

TABLE 1 – Table d'observation non close

$(S \cup S\Sigma)/E$	ε	n° ligne
ε	0	0
b	1	1
b	1	1
bb	1	1
a	1	1
ba	0	0

TABLE 2 – Table d'observation close

Définition 2.4 Une table d'observation $T = (S, E, f)$ est dite cohérente si pour tout $s, s' \in S$ tel que $\text{ligne}(s) = \text{ligne}(s')$ on a, $\forall a \in \Sigma$, $\text{ligne}(s \cdot a) = \text{ligne}(s' \cdot a)$. \diamond

Si la table d'observation T n'est pas cohérente alors il existe $s, s' \in S$ tels que $\text{ligne}(s) = \text{ligne}(s')$ et $a \in \Sigma$, $e \in E$ tels que $f(sae) \neq f(s'ae)$. Le mot ae est alors ajouté à E .

Exemple : La table d'observation Table 3 n'est pas cohérente car $\text{ligne}(\varepsilon) = \text{ligne}(aa)$ mais $\text{ligne}(a) \neq \text{ligne}(aaa)$. On ajoute donc le mot "a" à l'ensemble E , et on obtient la Table 4.

2.3 Automate associé

Définition 2.5 Si une table d'observation $T = (S, E, f)$ est close et cohérente, l'automate associé à cette table d'observation est donné par $M(S, E, f) = (\Sigma, Q, q_0, F, \delta)$ avec :

$$Q = \{\text{ligne}(s); s \in S\}$$

$$q_0 = \text{ligne}(\varepsilon)$$

$(S \cup S\Sigma)/E$	ε	n° ligne
ε	0	0
b	1	1
a	1	1
aa	0	0
aaa	0	0
<hr/>		
b	1	1
bb	1	1
ab	1	1
aab	0	0
aaab	0	0
a	1	1
ba	0	0
aa	0	0
aaa	0	0
aaaa	0	0

TABLE 3 – Table d'observation non cohérente

$(S \cup S\Sigma)/E$	ε	a	n° ligne
ε	0	1	0
b	1	0	1
a	1	0	1
aa	0	0	2
aaa	0	0	2
<hr/>			
b	1	0	1
bb	1	0	1
ab	1	0	1
aab	0	0	2
aaab	0	0	2
a	1	0	1
ba	0	0	2
aa	0	0	2
aaa	0	0	2
aaaa	0	0	2

TABLE 4 – Table d'observation cohérente

$$F = \{ligne(s); s \in S, f(s) = \text{true}\}$$

$$\delta(ligne(s), a) = ligne(s \cdot a)$$

◊

Montrons que la définition 2.5 est correcte. Soit (S, E, f) une table d'observation close et cohérente. Soient $s \in S$ et $a \in \Sigma$. Il existe $s' \in S$ avec $ligne(s') \in Q$ tel que $ligne(s \cdot a) = ligne(s')$, par clôture. On a donc : $\delta(ligne(s), a) = ligne(s \cdot a) = ligne(s') \in Q$. Finalement, pour tout $s \in S$ et pour tout $a \in \Sigma$, on a bien $\delta(ligne(s), a) \in Q$.

Exemple : L'automate fini déterministe associé à la table d'observation close et cohérente définie par Table 4 est donné Figure 2.

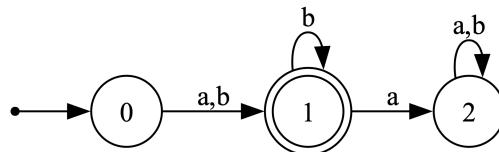


FIGURE 2 – Automate associé à Table 4

A travers plusieurs propositions issues de [1], nous allons maintenant montrer que l'automate construit de cette manière à partir d'une table d'observation reconnaît le langage \mathcal{L}_m et que c'est un automate avec un nombre minimal d'états. Les preuves de ces propositions se trouvent en annexe A.

Proposition 2.1 Si (S, E, f) est une table d'observation close et cohérente alors, en notant $M(S, E, f) = (\Sigma, Q, q_0, F, \delta)$ l'automate associé, pour tout $s \in S \cup S\Sigma$ on a : $\delta^*(q_0, s) = ligne(s)$ △

Proposition 2.2 Si (S, E, f) est une table d'observation close et cohérente alors, en notant $M(S, E, f) = (\Sigma, Q, q_0, F, \delta)$ l'automate associé, pour tout $s \in S \cup S\Sigma$ et $e \in E$ on a : $\delta^*(q_0, s \cdot e) \in F \iff f(s \cdot e) = \text{true}$. △

Proposition 2.3 Soit $T = (S, E, f)$ une table d'observation. On note n le nombre de lignes différentes pour S . Tout automate compatible avec f a au moins n états. △

Définition 2.6 Deux automates déterministes complets $A_1 = (\Sigma, Q_1, q_0^1, F_1, \delta_1)$ et $A_2 = (\Sigma, Q_2, q_0^2, F_2, \delta_2)$ sont isomorphes s'il existe une application

$$\phi : Q_1 \longrightarrow Q_2$$

telle que :

- ϕ est bijective ;
- $\phi(q_0^1) = q_0^2$;

- $\phi(F_1) = F_2$;
- $\phi(\delta_1(q, a)) = \delta_2(\phi(q), a)$ pour tout $q \in Q_1$, $a \in \Sigma$.

◊

Proposition 2.4 Soit (S, E, f) une table d'observation close et cohérente. Soit $M(S, E, f)$ l'automate fini déterministe associé. On note n son nombre d'états. Tout automate M' compatible avec f ayant n états ou moins est isomorphe à $M(S, E, f)$. △

Cette dernière proposition nous permet de conclure sur la minimalité de l'automate associé $M(S, E, f)$. Tout autre automate compatible avec f a soit plus d'états soit est isomorphe à $M(S, E, f)$.

3 L^* et son implémentation

3.1 Algorithme L^*

On revient dans le cas d'un enseignant minimalement compétent qui répond toujours correctement lorsque l'élève lui demande si un automate A reconnaît \mathcal{L}_m .

Pour apprendre un automate fini déterministe, on met en place l'algorithme L^* dont le code OCaml est ci-dessous (en annexe B se trouve tous les modules et fonctions).

```

let rec make_closed_and_consistent table (teacher:Teacher.teacher) =
  try
    check_closed table;
    check_consistent table ;
    ()
  with
    | Notclosed w -> Obs.add_to_s table w teacher.member;
      make_closed_and_consistent table teacher
    | Notconsistent w -> Obs.add_to_e table w teacher.member ;
      make_closed_and_consistent table teacher

let l_star (teacher:Teacher.teacher) =
  let t = Obs.initial_table teacher in
  let rec aux t =
    make_closed_and_consistent t teacher;
    let auto = construct_auto t in
    match (teacher.counter_example auto) with
    | None -> auto (*L'automate reconnaît le langage mystère*)
    | Some w ->
        (*Ajout du contre-exemple à S et de tous ses préfixes*)
        List.iter (fun x-> Obs.add_to_s t x teacher.member) (all_prefix w) ;
        aux t
  in
  aux t

```

L'algorithme consiste à créer une table d'observation close et cohérente $T = (S, E, f)$ à partir de tests d'appartenance, la fonction f indique si un mot appartient au langage \mathcal{L}_m . Une fois cette table construite, l'automate $M = (S, E, f)$ associé est construit. On demande alors à l'enseignant si l'automate M reconnaît le langage. Si ce n'est pas le cas, l'enseignant renvoie un contre-exemple qui est ajouté à S ainsi que tous ses préfixes.

Cet algorithme a été appliqué au langage \mathcal{L}_m décrit par la Figure 1. L'automate appris est décrit Figure 2.

3.2 Correction et terminaison de l'algorithme L^*

Correction Si l'algorithme L^* renvoie un automate alors il reconnaît le langage \mathcal{L}_m car aucun contre-exemple n'existe, il n'y a pas de mot qui appartient à un langage et pas à l'autre.

Terminaison Notons n le nombre minimum d'états d'un automate fini déterministe reconnaissant \mathcal{L}_m . À chaque appel à `make_close_and_consistent` on a

- Si la table d'observation $T = (S, E, f)$ n'est pas cohérente alors il existe $s, s' \in S$ tels que $ligne(s) = ligne(s')$ et $a \in \Sigma$, $e \in E$ tels que $f(sae) \neq f(s'ae)$. Le mot ae est ajouté à E et le nombre de lignes distinctes augmente d'au moins un car alors $ligne(s) \neq ligne(s')$ dans la nouvelle table d'observation.
- Si la table d'observation $T = (S, E, f)$ n'est pas close alors il existe $t \in S\Sigma$ tel que pour tout $s \in S$ $ligne(t) \neq ligne(s)$. Le mot t est alors ajouté à S et on a $ligne(t) \neq ligne(s)$. Le nombre de lignes distinctes sur S augmente d'au moins un.

Le nombre de lignes de la table d'observation de S croît de 1 à chaque appel à `make_close_and_consistant` et ce nombre est majoré par n . Il y a donc au plus n passages dans la boucle.

Remarque : L'élève fera au plus n conjectures d'automates (au plus $n - 1$ incorrectes et 1 correcte).

3.3 Complexité de l'algorithme L^*

La taille d'un mot de S est majorée par $l_m(S) = O(n^2)$ car un contre-exemple a une taille d'au plus n^2 . L'opération la plus coûteuse dans un appel à `make_close_and_consistant` est l'appel à `check_consistent` qui se fait en $O(|S|^2|\Sigma||E|l_m(S)) = O(n^5)$ dans le pire cas (si l'on ne considère pas la taille de l'alphabet comme paramètre du problème et donc une constante extérieure). Il y a au plus n appels à `make_close_and_consistant` donc la complexité totale dans le pire cas est en $O(n^6)$. La complexité de L^* est donc polynomiale. En pratique, on observe que la complexité en temps est en $O(n^{2.1})$ en moyenne et en $O(n^{2.5})$ dans le pire cas (Figure 3). J'ai pour cela généré aléatoirement des automates finis déterministes complets (100 automates) pour une taille d'alphabet donnée ($|\Sigma| = 2, 3, 4$), j'ai construit les enseignants associés puis j'ai mesuré le temps que mettait l'élève à les apprendre. J'ai enregistré le temps moyen et le pire temps pour un nombre d'états donné puis j'ai estimé α de $O(n^\alpha)$ en faisant une régression linéaire de $\log(\text{temps})$ en fonction de $\log(\text{nombre d'états})$.

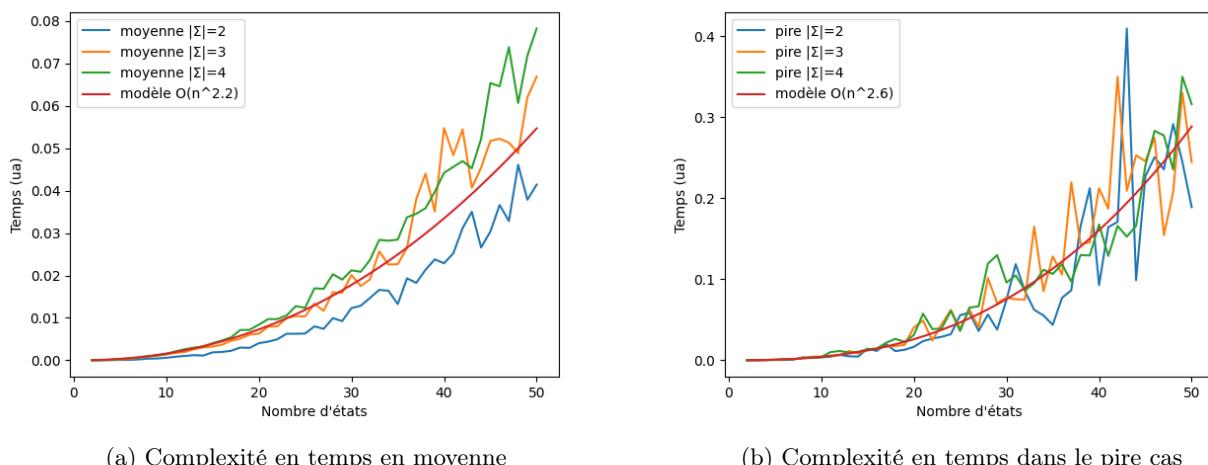
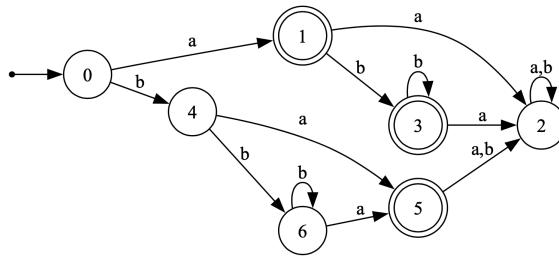
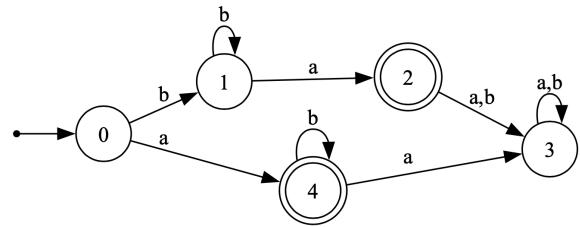


FIGURE 3 – Complexités expérimentales

3.4 Application de L^* aux langages réguliers

Il est possible d'apprendre une expression régulière. En mettant en place la traduction d'une expression régulière en automate fini puis en le rendant déterministe et complet, on obtient un automate fini déterministe complet reconnaissant l'expression régulière. Apprendre cet automate grâce à L^* permet de montrer que l'on peut apprendre un langage régulier et même minimiser l'automate reconnaissant le langage.

Dans l'exemple ci-dessous (Figure 4), l'automate A_m est l'automate fini déterministe connu par l'enseignant, il a été construit à partir de l'automate de Glushkov de l'expression régulière $(ab^*)|(bb^*a)$ puis cet automate a été déterminisé (grâce à l'automate des états). Le second automate (Figure 4 (b)) est celui construit par l'élève après avoir mis en place L^* . L'automate appris en bien minimal.

(a) A_m connu par l'enseignant(b) Automate construit grâce à L^* FIGURE 4 – Automates traduisant le langage $\mathcal{L} = \{(ab^*)|(bb^*a)\}$

4 Cas d'un enseignant imparfait

On peut imaginer que l'enseignant ne connaît pas explicitement l'automate A_m et ne puisse pas trouver de mot de la différence symétrique des deux langages facilement mais qu'il puisse toujours vérifier si un mot appartient au langage \mathcal{L}_m (par exemple s'il connaît l'automate sous forme non-déterministe). Dans ce cas, il est possible d'utiliser un oracle (ou enseignant) probabiliste pour déterminer si l'élève a une conjecture "à peu près correcte".

On suppose que l'on possède une fonction permettant de tirer aléatoirement une chaîne de caractère de Σ^* . On note $P(x)$ la probabilité de tirer l'élément $x \in \Sigma^*$. L'oracle est capable de répondre aux requêtes d'appartenance (`member`) et une nouvelle fonction `Ex` est définie. Lorsqu'elle est appelée, une chaîne de caractères $c \in \Sigma^*$ est choisie et la fonction renvoie (c, b) où b est un booléen indiquant si la chaîne de caractères est reconnue par A_m .

On pose ϵ la probabilité que l'appel `Ex()` renvoie un mot appartenant à la différence symétrique des deux langages (symbole \oplus).

On va alors montrer qu'à l'aide des deux fonctions `member` et `Ex` il est possible de construire une ϵ -approximation du langage inconnu.

Définition 4.1 V est une ϵ approximation de W si : $\sum_{x \in V \oplus W} P(x) \leq \epsilon$ ◊

Dans l'algorithme L^* , pour tester une conjecture, on réalise l'Algorithm 1 : CounterExample écrit en pseudo-code ci-dessous.

Il effectue $r_i = \lceil \frac{1}{\epsilon}(\ln(\frac{1}{\delta}) + (i+1)\ln(2)) \rceil$ appels à `Ex()` pour chaque i représentant le nombre de conjectures déjà effectuées. Si un mot appartient à un langage mais pas à l'autre alors il le renvoie, sinon il renvoie "Vrai".

Algorithm 1 : CounterExample

Entrée : Un automate fini déterministe $M(S, E, f)$, le nombre i de conjectures déjà effectuée, ϵ et δ

Sortie : Vrai si l'automate reconnaît le langage, un contre-exemple sinon

```

1   $r_i \leftarrow \lceil \frac{1}{\epsilon}(\ln(\frac{1}{\delta}) + (i+1)\ln(2)) \rceil$ 
2   $k \leftarrow 0$ 
3  Tant que  $k < r_i$  Faire
4     $(w, b) \leftarrow \text{Ex}()$ 
5    si  $b$  et  $w \notin \mathcal{L}(M)$  alors
6      Renvoyer  $w$ 
7    fin
8    sinon
9      si  $\text{non}(b)$  et  $w \in \mathcal{L}(M)$  alors
10     Renvoyer  $w$ 
11   fin
12 fin
13 FinTantQue
14 Renvoyer Vrai

```

Pour construire l'automate, l'algorithme L^* a besoin d'au plus $n - 1$ contre-exemples. On a donc besoin d'au plus $O(\frac{1}{\epsilon}((n-1)\ln(\frac{1}{\delta}) + n^2))$ appels à `Ex()` au total.

En effet,

$$\begin{aligned}\sum_{i=0}^{n-2} r_i &\leq \sum_{i=0}^{n-2} \frac{1}{\epsilon} \left(\ln\left(\frac{1}{\delta}\right) + (i+1)\ln(2) \right) + 1 \\ \sum_{i=0}^{n-2} r_i &\leq \frac{1}{\epsilon} \left((n-1)\ln\left(\frac{1}{\delta}\right) + \ln(2) \sum_{i=0}^{n-2} (i+1) \right) + n - 1 \\ \sum_{i=0}^{n-2} r_i &\leq O\left(\frac{1}{\epsilon} \left((n-1)\ln\left(\frac{1}{\delta}\right) + n^2 \right)\right)\end{aligned}$$

Après avoir essayé i conjectures, la probabilité que l'automate ne soit pas correct est d'au plus $(1-\epsilon)^{r_i}$. Au bout de $n-1$ conjectures, la probabilité que l'automate construit ne soit pas une ϵ -approximation A_m est majorée par δ .

En effet, par concavité, on a : $\sum_{i=0}^{n-2} (1-\epsilon)^{r_i} \leq \sum_{i=0}^{n-2} e^{-\epsilon r_i}$

La définition de r_i implique : $\sum_{i=0}^{n-2} (1-\epsilon)^{r_i} \leq \sum_{i=0}^{n-2} \frac{\delta}{2^{i+1}}$

Finalement, $\sum_{i=0}^{n-2} (1-\epsilon)^{r_i} \leq \delta$

5 Conclusion

L'algorithme L^* est un type d'apprentissage actif qui a été largement réutilisé et adapté pour apprendre des automates finis déterministes. Il peut être adapté à d'autres types d'automates comme les automates de Büchi [2]. Actuellement il peut être utilisé pour mieux comprendre comment fonctionnent des modèles d'apprentissage comme des réseaux de neurones [3].

Références

- [1] Dana Angluin, *Learning regular sets from queries and counterexamples*, Information and Computation, Volume 75, Issue 2, Pages 87-106, 1987.
- [2] Dana Angluin and Dana Fisman, *Learning regular omega languages*, Theoretical Computer Science, Volume 650, Pages 57-72, 2016.
- [3] Weiss, G., Goldberg, Y. and Yahav, E. *Extracting automata from recurrent neural networks using queries and counterexamples (extended version)*. Mach Learn, Volume 113, Pages 2877–2919, 2024.

A Preuves

Preuve Proposition 2.1.

Montrons par récurrence sur la longueur du mot $s \in S \cup S\Sigma$ la propriété \mathcal{P} .

$$\mathcal{P}(n) : \text{« Pour tout } s \in S \cup S\Sigma, |s| = n, \text{ on a } \delta^*(q_0, s) = \text{ligne}(s) \text{ »}$$

Initialisation

Pour $|s| = 0$ alors $s = \epsilon$ et $q_0 = \text{ligne}(s)$ par définition. On a donc $\delta^*(q_0, \epsilon) = \delta(q_0, \epsilon) = \text{ligne}(\epsilon)$.

Héritéité

Supposons que pour tout mot $s \in S \cup S\Sigma$ de longueur inférieure ou égale à n $\delta^*(q_0, s) = \text{ligne}(s)$.

Soit $s \in S \cup S\Sigma$ de longueur $n + 1$. On a donc $s = s' \cdot a$ avec $a \in \Sigma$.

Comme $s \in S \cup S\Sigma$, si $s \in S\Sigma$ alors on a $s' \in S$.

Si $s \in S$, comme S est clôt par préfixe on a aussi $s' \in S$.

On a alors $|s'| = n - 1$ et

$$\text{Par définition de } \delta : \delta^*(q_0, s) = \delta^*(q_0, s' \cdot a) = \delta(\delta^*(q_0, s'), a)$$

$$\text{Par hypothèse de récurrence : } \delta^*(q_0, s') = \delta(\text{ligne}(s'), a) = \text{ligne}(s' \cdot a) = \text{ligne}(s)$$

Preuve Proposition 2.2.

Montrons par récurrence sur la longueur du mot $e \in E$ la propriété \mathcal{P} .

$$\mathcal{P}(n) : \text{« Pour tout } e \in E, |e| = n, \text{ pour tout } s \in S \cup S\Sigma, \text{ on a } \delta^*(q_0, s \cdot e) \in F \iff f(s \cdot e) = \text{true} \text{ »}$$

Initialisation

Pour $|e| = 0$ on a $e = \epsilon$. Soit $s \in S \cup S\Sigma$.

$\delta^*(q_0, s \cdot e) = \delta^*(q_0, s) = \text{ligne}(s) \in F$ par définition.

— Si $s \in S$ alors par définition de S $f(s) = \text{true}$.

— Si $s \in S\Sigma$, comme la table d'observation est close il existe $s' \in S$ tel que $\text{ligne}(s) = \text{ligne}(s')$.

On a alors

$$\text{ligne}(s) \in F \iff \text{ligne}(s') \in F \iff f(s') = \text{true} \iff f(s) = \text{true}$$

Héritéité

Supposons que pour tout mot $e \in E$ de longueur inférieure ou égale à n , $P(|e|)$

Soit $e \in E$ de longueur $n + 1$. On a donc $e = a \cdot e'$ avec $a \in \Sigma$ et $|e| = n$.

Or $e \in E$ et E est clôt par suffixe d'où $e' \in E$.

Soit $s \in S \cup S\Sigma$. Comme la table d'observation est close, il existe $s' \in S$ tel que $\text{ligne}(s) = \text{ligne}(s')$.

Or on a

$$\delta^*(q_0, s \cdot e) = \delta^*(\delta^*(q_0, s), e)$$

$$\text{Par la proposition 2.1 : } \delta^*(q_0, s \cdot e) = \delta^*(\text{ligne}(s), e)$$

$$\text{Par clôture : } \delta^*(q_0, s \cdot e) = \delta^*(\text{ligne}(s'), a \cdot e')$$

$$\delta^*(q_0, s \cdot e) = \delta^*(\delta(\text{ligne}(s'), a), e')$$

$$\text{Par définition de } \delta : \delta^*(q_0, s \cdot e) = \delta^*(\text{ligne}(s' \cdot a), e')$$

$$\delta^*(q_0, s \cdot e) = \delta^*(q_0, s' \cdot a \cdot e')$$

Et on a $s' \cdot a \in S\Sigma$ donc par hypothèse de récurrence sur e'

$$\delta^*(q_0, s' \cdot a \cdot e') \in F \iff f(s' \cdot a \cdot e') = \text{true}$$

$$\iff \delta^*(q_0, s \cdot e) \in F \iff f(s \cdot e) = \text{true} \quad \text{Car } \text{ligne}(s) = \text{ligne}(s')$$

Preuve Proposition 2.3.

Soit $A = M(S, E, f) = (\Sigma, Q, q_i, F, \delta)$ un automate fini déterministe compatible avec f . On note n le nombre de lignes différentes dans la table d'observation $T = (S, E, f)$

Si $n = 1$ c'est immédiat, l'automate a bien au moins un état.

Sinon : soient $s, s' \in S$ tel que $ligne(s) \neq ligne(s')$. Il existe $e \in E$ tel que $f(se) \neq f(s'e)$. On a donc $q := \delta^*(q_0, se) \in F$ et $q' := \delta^*(q_0, s'e) \notin F$ (ou inversement). Ainsi $q \neq q'$.

Il y a donc au moins n états distincts dans Q .

Preuve Proposition 2.4.

Soit $M' = (\Sigma, Q', q'_0, F', \delta')$ compatible avec (S, E, f) avec au plus n .

Nous allons construire un isomorphisme ϕ entre M' et M .

Si $s, s' \in S$ tel que $\delta'^*(q'_0, s) = \delta'^*(q'_0, s')$ alors pour tout $e \in E$ $f(se) = f(s'e)$ car M' est compatible avec f , donc $ligne(s) = ligne(s')$.

On définit alors

$$\begin{aligned} \phi : \quad Q &\longrightarrow Q' \\ ligne(s) &\longmapsto \delta'^*(q'_0, s) \end{aligned}$$

Cette définition est correcte car elle ne dépend pas du représentant $s \in S$. ϕ est injective.

Par la proposition 2.3, comme M' est compatible avec f , M' a exactement n états. L'application ϕ est donc bijective par égalité des cardinaux de Q et Q' .

Montrons alors que ϕ est un isomorphisme :

- $\phi(q_0) = \phi(ligne(\epsilon)) = \delta'^*(q'_0, \epsilon) = q'_0$
- Montrons que $\phi(F) = F'$

$$\begin{aligned} ligne(s) \in F &\iff s \in L(M) && \text{Par définition de } F \\ &\iff f(s) = \text{true} && M \text{ compatible avec } f \\ &\iff s \in L(M') && M' \text{ compatible avec } f \\ &\iff \delta'^*(q'_0, s) \in F' && \text{Par définition de } F' \\ &\iff \phi(ligne(s)) \in F' && \text{Par définition de } \phi \end{aligned}$$

Donc $\phi(F) = F'$.

- Soient $q \in Q_1$, $a \in \Sigma$. Il existe $s' \in S$ tel que $ligne(s \cdot a) = ligne(s')$ car la table d'observation est fermée.

D'une part on a

$$\begin{aligned} \phi(\delta(ligne(s), a)) &= \phi(ligne(s \cdot a)) \\ &= \phi(ligne(s')) \\ &= \delta'^*(q'_0, s') && \text{Par définition de } \phi \end{aligned}$$

D'autre part on a

$$\delta'^*(\phi(ligne(s)), a) = \delta'^*(\delta'^*(q'_0, s), a) = \delta'^*(q'_0, s \cdot a)$$

Or par surjectivité de ϕ il existe $s'' \in S$ tel que $\delta'^*(q'_0, s \cdot a) = \phi(ligne(s''))$. On a alors pour tout $e \in E$ on a $\delta'^*(q'_0, s'' \cdot e) = \phi(ligne(s'')) = \delta'^*(q'_0, s \cdot a \cdot e)$ d'où $f(s''e) = f(sae)$ et donc il y a l'égalité des lignes $ligne(s'') = ligne(sa) = ligne(s')$. On a donc bien

$$\phi(\delta(ligne(s), a)) = \delta'^*(\phi(ligne(s)), a)$$

Donc ϕ est un isomorphisme de Q dans Q' .

B Programmes OCaml

B.1 Base

```
1 type letter = int
2 type word = letter list
3
4
5 type dfa =
6 {q0 : int;
7 nb_letters : int;
8 nb_states : int;
9 accepting : bool array;
10 delta : int array array}
11
12 let rec delta_star a q w =
13   match w with
14   | [] -> q
15   | x::xs -> delta_star a (a.delta.(q).(x)) xs
16 ;;
17
18 let create_dfa nb_letters states is_accepting=
19   let delt = Array.make_matrix states nb_letters (-1) in
20   {q0 =0;
21   nb_letters = nb_letters;
22   nb_states = states;
23   accepting = is_accepting;
24   delta = delt}
25 ;;
26
27
28 let add_transition automaton q a q' =
29   automaton.delta.(q).(a)<- q';
30
31 let random_auto n q =
32   (*n : Nombre de lettres dans l'alphabet, q : nombre d'états dans l'automate*)
33   let accepting_states = Array.make q false in
34   let nb_accepting = Random.int (q-1) in
35   for i =0 to nb_accepting do
36     accepting_states.(q-1-i)<- true;
37   done;
38   if Random.bool () then accepting_states.(0)<- true;
39   let auto = create_dfa n q accepting_states in
40   for i =0 to q-1 do
41     for j =0 to n-1 do
42       let q' = Random.int q in
43       add_transition auto i j q';
44     done;
45   done;
46   auto;;
47
48
```

B.2 L'enseignant

```
1 type teacher = {
2 nb_letters_t : int;
3 member : My_base.word -> bool;
```

```

4 counter_example : My_base.dfa -> My_base.word option;
5 }
6
7
8 exception Trouve of My_base.word
9
10 let shortest_word (a:My_base.dfa) =
11   let file = Queue.create () in
12   Queue.push (a.q0, []) file;
13   let vu = Array.make a.nb_states false in
14   try
15     while not (Queue.is_empty file) do
16       let (q,u) = Queue.pop file in
17       if a.accepting.(q)
18         then raise (Trouve u)
19       else
20         (for i = 0 to a.nb_letters -1 do
21           let q' = a.delta.(q).(i) in
22           if not vu.(q') then (
23             vu.(q') <- true;
24             Queue.push (q',(i::u)) file
25           )
26           done;
27         )
28       done;
29     None
30   with
31   | Trouve u -> Some (List.rev u)
32 ;;
33
34 let symmetric_difference (a:My_base.dfa) (a':My_base.dfa) =
35   let n = a.nb_states in
36   let n' = a'.nb_states in
37   let m = a.nb_letters in
38   let d = Array.make_matrix (n*n') m (-1) in
39   let ac = Array.make (n*n') false in
40   for q=0 to n-1 do
41     for q' = 0 to n'-1 do
42       if (a.accepting.(q) && not a'.accepting.(q')) ||
43         (a'.accepting.(q') && not a.accepting.(q)) then
44           ac.(n'*q+q') <- true;
45       for i =0 to m-1 do
46         d.(n'*q+q').(i) <- n'*a.delta.(q).(i) + a'.delta.(q').(i);
47         done;
48       done;
49     done;
50   {My_base.q0 = a.q0*n'+a'.q0 ;
51   nb_letters = m;
52   nb_states = n*n';
53   accepting = ac;
54   delta = d
55   }
56 ;;
57
58 let create_teacher a =
59   let member w =
60     let q = My_base.delta_star a (a.q0) w in

```

```

61     a.accepting.(q)
62     in
63     let ce a' =
64       let b = symmetric_difference a a' in
65       shortest_word b
66     in
67     {nb_letters_t = a.nb_letters;
68      member = member;
69      counter_example = ce}
70 ;;

```

B.3 La structure Trie

```

1 type trie = {
2   mutable final : bool;
3   children : (My_base.letter, trie) Hashtbl.t;
4 }
5
6 let create_node () = {
7   final = false;
8   children = Hashtbl.create 10;
9 }
10
11 let rec inside t word =
12   match word with
13   | [] -> t.final = true
14   | x::xs -> begin
15     match Hashtbl.find_opt t.children x with
16     | Some child -> inside child xs
17     | None -> false
18   end
19 ;;
20
21 let rec insert t word =
22   match word with
23   | [] -> t.final <- true
24   | x::xs ->
25     let child = begin
26       match Hashtbl.find_opt t.children x with
27       | Some t' -> t'
28       | None -> let n = create_node () in
29         Hashtbl.add t.children x n;
30         n
31     end
32     in
33     insert child xs
34 ;;
35
36 let words t =
37   let rec aux prefix tr =
38     let current = if tr.final then [List.rev prefix] else [] in
39     let rest =
40       Hashtbl.fold (fun a child acc ->
41         (aux (a :: prefix) child) @ acc
42       ) tr.children []
43     in
44     current @ rest

```

```

45   in
46   aux [] t
47 ;;
48
49 let fold f t acc =
50   let rec aux prefix t acc =
51     let acc' = if t.final then f (List.rev prefix) acc else acc
52     in
53       Hashtbl.fold (fun a child acc_inner ->
54         aux (a :: prefix) child acc_inner
55       ) t.children acc'
56   in
57   aux [] t acc
58 ;;
59
60 let iter f t =
61   let rec aux prefix t =
62     if t.final then f (List.rev prefix);
63     Hashtbl.iter (fun a child ->
64       aux (a :: prefix) child
65     ) t.children
66   in
67   aux [] t
68 ;;
69
70

```

B.4 La table d'observation

```

1 type observation_table =
2   {nb_let : int;
3    prefix : Tries.trie;
4    suffix : Tries.trie;
5    f : (My_base.word*My_base.word, bool) Hashtbl.t;
6    row_number : (My_base.word, int) Hashtbl.t;
7    unique_rows : (bool list, int) Hashtbl.t;
8    mutable next_row_id : int;}
9
10 let add_row t w l =
11   match Hashtbl.find_opt t.unique_rows l with
12   | None -> Hashtbl.replace t.unique_rows l t.next_row_id;
13   | _ -> Hashtbl.replace t.row_number w t.next_row_id ;
14   t.next_row_id <- t.next_row_id +1
15 | Some i -> Hashtbl.replace t.row_number w i
16 ;;
17
18 exception Compute_row
19 let compute_row t w =
20   let rec aux l =
21     match l with
22     | [] -> []
23     | e::es -> begin match Hashtbl.find_opt t.f (w,e) with
24       | Some x -> x :: (aux es)
25       | None -> raise Compute_row
26     end
27   in
28   let e_words = Tries.words t.suffix in

```

```

29     aux e_words
30   ;;
31
32 let update_rows t w =
33   let row = compute_row t w in
34   add_row t w row
35 ;;
36
37 let create_table nb_letters =
38   let prefix = Tries.create_node() in
39   Tries.insert prefix [];
40   let suffix = Tries.create_node() in
41   Tries.insert suffix [];
42   {nb_let = nb_letters;
43    prefix = prefix;
44    suffix = suffix;
45    f = Hashtbl.create 10;
46    row_number = Hashtbl.create 10;
47    unique_rows = Hashtbl.create 10;
48    next_row_id = 0;}
49
50 let nb_rows t =
51   Hashtbl.length t.unique_rows
52 ;;
53
54 let nb_letters t =
55   t.nb_let
56 ;;
57
58 exception Pas_row_nb
59 let get_row_number t w =
60   match Hashtbl.find_opt t.row_number w with
61   | Some n -> n
62   | None -> raise Pas_row_nb
63 ;;
64
65 exception Pas_compute
66 let compute_f t w e =
67   match Hashtbl.find_opt t.f (w,e) with
68   | Some b -> b
69   | None -> raise Pas_compute
70 ;;
71 ;;
72
73 let iter_s t g =
74   let words = Tries.words t.prefix in
75   List.iter g words
76 ;;
77
78 let iter_sa t g =
79   let words = Tries.words t.prefix in
80   let new_words = ref [] in
81   for i = 0 to (nb_letters t) - 1 do
82     new_words := (List.map (fun x -> x@[i]) words) @ !new_words
83   done;
84   List.iter g !new_words
85 ;;

```

```

86
87 exception Different of My_base.word
88 let separate_rows t u u' =
89   let mot_dif e =
90     if compute_f t u e <> compute_f t u' e then raise (Different e)
91   in
92   if get_row_number t u = get_row_number t u'
93     then None
94   else begin
95     try Tries.iter mot_dif t.suffix;
96     None
97   with
98   | Different e -> Some e
99 end
100
101 let add_to_s table w f =
102   if not (Tries.inside table.prefix w) then (
103     Tries.insert table.prefix w;
104     let aux x =
105       let wx = w@x in
106       let f_wx = (f wx) in
107       Hashtbl.add table.f (w,x) f_wx;
108       for i = 0 to nb_letters table -1 do
109         let w' = w@[i] in
110         let wax = w'@x in
111         let f_wax = (f wax) in
112         Hashtbl.replace table.f (w',x) f_wax;
113       done;
114     in
115     Tries.iter aux table.suffix;
116     let l = compute_row table w in
117     add_row table w l;
118     for i=0 to table.nb_let -1 do
119       let w' = w@[i] in
120       let l' = compute_row table w' in
121       add_row table w' l';
122     done;
123   ;;
124
125 let recreate_rows t =
126   t.next_row_id<- 0;
127   Hashtbl.clear t.unique_rows;
128   Hashtbl.clear t.row_number;
129
130   iter_s t (update_rows t);
131   iter_sa t (update_rows t);;
132
133 let add_to_e table w f =
134   if not (Tries.inside table.suffix w) then (
135     Tries.insert table.suffix w;
136     let aux x=
137       let xw = x@w in
138         Hashtbl.replace table.f (x,w) (f xw);
139         for i = 0 to nb_letters table -1 do
140           let x' = x@[i] in
141           let xaw = x'@w in
142             let f_xaw = (f xaw) in

```

```

143         Hashtbl.replace table.f (x',w) f_xaw;
144     done;
145     in
146     Tries.iter aux table.prefix;
147     recreate_rows table)
148 ;;
149
150 let initial_table (teach: Teacher.teacher) =
151   let table = create_table teach.nb_letters_t in
152   let f_0 = (teach.member []) in
153   Hashtbl.add table.f ([] ,[]) f_0 ;
154   update_rows table [];
155   for i=0 to table.nb_let - 1 do
156     Hashtbl.add table.f ([i], []) (teach.member [i]);
157     update_rows table [i];
158   done;
159   table
160 ;;
161

```

B.5 L'élève

```

1 let construct_auto t =
2   let nb_etats = Obs.nb_rows t in
3   let nb_letters = Obs.nb_letters t in
4   let acc = Array.make nb_etats false in
5   let delta = Array.make_matrix nb_etats nb_letters (-1) in
6   let test_fin w =
7     let row_number = Obs.get_row_number t w in
8     acc.(row_number) <- (Obs.compute_f t w [])
9   in
10  Obs.iter_s t test_fin;
11  let constru w =
12    let ligne = Obs.get_row_number t w in
13    for j = 0 to nb_letters -1 do
14      let q' = Obs.get_row_number t (w@[j]) in
15      delta.(ligne).(j) <- q';
16      done;
17    in
18  Obs.iter_s t constru ;
19
20 {My_base.q0 = Obs.get_row_number t [];
21 nb_letters = nb_letters;
22 nb_states = nb_etats;
23 accepting = acc;
24 delta = delta}
25 ;;
26
27 exception Notclosed of My_base.word
28 exception Notconsistent of My_base.word
29
30 let check_closed t =
31   let test = ref false in
32   let aux l_w u =
33     if Obs.get_row_number t u = l_w then
34       test := true;
35   in

```

```

36 let aux2 w =
37   test := false;
38   Obs.iter_s t (aux (Obs.get_row_number t w));
39   if not !test then
40     (raise (Notclosed w) )
41
42 in
43
44 Obs.iter_sa t aux2
45 ;;
46
47 let check_consistent t =
48   let m = Obs.nb_letters t in
49   let aux u v =
50     if Obs.get_row_number t v = Obs.get_row_number t u then
51       for i = 0 to m-1 do
52         let w1 = (u@[i]) in
53         let w2 = (v@[i]) in
54         match Obs.separate_rows t w1 w2 with
55         | None -> ()
56         | Some w ->
57           raise (Notconsistent (i::w))
58     done;
59   in
60   Obs.iter_s t (fun u -> (Obs.iter_s t (aux u)))
61 ;;
62
63 let all_prefix w =
64   let list_pref = ref [w] in
65   let rec aux u =
66     match u with
67     | [] -> ()
68     | _ :: xs -> list_pref := (List.rev xs):: !list_pref; aux xs
69   in
70   aux (List.rev w);
71   !list_pref
72 ;;
73
74 let rec make_closed_and_consistent table (teacher:Teacher.teacher) =
75   try
76     check_closed table;
77     check_consistent table ;
78     ()
79   with
80   | Notclosed w -> Obs.add_to_s table w teacher.member;
81     make_closed_and_consistent table teacher
82   | Notconsistent w -> Obs.add_to_e table w teacher.member ;
83     make_closed_and_consistent table teacher
84 ;;
85 let l_star (teacher:Teacher.teacher) =
86   let t = Obs.initial_table teacher in
87   let rec aux t =
88     make_closed_and_consistent t teacher;
89     let auto = construct_auto t in
90     match (teacher.counter_example auto) with
91     | None -> auto (*L'automate reconnaît le langage mystère*)
92     | Some w ->

```

```
93 (*Ajout du contre-exemple à S et de tous ses préfixes*)
94 List.iter (fun x-> Obs.add_to_s t x teacher.member) (all_prefix w) ;
95 aux t
96 in
97 aux t
98 ;;
```