Complexité du problème d'intersection d'automates

Michael Blondin
DIRO, Université de Montréal

13 décembre 2009

Résumé

Ce document est la synthèse d'un projet *honor* effectué sous la supervision de Pierre McKenzie dans le cadre du cours IFT4055. Nous nous intéressons au problème d'intersection d'automates (*Product Emptiness Problem*) qui consiste à déterminer si k automates n'acceptent aucun mot en commun. Nous étudions la complexité de ce problème pour plusieurs modèles et restrictions d'automates, puis nous traitons de son importance en complexité du calcul.

1 Introduction

La théorie des automates est un domaine fondamental qui a marqué la naissance de l'informatique. Plusieurs chercheurs ont étudié ce domaine et de nombreux résultats importants ont été découverts. Cependant, quelques questions, ayant des applications moins directes, restent sans réponse. Récemment, Karakostas, Lipton et Viglas [1] se sont intéressés aux conséquences de la découverte d'un algorithme efficace pour la résolution du problème d'intersection d'automates (appelé PEP, *Product Emptiness Problem*, par Lipton [2]). Un article récent, tiré du blog personnel de Richard Lipton [2], a relancé l'intérêt pour PEP. Ce problème consiste à déterminer si l'intersection de langages, acceptés par k automates finis déterministes, est vide. Il est montré dans [1] qu'un algorithme s'exécutant en temps $o(n^k)$ pour résoudre ce problème aurait des conséquences importantes en complexité du calcul. Cela permettrait, entre autres, de montrer que $\text{NL} \neq \text{P}$.

Puisque ce problème a été peu étudié dans la littérature pour les différents types d'automates, nous nous intéressons à classifier ses variantes pour plusieurs modèles connus.

Nous introduisons d'abord ces différents modèles, puis nous effectuons une revue de la littérature concernant les résultats connus. Par la suite, nous

donnons de nouveaux résultats sur la complexité du problème. Finalement, nous présentons les conséquences d'un algorithme plus efficace pour résoudre PEP ainsi qu'une proposition de démarche pour montrer qu'un tel algorithme n'existe pas.

2 Généralités et notation

Nous donnons quelques définitions pour familiariser le lecteur avec la notation utilisée. Cependant, nous supposons que le lecteur est familier avec plusieurs notions (langages, machine de Turing, classes de complexité, etc.) et le renvoyons à [3, 4] pour des définitions précises.

Définition 2.0.1. Soit Σ un alphabet, nous dénotons Σ^* l'ensemble de tous les mots pouvant être formés à partir de Σ sous la concaténation. Nous dénotons $\varepsilon \in \Sigma^*$ (*mot vide*) l'unique élément ayant la propriété $\varepsilon w = w = w\varepsilon$. Autrement dit, Σ^* forme un monoïde sous la concaténation avec ε comme élément neutre.

Définition 2.0.2. Soit Σ un alphabet et $S \subseteq \Sigma^*$. Nous disons que S est un *langage* sur *alphabet* Σ et que les éléments de S sont des *mots*. Pour $w \in S$, nous définissons $|w|$ comme étant la *longueur* de w et $|w|_\sigma$ comme étant le nombre de $\sigma \in \Sigma$ contenus dans w.

Exemple 2.0.3. Soit $w = 11010 \in \{0,1\}^*$, alors $|w| = 5, |w|_0 = 2, |w|_1 = 3$.

Définition 2.0.4. Soit G un graphe. Nous notons $V(G)$ l'*ensemble des sommets* de G et $E(G)$ l'*ensemble des arêtes* de G.

À moins d'indications contraires :
- log désigne le logarithme en base 2
- ppcm désigne le plus petit commun multiple
- $|m|$, où m est un entier, désigne la taille de l'encodage de m en binaire (ie. $|m| = \lfloor \log(m) + 1 \rfloor$)
- $|S|$, où S est un ensemble, désigne la cardinalité de S
- n désigne la taille de l'entrée d'un problème
- Pour $\sigma \in \Sigma$, $\sigma^i = \sigma \cdots \sigma$ (i fois)

3 Automates

3.1 Automates déterministes

Définition 3.1.1. Un *automate fini déterministe (DFA)* [3, p. 46] est un quintuplet $\langle Q, s, F, \Sigma, \delta \rangle$ où

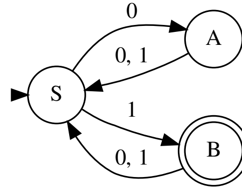
– Q est un ensemble fini (*états*)
– $s \in Q$ (*état initial*)
– $F \subseteq Q$ (*états finaux*)
– Σ est un ensemble fini (*alphabet*)
– $\delta : Q \times \Sigma \to Q$ (*fonction de transition*).

Un automate fini déterministe peut être représenté par un graphe dirigé où Q est l'ensemble des sommets. L'arête $(q, r) \in Q^2$ étiquetée par $\sigma \in \Sigma$ est ajoutée au graphe ssi $\delta(q, \sigma) = r$.

Exemple 3.1.2. L'automate $\langle \{S, A, B\}, S, \{B\}, \{0, 1\}, \delta \rangle$ où

$$\begin{array}{lllll} \delta(S,0) & = & A & & \\ \delta(S,1) & = & B & & \\ \delta(A,0) & = & \delta(A,1) & = & S \\ \delta(B,0) & = & \delta(B,1) & = & S \end{array}$$

peut être représenté de cette façon :



Définition 3.1.3. Soit $n = |Q|$, l'*encodage* $\langle A \rangle$ d'un automate fini déterministe A est la matrice d'adjacence $M \in \Sigma^{n \times n}$ telle que $M_{q,r} = \sigma$ ssi $\delta(q, \sigma) = r$. Bien que d'autres encodages de graphes dirigés avec coûts puissent être utilisés, nous utiliserons toujours la représentation matricielle. Notons que $|\langle A \rangle| \in O(n^2)$.

Définition 3.1.4. Soit $w = w_1 \cdots w_n$ où $w_i \in \Sigma$ et $A = \langle Q, s, F, \Sigma, \delta \rangle$ un automate fini déterministe. Nous disons que A *accepte* w s'il existe une suite d'états $q_1 \cdots q_n$ telle que

$$\begin{array}{lll} \delta(s, w_1) & = & q_1 \\ \delta(q_{i-1}, w_i) & = & q_i \quad (1 < i \leq n) \\ q_n & \in & F. \end{array}$$

Notons que si A accepte w alors cette suite d'états est unique. Nous disons que A *accepte* ε si $s \in F$. Nous dénotons $L(A) = \{w \in \Sigma^* \mid A \text{ accepte } w\}$ le *langage reconnu* par A.

Exemple 3.1.5. Le langage reconnu par l'automate de l'exemple 3.1.2 est l'ensemble des mots de longueur impaire terminant par 1.

3.2 Automates non déterministes

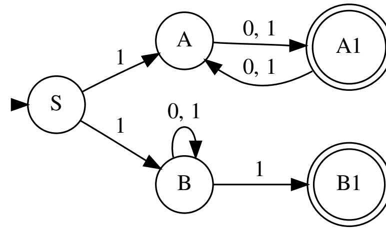
Définition 3.2.1. Un *automate fini non déterministe (NFA)* [3, p. 57] est un quintuplet $\langle Q, s, F, \Sigma, \delta \rangle$ où
– Q est un ensemble fini (*états*)
– $s \in Q$ (*état initial*)
– $F \subseteq Q$ (*états finaux*)
– Σ est un ensemble fini (*alphabet*)
– $\delta : Q \times \Sigma \to \mathcal{P}(Q)$ (*fonction de transition*).

Un automate fini non déterministe est une généralisation d'un automate fini déterministe. Il est possible d'utiliser la même représentation par un graphe dirigé en stockant un ensemble de lettres ($S \subseteq \Sigma$) par entrée plutôt qu'une seule lettre. Notons que ce modèle n'inclut pas d'ε–transition.

Exemple 3.2.2. L'automate $\langle \{S, A, A_1, B, B_1\}, S, \{A_1, B_1\}, \{0, 1\}, \delta \rangle$ où

$$\begin{array}{lclcl} \delta(S,0) & = & \emptyset & & \\ \delta(S,1) & = & \{A, B\} & & \\ \delta(A,0) & = & \{A_1\} & = & \delta(A,1) \\ \delta(A_1,0) & = & \{A\} & = & \delta(A_1,1) \\ \delta(B,0) & = & \{B\} & & \\ \delta(B,1) & = & \{B, B_1\} & & \end{array}$$

peut être représenté de la façon suivante :



Définition 3.2.3. Soit $w = w_1 \cdots w_n$ où $w_i \in \Sigma$ et $A = \langle Q, s, F, \Sigma, \delta \rangle$ un automate fini non déterministe. Nous disons que A *accepte* w s'il existe une suite d'états $q_1 \cdots q_n$ tels que

$$\begin{array}{lcll} q_1 & \in & \delta(s, w_1) & \\ q_i & \in & \delta(q_{i-1}, w_i) & (1 < i \leq n) \\ q_n & \in & F. & \end{array}$$

Nous disons que A *accepte* ε si $s \in F$. Nous dénotons $L(A) = \{w \in \Sigma^* \mid A \text{ accepte } w\}$ le *langage reconnu* par A.

Exemple 3.2.4. Le langage reconnu par l'automate de l'exemple 3.2.2 est l'union de l'ensemble des mots de longueur paire débutant par 1 et l'ensemble des mots débutant et terminant par 1.

Théorème 3.2.5. *Soit A un NFA. Il existe un algorithme qui, sur entrée $\langle A \rangle$, permet de vérifier si $L(A) = \emptyset$ en temps linéaire.*

Démonstration. Il suffit d'effectuer une recherche en largeur sur A et de stocker un bit indiquant si un état final a été rencontré lors du parcours. S'il n'y a plus d'états à visiter et qu'aucun état final n'a été rencontré alors $L(A) = \emptyset$, autrement $L(A) \neq \emptyset$ puisqu'un chemin de l'état initial vers un état final définit un mot. Il est bien connu, qu'étant donné la matrice d'adjacence d'un graphe, le parcours en largeur s'effectue en temps linéaire. [8, p. 302] □

Théorème 3.2.6. *Soient A, B des automates finis non déterministes. Il existe un algorithme qui, sur entrée $\langle A, B \rangle$, construit un automate fini non déterministe C tel que $L(C) = L(A) \cap L(B)$ et $|Q_C| = |Q_A| \cdot |Q_B|$. Cet algorithme s'exécute en temps $O(n^2)$. [3, th. 4.8]*

Puisque l'intersection est une opération associative, nous avons (par induction) :

Corollaire 3.2.7. *Soient $A_1, \ldots, A_k$ des automates finis non déterministes. Il existe un algorithme qui, sur entrée $\langle A_1, \ldots, A_k \rangle$, construit un automate fini non déterministe A tel que $L(A) = L(A_1) \cap \cdots \cap L(A_k)$ et $|Q_A| = |Q_{A_1}| \cdots |Q_{A_k}|$. Cet algorithme s'exécute en temps $O(n^k)$.*

Corollaire 3.2.8. *Soit $L = L(A_1) \cap \cdots \cap L(A_k)$. Si $L \neq \emptyset$, alors existe un mot w tel que $w \in L$ et $|w| \leq |Q_{A_1}| \cdots |Q_{A_k}|$.*

3.3 Automates bidirectionnels

Définition 3.3.1. Un *automate fini non déterministe bidirectionnel (2NFA)* [5, 6] est un quintuplet $\langle Q, s, F, \Sigma, \delta \rangle$ où
- Q est un ensemble fini (*états*)
- $s \in Q$ (*état initial*)
- $F \subseteq Q$ (*états finaux*)
- Σ est un ensemble fini tel que $\Sigma \cap \{\vdash, \dashv\} = \emptyset$ (*alphabet*)
- $\delta : Q \times (\Sigma \cup \{\vdash, \dashv\}) \to \mathcal{P}(Q \times \{L, R\})$ (*fonction de transition*)

Un 2NFA est un automate muni d'une tête de lecture bidirectionnelle. Sur entrée $w \in \Sigma^*$, le mot $\vdash w \dashv$ est placé sur le ruban de lecture de l'automate et la tête de lecture est placée sur $\vdash$. Soient $q, r \in Q$ et $\sigma \in \Sigma \cup \{\vdash, \dashv\}$. Lorsque la lettre σ est lue, $(r, R) \in \delta(q, \sigma)$ indique que l'automate peut effectuer une

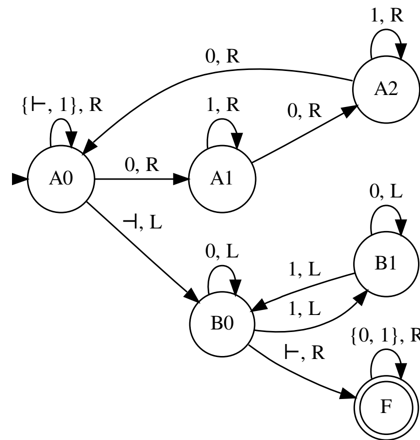
transition de l'état q vers l'état r en déplaçant sa tête de lecture vers la droite (L pour la gauche). Si la tête de lecture est à l'extrémité $\dashv$ ($\vdash$) alors un déplacement vers la droite (gauche) garde la tête de lecture à la même position.

Il est possible de penser à un 2NFA comme étant une machine de Turing non déterministe sans ruban d'écriture. L'acceptation d'un mot se fait de façon similaire.

Définition 3.3.2. Soient $w \in \Sigma^*$ et A un 2NFA. Nous disons que A *accepte* w s'il existe, à partir de l'état initial, une suite de transitons valides de A qui mènent à un état final au moment où la tête de lecture est à la dernière position du ruban ($\dashv$). Nous dénotons $L(A) = \{w \in \Sigma^* \mid A \text{ accepte } w\}$ le *langage reconnu* par A.

Définition 3.3.3. Un *automate fini déterministe bidirectionnel (2DFA)* est un 2NFA pour lequel la fonction de transition retourne toujours un ensemble de taille au plus 1, plus formellement $\forall q \in Q, \forall \sigma \in \Sigma \cup \{\vdash, \dashv\}, |\delta(q, \sigma)| \leq 1$. De façon similaire, un 2DFA est une machine de Turing déterministe sans ruban d'écriture.

Exemple 3.3.4. Le 2DFA $\langle \{A_0, A_1, A_2, B_0, B_1, F\}, A_0, \{F\}, \{0, 1\}, \delta \rangle$ suivant :



reconnaît l'ensemble des mots w tels que $|w|_0$ est un multiple de 3 et $|w|_1$ est un multiple de 2.

Le résultat suivant peut sembler contre-intuitif. Il montre que l'ajout d'une tête de lecture bidirectionnelle aux NFAs ne permet pas de reconnaître plus de langages.

Lemme 3.3.5. *Soit A un 2NFA à n états. Il existe un DFA B et une constante c tels que B possède moins de $c \cdot e^{(n^2)}$ états et $L(A) = L(B)$ [9, th. 4.2].*

Corollaire 3.3.6. *Soit A un 2NFA tel que $L(A) \neq \emptyset$, alors il existe un mot w tel que $w \in L(A)$ et $|w| \leq c \cdot e^{(n^2)}$.*

3.4 Automates à balayage

Définition 3.4.1. Un *automate fini non déterministe à balayage (SNFA)* (*quasi-sweeping automaton*) [6] est un automate bidirectionnel pour lequel les changements de direction de la tête de lecture et les choix non déterministes s'effectuent seulement aux extrémités du ruban (ie. $\vdash$ ou $\dashv$).

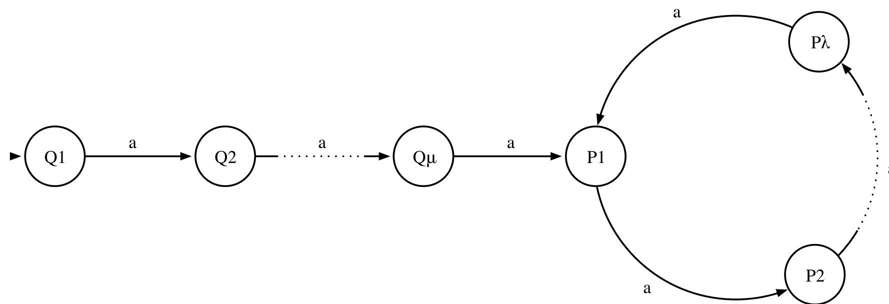
Définition 3.4.2. Un *automate fini déterministe à balayage (SDFA)* (*sweeping automaton*) [7] est est un SNFA pour lequel la fonction de transition retourne toujours un ensemble de taille au plus 1, plus formellement $\forall q \in Q, \forall \sigma \in \Sigma \cup \{\vdash, \dashv\}, |\delta(q, \sigma)| \leq 1$.

Exemple 3.4.3. L'automate de l'exemple 3.3.4 est un automate fini déterministe à balayage (SDFA).

3.5 Automates unaires

Définition 3.5.1. Un *automate fini unaire* est un automate fini avec alphabet unaire. Nous nommons normalement cet alphabet $\{a\}$. Pour désigner la variante unaire d'un modèle d'automate, nous préfixons son nom par la lettre u (ex : uDFA, u2NFA).

Tel que noté dans [10], un uDFA connexe a toujours la forme suivante :



Nous nommons *queue* le sous-graphe induit formé des états q_1 à p_1 et *cycle* le sous-graphe induit formé des états p_1 à p_λ. Un uDFA connexe peut donc être représenté par la paire (μ, λ) ainsi que la liste de ses états finaux. Notons que $\mu \geq 0, \lambda \geq 1$. Dans le cas général, un uDFA est une union disjointe de tels graphes. Si nous sommes seulement intéressés par la taille nécessaire pour représenter un langage, nous pouvons considérer les uDFAs comme étant connexes.

Lemme 3.5.2. *Soient A_1, A_2 deux uDFAs de taille $(\mu_1, \lambda_1), (\mu_2, \lambda_2)$. Le langage $L = L(A_1) \cap L(A_2)$ est accepté par un uDFA de taille* $(\max(\mu_1, \mu_2), \mathrm{ppcm}(\lambda_1, \lambda_2))$ *[10, th. 4].*

Puisque max et ppcm sont des opérations associatives, nous avons (par induction) :

Corollaire 3.5.3. *Soient $A_1, \ldots, A_k$ des uDFAs. Le langage $L = L(A_1) \cap \cdots \cap L(A_k)$ est accepté par un uDFA de taille* $(\max(\mu_1, \ldots, \mu_k), \mathrm{ppcm}(\lambda_1, \ldots, \lambda_k))$.

Corollaire 3.5.4. *Soit $L = L(A_1) \cap \cdots \cap L(A_k)$. Si $L \neq \emptyset$, alors il existe un mot w tel que $w \in L$ et $|w| \leq \max(\mu_1, \ldots, \mu_k) + \mathrm{ppcm}(\lambda_1, \ldots, \lambda_k)$.*

4 Problème d'intersection d'automates (PEP)

4.1 Définition

Définition 4.1.1. Définissons le problème PEP_k (du nom anglais *Product Emptiness Problem*) :

Données : $A_1, \ldots, A_k$, k automates finis déterministes.
Problème : Déterminer si $L(A_1) \cap \cdots \cap L(A_k) = \emptyset$.

Les variantes suivantes sont définies de façon similaire en modifiant le type d'automates :

Nom	**Type d'automates**
NPEP_k	Non déterministes
uPEP_k	Alphabet unaire $\{a\}$
SPEP_k	À balayage
$\mathrm{2PEP}_k$	Avec tête de lecture bidirectionnelle

À défaut de trouver une notation standard dans la littérature, nous proposons le préfixe « u » pour signifier *unaire* et « S » pour signifier *à balayage*. Il est possible d'utiliser plusieurs préfixes pour définir une variante du langage.

Définition 4.1.2. PEP $= \bigcup_{k \geq 1} \mathrm{PEP}_k$. Les variantes pour les différents modèles d'automates sont définies de façon similaire. Il est sous-entendu ici que l'encodage est choisi de façon à ce que $L(\mathrm{PEP}_k) \cap L(\mathrm{PEP}_{k'}) = \emptyset$, lorsque $k \neq k'$.

4.2 Automates standards

Théorème 4.2.1. *Étant donné $w = \langle A_1, \ldots, A_k \rangle$, il existe un algorithme qui décide si $w \in$ (N)PEP$_k$ en temps $O(n^k)$.*

Démonstration. Il suffit de construire un automate A reconnaissant l'intersection des k automates, puis de vérifier si $L(A) = \emptyset$. Par le corollaire 3.2.7 et le théorème 3.2.5, nous concluons que cet algorithme s'exécute en temps $O(n^k)$. □

Théorème 4.2.2. *PEP est PSPACE-complet [11].*

Corollaire 4.2.3. *NPEP est PSPACE-complet.*

Théorème 4.2.4. *NPEP$_k$ $\in$ NL.*

Démonstration. Nous donnons une machine de Turing non déterministe pour $\overline{\mathrm{NPEP}_k}$ fonctionnant en espace logarithmique. Puisque NL est fermé sous la complémentation [12], cela montre que $\mathrm{NPEP}_k \in$ NL. Voici cette machine :

$M(\langle A_1, \ldots, A_k \rangle)$:= Pour $i \leftarrow 1..k$
  $q_i \leftarrow s_i$
 $j \leftarrow 0$
 Tant que $(j \leq |Q_1| \cdots |Q_k|)$
  Si $(\forall i, q_i \in F_i)$ alors
   Retourner 1
  Choisir $\sigma \in \Sigma$ de façon non déterministe
  Pour $i \leftarrow 1..k$
   Choisir $r \in \delta_i(q_i, \sigma)$ de façon non déterministe
   $q_i \leftarrow r$
  $j \leftarrow j + 1$
 Retourner 0

La machine M choisit une lettre à chaque itération et effectue une transition valide sur chaque automate. Si un mot w est accepté par tous les automates, alors il existe un chemin de calcul de M qui choisira w et les transitions qui feront accepter w par $A_1, \ldots, A_k$. Par le corollaire 3.2.8, nous savons que si un tel mot existe, alors il en existe un de taille inférieure ou égale à

$|Q_1| \cdots |Q_k|$; M rejette donc après ce nombre d'itérations. La machine utilise $O((k+1)\log n)$ espace, puisqu'un seul état par automate ainsi qu'un compteur sur le nombre d'itérations sont stockés ; ceux-ci peuvent être encodés par $\log n$ bits. □

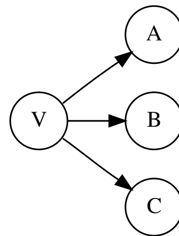
Corollaire 4.2.5. *$uPEP_k, PEP_k, uNPEP_k \in NL$.*

Théorème 4.2.6. *PEP_1 est NL-complet, selon la réductibilité log-espace.*

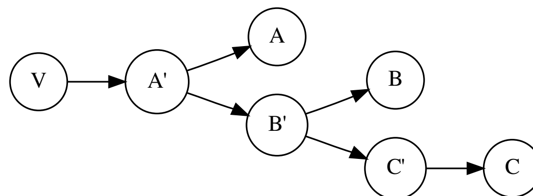
Démonstration. Nous donnons une réduction log-espace de PATH [4, p. 259] (REACHABILITY [13, p. 3]) vers $\overline{\text{PEP}_1}$. Puisque PATH est NL-complet [13, th. 16.2] et NL est fermé sous la complémentation [12], cela complète la preuve. Nous disons que $a \to b$ dans un graphe G si l'arc (a, b) est dans $E(G)$. Étant donné un graphe G, nous construisons un graphe G'. Soit v un sommet de G tel que $v \to v_1, \ldots, v \to v_k$. Nous ajoutons à G' les sommets v, v_i, v'_i $(1 \leq i \leq k)$, ainsi que les arcs suivants :

$$\begin{array}{lcll} v & \to & v'_1 & \\ v'_i & \to & v_i & (1 \leq i \leq k) \\ v'_i & \to & v'_{i+1} & (1 \leq i < k) \end{array}$$

Par exemple, le graphe suivant :



devient :



Une simple induction montre que $v \to v_i$ $(1 \leq i \leq k)$ dans G'. Du même coup, s'il existe un chemin de $v \in V(G)$ vers $w \in V(G)$ alors il existe un chemin de $v \in V(G')$ vers $w \in V(G')$. Notons que s'il existe un chemin de $v \in V(G')$ vers $w \in V(G')$ et que $v \in V(G)$ alors il existe aussi un chemin entre ces deux sommets dans G. Tous les sommets de G' sont de degré sortant 2, le graphe peut donc représenter un automate déterministe sur alphabet $\{0, 1\}$.

Nous rappelons que le problème PATH consiste à déterminer s'il existe un chemin de $s \in V(G)$ vers $t \in V(G)$ étant donné $\langle G, s, t\rangle$. Nous pouvons ainsi construire un automate A formé par le graphe G' avec état initial s et état final t. S'il existe un chemin de s vers t dans G' alors ce chemin définit un mot accepté par A. S'il n'existe pas de chemin alors A n'accepte aucun mot.

Cette réduction s'effectue en espace logarithmique (et en temps linéaire). □

Puisqu'il est possible d'ajouter des automates acceptant Σ^* sans modifier le résultat de l'intersection, nous avons le corollaire suivant :

Corollaire 4.2.7. *PEP_k, $NPEP_k$ sont NL-complet, selon la réductibilité log-espace.*

Théorème 4.2.8. *$uNPEP_1$ est NL-complet, selon la réductibilité log-espace.*

Démonstration. Nous donnons une réduction log-espace de PATH [4, p. 259] vers $\overline{\text{uNPEP}_1}$. Puisque PATH est NL-complet [13, th. 16.2] et que NL est fermé sous la complémentation [12], cela complète la preuve. Étant donné $\langle G, s, t\rangle$, nous construisons l'automate A sur alphabet $\{a\}$ formé du graphe G, l'état initial s et l'état final t. Si $v \to w$ alors $\delta(v, a) = w$. S'il existe un chemin de s vers t dans G, ce chemin définit un mot accepté par A. S'il n'existe pas de chemin alors A n'accepte aucun mot.

Cette réduction s'effectue en espace logarithmique (et en temps linéaire). □

Corollaire 4.2.9. *$uNPEP_k$ est NL-complet, selon la réductibilité log-espace.*

Théorème 4.2.10. *$uPEP_k \in L$.*

Démonstration. Nous donnons une machine de Turing M déterministe pour

uPEP$_k$:

$M(\langle A_1, \ldots, A_k \rangle)$:= Vérifier que A_i est un uDFA, $\forall i \in [k]$
Pour $i \leftarrow 1..k$
  Calculer (μ_i, λ_i)
$\mu \leftarrow \max(\mu_1, \ldots, \mu_k)$
$\lambda \leftarrow \text{ppcm}(\lambda_1, \ldots, \lambda_k)$
$p \leftarrow \mu + \lambda$
Si $\forall i \in [k], s_i \in F_i$ alors retourner 1
Sinon $\forall i \in [k], q_i \leftarrow s_i$
Pour $j \leftarrow 1..p$
  Pour $i \leftarrow 1..k$
    $q_i \leftarrow \delta(q_i, a)$
Si $\forall i \in [k], s_i \in F_i$ alors retourner 1
Retourner 0

La machine M calcule une borne supérieure p sur la taille du plus petit mot accepté par les k automates, tel que donné par le corollaire 3.5.4. Par la suite, les k automates sont simulés sur chacun des mots de taille inférieure ou égale à p. Si l'intersection n'est pas vide, alors M retourne 1, sinon M retourne 0.

Montrons maintenant que M fonctionne en espace logarithmique. Notons d'abord que les variables μ_i, λ_i, q_i nécessitent seulement $O(\log n)$ bits puisque ce sont des entiers bornés par la taille des automates.

Pour calculer (μ_i, λ_i), il ne suffit que d'initialiser un compteur m à 0, de *débuter* à l'état initial s_i puis d'*avancer* dans l'automate A_i (en utilisant l'unique transition) tout en incrémentant m. À chaque transition, nous vérifions si l'état actuel est de degré entrant 2. Lorsqu'un tel état est atteint, nous sommes à l'état qui relie la queue et le cycle de A_i. Le calcul de (μ_i, λ_i) en découle et nous n'avons utilisé qu'un espace logarithmique.

Nous savons que $|\max(a,b)|, |\text{ppcm}(a,b)| \in O(|a| + |b|)$ et que le calcul de $\max(a,b), \text{ppcm}(a,b)$ peut être effectué en espace linéaire en $|a|+|b|$. Puisque μ_i, λ_i sont de taille logarithmique (ie. $|\mu_i|, |\lambda_i| \in O(\log n)$), nous concluons que μ, λ se calculent en espace logarithmique. □

Définition 4.2.11. Le problème suivant (*Directed Forest Accessibility*) est défini dans [14] :

Données :	Un graphe dirigé acyclique G où tous les sommets sont de degré sortant 0 ou 1. Deux sommets u et v.
Problème :	Déterminer s'il existe un chemin dirigé de u vers v

Lemme 4.2.12. *Le problème « Directed Forest Accessibility » est L-complet, selon la réductibilité NC^1 [14].*

Théorème 4.2.13. *Le problème uPEP$_1$ restreint aux automates avec un seul état final est L-complet, selon la réductibilité NC^1.*

Démonstration. Nous donnons une réduction de “Directed Forest Accessibility” vers uPEP$_1$ en construisant un automate possédant un seul état final. Étant donné $\langle G, u, v\rangle$ comme décrit à la définition 4.2.11, nous construisons un automate A sur alphabet $\{a\}$. Les états de A sont les sommets de G, l'état initial est u, l'état final est v et la fonction de transition est :

$$\delta(x, a) = y \quad \Leftrightarrow \quad (x, y) \in E(G)$$

S'il existe un chemin de u vers v dans G alors ce chemin définit un mot accepté par A. Dans le cas contraire, il est impossible d'atteindre l'unique état final et A n'accepte donc aucun mot. □

Corollaire 4.2.14. *uPEP$_k$ est L-complet.*

Définition 4.2.15. Le problème des congruences linéaires restreintes (L-CON) est défini dans [17] :

Données :	$A \in \mathbb{Z}^{a\times b}, \mathbf{b} \in \mathbb{Z}^a, q \in \mathbb{Z}$ où la factorisation de q est donnée en unaire.
Problème :	Déterminer s'il existe $\mathbf{x} \in \mathbb{Z}^b$ tel que $A\mathbf{x} \equiv \mathbf{b} \pmod q$.

Lemme 4.2.16. *LCON $\in NC^3 \subseteq P$ [17].*

Théorème 4.2.17. *Le problème uPEP restreint aux automates avec un seul état final est dans $NC^3 \subseteq P$.*

Démonstration. Étant donné k uDFAs $A_1, \ldots, A_k$ ayant chacun un seul état final, nous donnons un algorithme qui détermine, en temps polynômial, si un mot est accepté par tous les automates. Puisque P est fermé sous la complémentation, cela complète la preuve.

Soit d_i la distance entre l'état initial et l'état final de l'automate A_i. Si l'unique état final de A_i se situe dans sa queue, alors A_i accepte un seul mot et il suffit de vérifier que les k automates acceptent ce mot. Nous pouvons donc supposer que l'état final se situe dans le cycle (ie. $d_i \geq \mu_i$).

Tous les mots acceptés par A_i sont de la forme $a^{d_i + x\lambda_i}$ où x est un entier non négatif. En d'autres termes, un mot w accepté par A_i est de longueur $|w| \equiv d_i \pmod{\lambda_i}$. Un mot w accepté par tous ces automates est donc de longueur $|w| \equiv d_i \pmod{\lambda_i}$ pour tout $i \in [k]$. Considérons le système

d'équations linéaires S suivant :

$$\begin{pmatrix} 1 & -\lambda_1 & 0 & \cdots & 0 \\ 1 & 0 & -\lambda_2 & \cdots & 0 \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ 1 & 0 & 0 & \cdots & -\lambda_k \end{pmatrix} \begin{pmatrix} m \\ x_1 \\ x_2 \\ \vdots \\ x_k \end{pmatrix} = \begin{pmatrix} d_1 \\ d_2 \\ \vdots \\ d_k \end{pmatrix}$$

La $i^{\text{ème}}$ équation de S est $m - x_i\lambda_i = d_i$, ce qui est équivalent à $m = d_i + x_i\lambda_i \equiv d_i \pmod{\lambda_i}$. Nous remarquons qu'il existe un mot accepté par tous les automates ssi il existe une solution à ce système. Posons $\Lambda = \lambda_1 \cdots \lambda_k$. Nous définissons le système S' de façon similaire à S en ajoutant les mêmes équations mod Λ. La $i^{\text{ème}}$ équation de S' est $m - x_i\lambda_i \equiv d_i \pmod{\Lambda}$, ce qui est équivalent à $m \equiv d_i + x_i\lambda_i \pmod{\Lambda}$.

Montrons qu'il existe une solution à S ssi il existe une solution à S'.

$\Rightarrow$) Soit $(m, x_1, \ldots, x_k)$ une solution pour S. Nous avons $\forall i \in [k]$, $m = d_i + x_i\lambda_i$ et du même coup :

$$m \bmod \Lambda = d_i + y_i\lambda_i \bmod \Lambda$$

Donc pour tout $i \in [k]$, $m \equiv d_i + y_i\lambda_i \pmod{\Lambda}$ et ainsi $(m, y_1, \ldots, y_k)$ est une solution pour S'.

$\Leftarrow$) Soit $(m, x_1, \ldots, x_k)$ une solution pour S'. Posons $m' = m \bmod \Lambda$. Nous avons $\forall i \in [k]$:

$$\begin{aligned} m' &= m \bmod \Lambda \\ &= d_i + x_i\lambda_i \bmod \Lambda \\ &= d_i + x_i\lambda_i + z_i\Lambda \\ &= d_i + x_i\lambda_i + z_i \cdot \lambda_1 \cdots \lambda_k \\ &= d_i + (x_i + z_i\Lambda/\lambda_i)\lambda_i \end{aligned}$$

Posons $y_i = x_i + z_i\Lambda/\lambda_i$. Notons que par la définition de Λ, y_i est un entier. Nous avons donc, $\forall i \in [k]$, $m' = d_i + y_i\lambda_i$ et ainsi $(m', y_1, \ldots, y_k)$ est une solution pour S.

Soit n la taille de l'entrée du problème. Puisque $d_i, \lambda_i \leq n$, il est possible de factoriser d_i, λ_i en NC_1 [17]. À partir de ces factorisations, il est simple d'obtenir la factorisation de Λ (encodée en unaire). Il suffit donc de construire S' et de vérifier s'il existe une solution. Par le lemme 4.2.16, cet algorithme s'exécute en temps polynômial. Plus précisement, la transformation de l'exemplaire uPEP_1 en un exemplaire de LCON s'effectue en NC_1 puisqu'un circuit booléen de profondeur logarithmique peut extraire μ_i, λ_i, construire Λ et produire le système de congruences. Nous avons donc, $\mathrm{uPEP}_1 \in \mathrm{NC}^3$. □

Théorème 4.2.18. *uPEP* $\in$ *coNP*

Démonstration. Nous montrons que $\overline{\text{uPEP}} \in$ NP en donnant une machine de Turing non déterministe pour $\overline{\text{uPEP}}$ s'exécutant en temps polynômial :

$$
\begin{aligned}
M(\langle A_1, \ldots, A_k \rangle) \quad := \quad & \max \leftarrow |Q_1| \cdots |Q_k| \\
& \text{Choisir } m \in [0..\max] \text{ de façon non déterministe} \\
& \text{Pour } i \leftarrow 1..k \\
& \quad \text{Calculer } (\mu_i, \lambda_i) \\
& \quad \text{Si } m \leq \mu_i \text{ alors} \\
& \quad\quad p_i \leftarrow m \\
& \quad \text{Sinon} \\
& \quad\quad p_i \leftarrow \mu_i + [(m - \mu_i) \mod \lambda_i] \\
& \text{Si } \forall i \in [n], a^{p_i} \in L(A_i) \text{ alors retourner 1} \\
& \text{Sinon retourner 0}
\end{aligned}
$$

De façon plus intuitive, la machine M choisit un mot a^m et vérifie si ce mot est accepté par tous les automates. Par le corollaire 3.2.8, s'il existe un tel mot, alors au moins un chemin de calcul de M accepte. Cependant, la longueur de ce mot n'est pas nécessairement de taille polynômiale, il faut donc être plus astucieux en exécutant chaque automate sur un mot *équivalent* de taille bornée par $|Q_i| = \mu_i + \lambda_i$.

Nous profitons du fait qu'un uDFA soit formé d'une queue et d'un cycle tels que décrits en 3.5. Soient m, p_i tels que définis dans M. Si $m \leq \mu_i$, alors $a^m \in L(A_i) \Leftrightarrow a^{p_i} \in L(A_i)$. Si $m > \mu_i$ alors $p_i = \mu_i + [(m - \mu_i) \mod \lambda_i]$. Supposons que nous exécutons A_i sur le mot a^m. Les μ_i premières lettres sont lues par les états de la queue, puis nous entrons dans le cycle de A_i. À ce point, nous lisons le mot $a^{m-\mu_i}$. Puisque cette section de l'automate est un cycle de taille λ_i, l'automate termine de lire le mot au $[(m-\mu_i) \mod \lambda_i]^{\text{ème}}$ état du cycle (en comptant à partir de 0). Nous avons donc :

$$
\begin{aligned}
a^m \in L(A_i) \quad & \Leftrightarrow \quad a^{\mu_i} a^{[(m-\mu_i) \mod \lambda_i]} \in L(A_i) \\
& \Leftrightarrow \quad a^{\mu_i + [(m-\mu_i) \mod \lambda_i]} \in L(A_i) \\
& \Leftrightarrow \quad a^{p_i} \in L(A_i)
\end{aligned}
$$

Tel que vu précédemment, le calcul de (μ_i, λ_i) s'effectue en espace logarithmique et ainsi en temps polynômial. Nous devons donc seulement montrer que le calcul de $[(m - \mu_i) \mod \lambda_i]$ s'effectue en temps polynômial.

Soit n la taille de l'entrée de M. Nous avons $|m| \leq \log |Q_1| + \ldots + \log |Q_k| \leq n \log n$ et $|\mu_i|, |\lambda_i| \leq \log |Q_i| \leq n$. En utilisant un algorithme simple de soustraction et de division, nous pouvons calculer $[(m - \mu_i) \mod \lambda_i]$ en temps $O(n^2 \log n)$ [16]. $\square$

4.3 Automates bidirectionnels et à balayage

Lemme 4.3.1. *$\overline{2NPEP_1}$ est PSPACE-complet ($\overline{u2NPEP_1}$ est NP-complet) [15, p. 265]. Du même coup, $2NPEP_1$ est PSPACE-complet ($u2NPEP_1$ est coNP-complet).*

Le lemme suivant montre qu'il est peu probable que le problème SPEP_k soit résoluble efficacement.

Lemme 4.3.2. *(u)PEP $\leq_P$ (u)SPEP_1.*

Démonstration. Étant donné k DFAs $\langle A_1, \ldots, A_k \rangle$, nous construisons un SDFA A tel que $L(A) = L(A_1) \cap \cdots \cap L(A_k)$.

Nous convertissons chaque DFA A_i en un SDFA A'_i qui fonctionne exactement de la même façon en déplaçant sa tête de lecture vers la droite. L'automate A exécute l'automate A_i, puis si celui-ci accepte, la tête de lecture est ramenée au début de l'entrée pour l'exécution de l'automate A_{i+1}.

Plus formellement, nous définissons δ de la façon suivante :

$$\begin{array}{lcll} \delta(q_i, \sigma \in \Sigma) & = & (\delta_i(q_i, \sigma), R) & (\forall i \in [k], \forall q_i \in Q_i) \\ \delta(f_i, \dashv) & = & (\text{rewind}_i, L) & (\forall i \in [k], \forall f_i \in F_i) \\ \delta(\text{rewind}_i, \sigma \in \Sigma) & = & (\text{rewind}_i, L) & (\forall i \in [k]) \\ \delta(\text{rewind}_i, \vdash) & = & (s_{i+1}, L) & (\forall i \in [k-1]) \end{array}$$

et $A = \langle Q_1 \cup \cdots \cup Q_k, s_1, F_k, \Sigma, \delta \rangle$. L'automate A se construit en temps $O(n^c)$ puisqu'il suffit de parcourir l'encodage d'un automate à la fois puis de le copier sur le ruban de sortie en apportant les quelques modifications. De plus, A est un automate à balayage puisque, par définition de δ, la tête de lecture ne change de direction qu'aux extrémités. Il est simple de vérifier que $L(A) = L(A_1) \cap \cdots \cap L(A_k)$ via la définition de δ.

Si les automates $A_1, \ldots, A_k$ sont unaires alors A l'est aussi. □

Corollaire 4.3.3. *SPEP_k, 2PEP_k SNPEP_k, 2NPEP_k sont PSPACE-complets.*

Lemme 4.3.4. *(uN)2PEP $\leq_P$ (uN)2PEP_1 et (uN)SPEP $\leq_P$ (uN)SPEP_1.*

Démonstration. Nous donnons une construction similiaire à la preuve du lemme 4.3.2. Étant donné k automates $\langle A_1, \ldots, A_k \rangle$, nous construisons un automate du même type tel que $L(A) = L(A_1) \cap \cdots \cap L(A_k)$.

Plus formellement, nous construisons $A = \langle Q_1 \cup \cdots \cup Q_k, s_1, F_k, \Sigma, \delta \rangle$ où δ est :

$$\begin{array}{lcll}
\delta(q_i, \sigma \in \Sigma \cup \{\vdash\}) & = & \delta_i(q_i, \sigma) & (\forall i \in [k], \forall q_i \in Q_i) \\
\delta(q_i, \dashv) & = & \delta_i(q_i, \dashv) & (\forall i \in [k], \forall q_i \in Q_i \setminus F_i) \\
\delta(f_i, \dashv) & = & (\text{rewind}_i, L) & (\forall i \in [k], \forall f_i \in F_i) \\
\delta(\text{rewind}_i, \sigma \in \Sigma) & = & (\text{rewind}_i, L) & (\forall i \in [k]) \\
\delta(\text{rewind}_i, \vdash) & = & (s_{i+1}, L) & (\forall i \in [k-1])
\end{array}$$

Si les automates sont unaires (à balayage), alors A l'est aussi. L'automate A se construit en temps polynômial et $L(A) = L(A_1) \cap \cdots \cap L(A_k)$. Nous laissons au lecteur le soin de compléter la preuve. □

Lemme 4.3.5. *SPEP, 2PEP SNPEP, 2NPEP sont PSPACE-complets.*

5 Sommaire

Nous donnons un tableau indiquant la complexité de chacun des problèmes selon le modèle d'automate.

Modèle	PEP_k	PEP
uDFA à un seul état final	L-complet	P
uDFA	L-complet	coNP
DFA	NL-complet	PSPACE-complet
uNFA	NL-complet	PSPACE-complet
NFA	NL-complet	PSPACE-complet
uSDFA	coNP (uPEP $\leq_P \cdot$)	coNP (uPEP $\leq_P \cdot$)
u2DFA	coNP (uPEP $\leq_P \cdot$)	coNP (uPEP $\leq_P \cdot$)
SDFA	PSPACE-complet	PSPACE-complet
2DFA	PSPACE-complet	PSPACE-complet
uSNFA	coNP (uPEP $\leq_P \cdot$)	coNP (uPEP $\leq_P \cdot$)
u2NFA	coNP-complet	coNP-complet
SNFA	PSPACE-complet	PSPACE-complet
2NFA	PSPACE-complet	PSPACE-complet

Nous remarquons que si uPEP est coNP-complet alors uSPEP, u2PEP et uSNPEP le sont aussi. Nous n'écartons toutefois pas la possibilité que uPEP soit dans P.

6 Conséquences d'un algorithme efficace

Nous rappelons que le problème PEP_k est résoluble en temps $O(n^k)$. Soit l'hypothèse suivante :

Hypothèse 6.0.6. *Étant donné $w = \langle A_1, \ldots, A_k \rangle$ où chaque automate est de taille σ, il existe un algorithme qui décide si $w \in PEP_k$ en temps*

$$\sigma^{(k/f(k))+d}$$

où $d > 0$ est une constante et f est une fonction non bornée qui dépend seulement de k. [1]

Nous présentons les conséquences dans le cas où l'hypothèse précédente était vraie. Les résultats suivants proviennent de [1] et nous y référons le lecteur pour obtenir les preuves.

Définition 6.0.7. SUBSET-SUM :

Données :	$S = \{x_1, \ldots, x_m\}$ et t où x_i, t sont des entiers
Problème :	Déterminer s'il existe $\{y_1, \ldots, y_l\} \subseteq S$ tel que $\sum_{i=1}^{l} y_i = t$

Théorème 6.0.8. *Si l'hypothèse 6.0.6 est vraie alors il existe un algorithme qui permet de résoudre le problème SUBSET-SUM en temps $O(2^{\epsilon n})$ pour tout $\epsilon > 0$. [1]*

Tel qu'indiqué dans un commentaire de Dániel Marx [2], il est possible de généraliser ce résultat en utilisant des résultats de la complexité parametrée (voir [18] pour un survol du domaine). Marx construit une réduction parametrée linéaire du problème k-CLIQUE [18] vers PEP$_k$. En utilisant un résultat de [19], cela démontre que l'existence d'un algorithme s'exécutant en temps $f(k)n^{o(k)}$ pour PEP$_k$ implique l'existence d'un algorithme s'exécutant en temps $2^{o(n)}$ pour 3SAT (ce qui n'est pas connu). Ici, la fonction f est une fonction calculable quelconque ; un temps de $2^{2^k} n^{o(k)}$ serait donc suffisant pour démontrer ce résultat.

Définition 6.0.9. FACTORING :

Données :	Un entier z
Problème :	Déterminer s'il existe deux entiers x, y tels que $z = xy$.

Théorème 6.0.10. *Si l'hypothèse 6.0.6 est vraie alors il existe un algorithme qui permet de résoudre le problème FACTORING en temps $O(2^{\epsilon n})$ pour tout $\epsilon > 0$. [1]*

Théorème 6.0.11. *Si l'hypothèse 6.0.6 est vraie alors*

$$NTIME(t) \subseteq DTIME(2^{\epsilon t})$$

pour tout $\epsilon > 0$. [1]

L'hypothèse suivante, qui est très peu différente de l'hypothèse 6.0.6, pourrait conduire à la résolution d'un problème ouvert important.

Hypothèse 6.0.12. *Étant donné $w = \langle A_1, \ldots, A_k, B\rangle$ où $\forall i \in [k], A_i$ est de taille σ et B est de traille σ', il existe un algorithme qui décide si*

$$\bigcap_{i=1}^{k} L(A_i) \cap B = \emptyset$$

en temps

$$\sigma^{(k/f(k))+d}\sigma'$$

où $d > 0$ est une constante et f est une fonction non bornée qui dépend seulement de k. [1]

Théorème 6.0.13. *Si l'hypothèse 6.0.12 est vraie alors NL $\neq$ P. [1]*

6.1 Proposition d'une technique de preuve

Nous proposons l'esquisse d'une démarche qui pourrait permettre de montrer que les hypothèses faites dans [1] sont fausses.

Comme remarqué dans la preuve du théorème 4.2.4, le problème PEP_{k+1} semble nécessiter $\log n$ bits de plus que le problème PEP_k. Il pourrait donc être possible de définir une hiérarchie dans NL où NL_k serait l'ensemble des langages résolubles par une machine de Turing non déterministe sur alphabet Σ (pour éviter une accélération artificielle via l'alphabet) nécessitant $k \log n + c$ bits de mémoire. Si PEP_k était complet pour NL_k selon une réductibilité assez fine (ex : réductibilité $(\log n + c)$-espace) et qu'il existait un problème dans NL_k nécessitant un temps $\Omega(n^k)$ alors PEP_k nécessiterait un temps $\Omega(n^k)$. Une réduction alternative pourrait être une réduction utilisant $k \log n$ bits de calcul mais seulement $\log n$ bits de sortie.

Il n'est pas clair que cette méthode soit prometteuse. D'abord, il est difficile de définir une hiérarchie aussi fine puisque plusieurs détails sont nécessaire pour une analyse pointilleuse de l'espace utilisée. De plus, il n'est pas clair que PEP_k soit complet pour NL_k. L'hypothèse selon laquelle il existe un problème dans NL_k nécessitant un temps $\Omega(n^k)$ semble plausible, mais difficile à prouver comme la majorité des bornes inférieures.

En revanche, si cette méhode s'avérait réalisable, le problème serait réduit à trouver un problème arbitraire dans NL_k nécessitant un temps $\Omega(n^k)$, ce qui pourrait être potentiellement plus simple à résoudre.

7 Conclusion

La complexité du problème d'intersection d'automates a été établie pour plusieurs modèles d'automates. La plupart des variantes sont complètes pour une classe de complexité, le problème est donc un candidat intéressant pour séparer certaines classes. Nous avons aussi observé que la production d'un algorithme efficace pour PEP_k aurait des conséquences inespérées, à savoir une preuve que $\text{NL} \neq \text{P}$. Finalement, nous avons esquissé une approche qui pourrait aider à montrer qu'un tel algorithme n'existe pas.

La complexité exacte du problème uPEP demeure une question ouverte. Nous avons démontré que le problème est dans la classe coNP et qu'une restriction du problème est dans P. Il serait donc intéressant de montrer que uPEP est coNP-complet ou bien dans la classe P.

Une autre question soulevée par ce travail concerne les propriétés des automates rendant PEP PSPACE-ardu. Par exemple, existe-t-il une classification intéressante de la complexité de PEP en fonction des propriétés algébriques des automates utilisés ? Quelles sont les restrictions suffisantes (ou nécessaires) pour rendre le problème tractable ?

Références

[1] George Karakostas, Richard J. Lipton, Anastasios Viglas. *On the complexity of intersecting finite state automata and NL versus NP*, Theoretical Computer Science Volume 302, Issues 1-3, 2003.

[2] Richard Lipton. *On The Intersection of Finite Automata*, blog personnel, 2009.

[3] John E. Hopcroft, Rajeev Motwani, Jeffrey D. Ullman. *Introduction to Automata Theory, Languages, and Computation*, Addison-Wesley ; 2nd edition, 521 pages, 2000.

[4] Michael Sipser. *Introduction to the Theory Of Computation*, Course Technology ; 2nd edition, 456 pages, 2005.

[5] Michael Sipser, William J. Sakoda. *Nondeterminism and the Size of Two Way Finite Automata*, Proceedings of the tenth annual ACM symposium on Theory of computing, 1978.

[6] Viliam Geffert, Carlo Mereghetti, Giovanni Pighizzini. *Converting two-way nondeterministic unary automata into simpler automata*, Mathematical foundations of computer science, 2003.

[7] Michael Sipser. *Lower Bounds on the Size of Sweeping Automata*, Journal of Computer and System Sciences, Volume 21, Issue 2, 1980.

[8] Gilles BRASSARD, Paul BRATLEY. *Fundamentals of Algorithmics*, Prentice Hall, 524 pages, 1996.

[9] Moshe Y. VARDI. *A Note on the Reduction of Two-Way Automata to One-Way Automata*, Information Processing Letters, 1989.

[10] Giovanni PIGHIZZINI, Jeffrey SHALLIT. *Unary Language Operations, State Complexity and Jacobsthal's Function*, International Journal of Foundations of Computer Science, 2002.

[11] Dexter KOZEN. *Lower Bounds for Natural Proof Systems*, Proceedings of the 18th Annual Symposium on Foundations of Computer Science, 1977.

[12] Neil IMMERMAN. *Nondeterministic Space is Closed Under Complementation*, SIAM Journal of Computing, 1998.

[13] Christos H. PAPADIMITRIOU. *Computational Complexity*, Addison-Wesley, 523 pages, 1993.

[14] Pierre MCKENZIE, Stephen A. COOK. *Problems Complete for Deterministic Logarithmic Space*, Journal of Algorithms 8, 1987.

[15] Michael GAREY, David S. JOHNSON. *Computers and Intractability : A Guide to the Theory of NP-Completeness*, W.H. Freeman and Company, 340 pages, 1979.

[16] Donald E. KNUTH. *The Art of Computer Programming, Volume 2 : Seminumerical Algorithms*, Addison-Wesley Professional ; 3rd edition, 784 pages, 1997.

[17] Pierre MCKENZIE, Stephen A. COOK. *The Parallel Complexity of Abelian Permutation Group Problems*, SIAM J. Comput. vol. 16, 1987.

[18] Jörg FLUM, Martin GROHE. *Parameterized complexity theory*, Springer, 493 pages, 2006.

[19] Jianer CHEN, Xiuzhen HUANG, Iyad A. KANJ, Ge XIA. *Linear FPT reductions and computational lower bounds*, Proceedings of the thirty-sixth annual ACM symposium on Theory of computing, 2004.