Quelques contributions en logique mathématique et en théorie des automates

Mohamed Dahmoune

▶ To cite this version:

Mohamed Dahmoune. Quelques contributions en logique mathématique et en théorie des automates. Mathématiques générales [math.GM]. Université Paris-Est, 2014. Français. NNT : 2014PEST1013 . tel-01328133

HAL Id: tel-01328133
https://theses.hal.science/tel-01328133v1

Submitted on 7 Jun 2016

HAL is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L'archive ouverte pluridisciplinaire **HAL**, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d'enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.

UNIVERSITE PARIS-EST

École doctorale MSTIC

Thèse de doctorat

Pour obtenir le titre de

Docteur de l'Université Paris-Est

Spécialité : INFORMATIQUE

Quelques contributions en logique mathématique et en théorie des automates

---

défendue par

MOHAMED DAHMOUNE

Directeur de thèse : PATRICK CÉGIELSKI et ALEXIS BÈS

préparée au LACL

Soutenue le 23 juin 2014 devant le jury composé de :

Directeur :	PATRICK CÉGIELSKI	Université Paris-Est - LACL
Encadrant :	ALEXIS BÈS	Université Paris-Est - LACL
Rapporteurs :	PASCAL CARON	Université de Rouen - LITIS
	JEAN-MARC TALBOT	Université Aix-Marseille - LIF
Examinateurs :	CHRISTIAN CHOFFRUT	Université Denis Diderot - LIAFA
	PIERRE VALARCHER	Université Paris-Est - LACL

TABLE DES MATIÈRES

1 Introduction **1**

1.1 Présentation du contexte et des résultats 1

1.1.1 Logiques des mots sur un alphabet infini 1

1.1.2 Minimisation du nombre de transitions 6

1.1.3 Recherche approchée de motif 7

1.1.4 Outil graphique de manipulation des automates finis 10

1.2 Définitions et notations 11

1.2.1 Théorie des automates 11

1.2.1.1 Mots et langages 11

1.2.1.2 Expressions rationnelles 12

1.2.1.3 Automates 13

1.2.1.3.1 Automate complet 15

1.2.1.3.2 Élimination des ε-transitions 15

1.2.1.3.3 Déterminisme 16

1.2.1.3.4 Automate homogène 16

1.2.1.3.5 Automate quotient 17

1.2.1.3.6 Automate minimal 17

1.2.2 Logique 18

1.2.2.1 Formalismes logiques 18

1.2.2.2 Structures automatiques 19

1.2.2.3 Interprétation logique du premier ordre 19

1.2.2.4 Interprétation logiques du second ordre 20

1.2.2.5 Structures universelles 20

1.2.2.6 $\mathfrak{M}$-automates 21

I Partie théorique **23**

2 Automaticité et décidabilité pour des logiques des mots sur un alphabet infini **25**

2.1 Structure de clone 27

2.1.1 Théorie du premier ordre de la structure de clone 27

2.1.2 Théorie monadique du second ordre de la structure de clone 29
2.2 Structure étendue du clone 30
2.2.1 Théorie du premier ordre de la structure étendue du clone 30
2.3 Structure de *differentia* 33
2.3.1 Théorie du premier ordre de la structure de *differentia* 33
2.3.2 Théorie monadique du second ordre de la structure de *differentia* . . . 37
2.4 Structure des différences 38
2.5 Structure d'applications exclusives 42
2.6 Structure de décomposition 45
2.7 Autres résultats 49
2.8 Conclusion 51

3 Réduction du nombre de transitions pour les automates finis triangulaires 53
3.1 CFS pour les automates homogènes 54
3.2 Automates triangulaires $\mathcal{A}_n$ 56
3.2.1 Nombre d'états dans l'automate réduit de $\mathcal{A}_n$ 58
3.2.2 CFS pour les automates $\mathcal{A}_n$ 58
3.2.3 Réduction polynomiale pour les automates $\mathcal{A}_n$ 66
3.2.4 Réduction quasi-linéaire pour les automates $\mathcal{A}_n$ 71
3.2.4.1 Les P-arbres 73
3.2.4.2 Triangle d'Ératosthène - Pascal 74
3.2.4.3 Les P-arbres et le triangle d'Ératosthène - Pascal 75
3.2.5 Minimalité asymptotique 77
3.2.6 Résultats expérimentaux 82
3.3 Les arbres premiers 84
3.4 Conclusion 85

II Partie pratique 87

4 Automates à trous pour les dictionnaires de mots 89
4.1 Notations et définitions 89
4.2 Solution naïve du problème de la recherche approchée 90
4.3 Automate à trous 91
4.4 Résultats expérimentaux 94
4.5 Conclusion 103

5 Outil graphique *womb* de manipulation des automates finis 111
5.1 Mode d'emploi 111
5.1.1 Menu Fichier 112
5.1.2 Menu Action 112
5.1.3 Menu Affichage 113
5.2 Implémentation 114
5.2.1 Affichage 115
5.2.2 Élimination des transitions spontanées 118
5.2.3 Complétion 120
5.2.4 Déterminisation 122

5.2.5 Complémentation . . . 124
5.2.6 Automate transposé . . . 124
5.2.7 Minimisation . . . 125
5.3 Conclusion . . . 127

Conclusion **129**

Bibliographie **130**

Index **137**

Abréviations **143**

Résumé **144**

LISTE DES TABLEAUX

2.1 Principaux résultats 25

3.1 Tableau de comparaison 83

4.1 Dictionnaires pseudo-aléatoires 94
4.2 Dictionnaires de mots naturels de plusieurs langues. 94
4.3 Complexités théoriques et estimations pratiques expérimentales. 110

5.1 Principaux résultats 129

TABLE DES FIGURES

1.1 Représentation graphique d'un automate 14
1.2 Un automate avec des ε transitions 15
1.3 Une exemple d'un $\mathfrak{M}$-automate. 22

2.1 L'automate $\mathcal{A}_{d(\Sigma^*)}$. 30
2.2 L'automate $\mathcal{A}_{d(\prec)}$. 31
2.3 L'automate $\mathcal{A}_{d(\sim)}$. 31
2.4 L'automate $\mathcal{A}_{d(\oplus)}$. 31
2.5 L'automate $\mathcal{A}_{d(\mathfrak{clone})}$. 32
2.6 L'automate $\mathcal{A}_{d(\mathfrak{less})}$. 32
2.7 L'automate $\mathcal{A}_{d(\mathfrak{mod}_{p,q})}$. 32
2.8 L'automate $\mathcal{A}_{\mu(\mathfrak{clone})}$. 41
2.9 L'automate $\mathcal{A}_{\mu(\mathfrak{diff}_3)}$. 41
2.10 L'automate $\mathcal{A}_{\mu(\mathfrak{diff}_4)}$. 41
2.11 L'automate $\mathcal{A}_{\mu(\mathfrak{diff}_i)}$. 42
2.12 L'automate $\mathcal{A}_{\lambda(\Sigma^*)}$. 44
2.13 L'automate $\mathcal{A}_{\lambda(\prec)}$. 44
2.14 L'automate $\mathcal{A}_{\lambda(\sim)}$. 44
2.15 L'automate $\mathcal{A}_{\lambda(R^{\oplus}_{f_j})}$. 45

3.1 Exemples de décompositions 54
3.2 Exemple d'un automate CFS 56
3.3 Un deuxième exemple d'un automate CFS 56
3.4 L'automate triangulaire de l'ordre 3 57
3.5 L'automate triangulaire de l'ordre 4 57
3.6 Table de transition de l'automate triangulaire 57
3.7 L'automate ayant un seul état final équivalent à $\mathcal{A}_2$ 59
3.8 L'automate ayant un seul état final équivalent à $\mathcal{A}_3$ 59
3.9 Décomposition en trois éléments. 60
3.10 Un exemple de décomposition CFSZ 61
3.11 Un 5-arbre binaire complet et un système de partition $Z(\mathcal{A}_3)$ associé 65

3.12 La fonction Split . . . 72
3.13 La fonction SplitAll . . . 74
3.14 P-arbres . . . 74
3.15 Les chemins de coût égal à 5 . . . 76
3.16 Résultats expérimentaux. . . . 84
3.17 Automate réduit . . . 86

4.1 Automate à trous. . . . 93
4.2 Taille du voisinage par rapport à la taille du dictionnaire. . . . 95
4.3 Nombre d'états et nombre de transitions de l'automate à trous. . . . 96
4.4 Rapport de taille selon k . . . 97
4.5 Rapport de taille selon $|D|$ . . . 98
4.6 Rapport de taille pour plusieurs dictionnaires de langues . . . 99
4.7 Rapport de taille entre l'automate à trous binaire et le voisinage selon k . . . 100
4.8 Rapport de taille entre l'automate à trous binaire et le voisinage selon $|D|$ . . . 101
4.9 Rapport de taille pour plusieurs dictionnaires de langues . . . 102
4.10 Nombre d'états de l'automate à trous binaire pour plusieurs valeurs de k . . . 103
4.11 Nombre d'états de l'automate à trous binaire pour plusieurs valeurs de $|D|$ . . . 104
4.12 Nombre d'états de l'automate à trous binaire selon L . . . 105
4.13 Taille de l'automate à trous binaire pour $k = 1$ . . . 106
4.14 Taille de l'automate à trous binaire pour $k = 2$ . . . 107
4.15 Taille de l'automate à trous binaire pour $k = 3$ . . . 108
4.16 Taille de l'automate à trous binaire pour $k = 4$ . . . 109

5.1 Menu principal. . . . 112
5.2 Menu Fichier . . . 112
5.3 Menu Fichier . . . 113
5.4 Menu Affichage . . . 113
5.5 Exemple d'un automate fini. . . . 115
5.6 Affichage spiral et circulaire. . . . 118
5.7 Le synchrone de $\mathcal{A}$. . . . 119
5.8 Le déterminisé de $\mathcal{A}$. . . . 122
5.9 Le transposé de $\mathcal{A}$. . . . 124
5.10 Le minimal de $\mathcal{A}$. . . . 127

LISTE DES ALGORITHMES

1 Division 35

2 Algorithme de décomposition itératif 62
3 Algorithme de décomposition récursif 62
4 Algorithme de décomposition dynamique 66
5 Algorithme de génération des Z-arbres 73
6 Successeur(t_p) 85

LISTINGS

3.1 Code en C++ de l'algorithme dynamique . . . 71
5.1 Types et déclarations (code en langage C++) . . . 114
5.2 Vérification de l'existence d'un état initial . . . 115
5.3 Dessin des transitions . . . 116
5.4 Dessin de l'automate . . . 117
5.5 Réorganisation des états selon la méthode circulaire . . . 117
5.6 Réorganisation des états selon la méthode spirale . . . 118
5.7 Élimination des ε-transitions . . . 120
5.8 Complétion d'un automate . . . 121
5.9 Déterminisation d'un automate . . . 123
5.10 Complémentation d'un automate . . . 124
5.11 Automate transposé . . . 125
5.12 Minimisation de Brzozowski . . . 125
5.13 Minimisation de Moore . . . 126

CHAPITRE 1

INTRODUCTION

Ce manuscrit détaille un ensemble de contributions théoriques et quelques études expérimentales portant essentiellement sur la théorie des automates et la logique mathématique. La première partie est divisée en deux : un premier chapitre montre quelques résultats de décidabilité de certaines théories logiques et l'automaticité de plusieurs structures logiques sur des mots écrits sur un alphabet infini. Un deuxième chapitre présente quelques résultats obtenus pour le problème de la réduction/minimisation du nombre de transitions dans un automate fini. La deuxième partie du manuscrit contient deux chapitres courts : un premier chapitre introduit les automates à trous et leurs application à la recherche approchée de motif dans les dictionnaires de mots. Le deuxième chapitre présente un outil graphique des automates finis implémentant les opérations les plus basiques de manipulation des automates.

1.1 Présentation du contexte et des résultats

Cette section introduit le cadre théorique et pratique motivant les questions discutées dans ce travail. Une sous-section résumant brièvement le contexte du problème traité par chaque chapitre et les principaux résultats obtenus.

1.1.1 Logiques des mots sur un alphabet infini

Depuis le début des années cinquante, de nombreux travaux réalisés par des mathématiciens, des logiciens et des informaticiens ont montré des liens entre les questions de définissabilité et décidabilité en logique et la théorie des automates.

Parmi les premiers résultats majeurs on a l'équivalence entre la définissabilité au second ordre monadique faible et la reconnaissabilité par automate [Büc60].

Ce résultat a été ensuite étendu notamment aux mots infinis et aux arbres, ce qui a permis le développement des notions d'automates associés.

Les travaux sur le problème de décidabilité ont permis entre autres de prouver de nouveaux résultats liés à la complexité et aux classes de complexité des problèmes.

Ces résultats ont permis non seulement d'avoir un cadre formel abstrait pour manipuler les théories logiques mais aussi de fournir des outils qui ont un très grand intérêt pour l'informatique comme la conception correcte des programmes grâce à la spécification, la modélisation, la vérification formelle et la démonstration automatique des systèmes informatiques. On cite par exemple, la vérification des programmes, les outils de la logique temporelle comme CTL et LTL et les outils de spécification et vérification formelles comme MONA, qui s'appuient sur la théorie des automates.

D'autre part, l'étude de la théorie des structures des mots finis sur un alphabet infini est considérée essentiellement dans :

1. les systèmes à états infinis,
2. les systèmes paramétrés lorsque l'éventail des valeurs des paramètres n'est pas borné,
3. les systèmes temporisés,
4. la théorie des bases de données [BLSS03, BMS+06],
5. la vérification (*model checking*) [DLS08].

Ainsi que pour de nombreuses autres applications typiques dans les systèmes de contrôle avec une source de données infinie, par exemple :

1. les systèmes à paramètres entiers [BHM03],
2. les systèmes de journalisation qui stockent des données appartenant à un domaine infini [Via09, BHJS07],
3. les données semi-structurées comme les documents XML qui peuvent être considérées comme des arbres pour lesquels les feuilles et les branches sont généralement associées à des valeurs appartenant à un domaine infini [CFB+02, BCC+03].

Notons que les solutions développées pour l'analyse de ces systèmes sont en général également applicables aux systèmes finis avec un très grand nombre d'états.

Exemple 1 (Système temporisé, *Timed system*)**.** Soit un ensemble fini d'états S. Un système temporisé passe par un nombre fini d'états à des instants bien précis $t_i \in \mathbb{R}^+$ ou $t_i \in \mathbb{N}$ (temps continu ou discret). Un comportement fini possible d'un tel système peut se modéliser comme une liste de couples ordonnée dans le temps, où chaque couple (état, instant) donne l'état du système à un instant précis. Ainsi le couple (s_i, t_i) veut dire que le système est dans l'état s_i à l'instant t_i. Par conséquent, et comme les t_i font partie d'un ensemble infini, un comportement fini possible dans un système temporisé peut être considéré comme un mot fini sur un alphabet infini : $(s_0, t_0).(s_1, t_1)\dots(s_n, t_n) \in (S \times \mathbb{R}^+)$ ou $(S \times \mathbb{N})$.

Exemple 2 (Gestion de processus, *Process management*)**.** Étant donné un ensemble fini d'actions A (créer, supprimer, arrêter ou suspendre un processus), l'historique d'un système de gestion de processus n'est rien d'autre qu'une liste de couples (action, processus) qui énumère la succession des actions exécutées sur l'ensemble des processus $p_i \in \mathbb{N}$ (p_i est l'identifiant d'un processus), ainsi le couple (a_i, p_i) veut dire qu'on a effectué l'action a_i sur le

processus p_i. Par conséquent, et comme les identifiants p_i font partie d'un ensemble infini, *a priori* non borné, un historique fini possible dans un système de gestion de processus peut être vu comme un mot fini sur un alphabet infini : $(a_0, p_0).(a_1, p_1)\ldots(a_n, p_n) \in (A \times \mathbb{N})$.

Il est naturel de s'intéresser à la spécification de tels systèmes, à la modélisation de leur comportement, ou encore à la vérification de leurs propriétés. Une approche naturelle consiste à essayer d'étendre les notions et les concepts de modèles de calcul et des formalismes logiques qui sont connus dans le cas d'un alphabet fini aux alphabets infinis, tout en conservant les bonnes propriétés de fermeture algébrique, clôture logique, ainsi que les propriétés d'expressivité et de décidabilité.

Une approche classique consiste à utiliser les automates finis. L'idée principale est de coder les comportements possibles du système par des mots sur un alphabet donné. Ainsi l'ensemble des comportements qui vérifient une certaine propriété du système peut être codé par un langage de mots sur l'alphabet donné. Si ce langage est rationnel alors il peut être accepté par un automate finis.

Lorsque l'ensemble des états possibles d'un système est fini, la spécification et l'étude des comportements finis de ce système peuvent être modélisés par des automates finis. D'autre part, l'analyse des comportements infinis peut être effectuée par des automates finis spécifiques comme les automates de Büchi, de Muller ou de Rabin. Tandis que lorsque le domaine du système est infini, sa vérification est en général incertaine. Par conséquent, la vérification automatique des systèmes à domaine infini et l'étude de leurs comportements, même finis, nécessite une extension des automates finis. Plusieurs formalismes ont été proposés. Mentionnons, par exemple, les automates à registres et les *pebble automata* [SF94, KF94, NSV01, Tan10], les $\mathfrak{M}$-automates [Bès08], les *data automata* [BDM$^+$11, BMSS09] et les *variable automata* [GKS10]. Ces méthodes formelles basées sur les automates, ainsi que d'autres, sont appliquées avec succès pour la vérification de plusieurs systèmes automatisés, mais ils présentent certaines limites :

1. Automates à registres [SF94, KF94, NSV01] :
   (a) le problème de l'universalité est indécidable,
   (b) le problème de l'appartenance est indécidable,
   (c) ces automates ne sont pas clos par complémentation mis à part pour certains fragments spécifiques qui limitent le nombre de registres [DLN07],
   (d) le problème du langage non vide est décidable.
2. *Pebble automata* [Tan10] :
   (a) le problème du langage non vide est indécidable,
   (b) le problème de l'universalité est indécidable,
   (c) le problème de l'appartenance est indécidable.
3. *Data automata* [BDM$^+$11, BMSS09] :
   (a) le problème de l'universalité est indécidable,
   (b) le problème de l'appartenance est indécidable,
   (c) ces automates ne sont pas clos par complémentation,
   (d) le problème du langage non vide est décidable.

4. *Variable automata* [GKS10] :
   (a) le problème de l'universalité est indécidable,
   (b) le problème de l'appartenance est indécidable,
   (c) ces automates ne sont pas clos par les opérations d'union et d'intersection.
5. Les versions déterministe et non déterministe ne sont pas équivalentes pour la plupart de ces modèles d'automates.
6. Certains de ces automates ne peuvent pas être utilisés pour reconnaître des propriétés basiques, comme la propriété qu'un mot x se termine par deux lettres identiques ($\mathfrak{clone}(x)$), ou la propriété que toutes les lettres du mot x sont distinctes ($\mathfrak{diff}(x)$).

Une autre approche consiste à se concentrer sur des formalismes logiques particuliers, telles que la logique du premier ordre et la logique monadique du second ordre. L'idée est de représenter le domaine et de traduire les propriétés de ces systèmes par des formules exprimées dans la théorie logique d'une structure spécifique. Cette approche a l'avantage d'avoir des propriétés de fermeture pour plusieurs opérations comme le produit cartésien, la projection et tous les opérateurs booléens. Toutefois, afin d'obtenir des résultats de décidabilité, nous devons considérer de fortes contraintes sur l'expressivité de ces théories. Nous mentionnons dans ce qui suit certains de ces formalismes logiques :

1. Eilenberg *et al.*, ont étudié dans [EES69] des structures avec comme domaine des mots sur un alphabet infini dénombrable Σ, avec les prédicats suivants :
   (a) la relation $\sim$ définissant l'ensemble de paires de mots de même longueur,
   (b) la relation unaire $\mathfrak{last}_a$ qui définit un mot qui se termine avec la lettre a (un prédicat pour chaque lettre de l'alphabet).

   Ils caractérisent les relations définissables et prouvent la décidabilité de la théorie du premier ordre de ces structures.
2. Choffrut & Grigorieff [CG09a, CG09b] ont étudié plusieurs variantes des structures d'Eilenberg, avec les fonctions et les prédicats suivants :
   (a) $\mathfrak{pred}(a_1 \dots a_n) = a_1 \dots a_{n-1}$ pour $a_1, \dots, a_n \in \Sigma$,
   (b) $\mathfrak{eqLast}$ est la relation binaire $\{(ua, va) \mid u, v \in \Sigma^*, a \in \Sigma\}$, c'est-à-dire l'ensemble des couples de mots qui se terminent par la même lettre,
   (c) $\mathfrak{eqLenEqLast}$ est une relation binaire définissant l'ensemble des paires de mots de même longueur et qui se terminent par la même lettre.

   Ils montrent en particulier que pour un alphabet infini Σ, le fragment $\exists\forall\forall$ de la théorie de la structure $(\Sigma^*; \prec, \varepsilon, \mathfrak{pred}, \mathfrak{eqLast})$ est indécidable. Ils montrent aussi que la théorie du premier ordre de la structure $(\Sigma^*; \prec, \sim, \mathfrak{eqLenEqLast}, (\mathfrak{last}_a)_{a\in\Sigma})$ est décidable.

   Ainsi, nous pouvons voir que l'ajout d'un simple prédicat comme $\mathfrak{eqLast}$ conduit à l'indécidabilité.
3. Dans [Bès08], Bès a montré qu'un cas particulier du théorème de composition de Feferman - Vaught [FV59] donne lieu à une notion naturelle d'automates pour les mots finis sur un alphabet infini. Cette nouvelle notion est appelée $\mathfrak{M}$-*automate* et elle possède de bonne propriétés de fermeture et de décidabilité.

   Remarquons que ce formalisme, comme les deux précédents, ne permet d'exprimer certaines propriétés simples comme celle de se terminer par deux lettres identiques, qui peuvent être utilisés pour définir des langages plus complexes comme $\Sigma^* aa \Sigma^*$ avec $a \in \Sigma$.

4. Shelah [She75] mentionne un résultat de Stupp [Stu75] qui a ensuite été amélioré par Muchnik. Étant donnée une structure initiale, appelée *la structure de base*, nous pouvons construire une nouvelle structure, appelée *la structure itérée*, dont le domaine est constitué de l'ensemble des séquences finies d'éléments dans le domaine de la structure de base avec de nouvelles relations, telles que le prédicat $\mathfrak{clone}$.

   Soient $\mathfrak{L}$ une signature et $\mathfrak{A} = (\Sigma; \dots, R_i, \dots)$ une $\mathfrak{L}$-structure relationnelle, où chaque R_i est une relation d'arité r_i. *La structure itérée* de $\mathfrak{A}$ est la structure :

$$\mathfrak{A}^* = (\Sigma^*; \prec, \mathfrak{clone}, \dots, R_i^*, \dots)$$

   de signature $\mathfrak{L}^* = \mathfrak{L} \cup \{\prec, \mathfrak{clone}\}$ telle que :

   (a) $\prec = \{(w, wu) \mid w, u \in \Sigma^*\}$,

   (b) $\mathfrak{clone} = \{waa \mid w \in \Sigma^*, a \in \Sigma\}$,

   (c) $R_i^* = \{(wa_1, \dots, wa_{r_i}) \mid w \in \Sigma^*, (a_1, \dots, a_{r_i}) \in R_i\}$.

   Muchnik a montré que pour chaque énoncé monadique du second ordre φ on peut effectivement construire un énoncé φ' tel que $\mathfrak{A} \models \varphi'$ si, et seulement si, $\mathfrak{A}^* \models \varphi$ pour toute structure $\mathfrak{A}$. Un corollaire important de ce résultat est que la théorie monadique du second ordre de la structure itérée est décidable si la théorie monadique du second ordre de la structure de base est décidable. La preuve originelle de Muchnik n'a jamais été publiée. Le résultat est mentionné par Semenov dans [Sem84]. La première publication de la preuve est due à Walukiewicz [Wal02]. Berwanger & Blumensath donnent plus de détails dans [BB01]. Récemment, Kuske & Lohrey [KL06, KL03] montrent plus de résultats pour la logique du premier ordre.

Nous nous intéressons dans le chapitre 2 aux formalismes logiques qui nous permettent d'exprimer des relations naturelles telles que les prédicats : $\mathfrak{clone}$, $\sim$, $\prec$ et $\mathfrak{diff}$ sur les mots finis écrits sur un alphabet infini dénombrable. Nous considérons des théories logiques classiques comme la théorie du premier ordre ainsi que la théorie monadique du second ordre. En fait, notre travail se concentre sur l'identification et l'ajout de quelques prédicats intéressants tout en conservant la décidabilité de ces théories.

Nous avons en premier lieu considéré *la structure du clone*

$$\mathfrak{S}_0 = (\Sigma^*; \prec, \mathfrak{clone}, \mathfrak{less}, \{\mathfrak{mod}_{p,q} \mid p, q \in \mathbb{N}\})$$

où Σ^* désigne l'ensemble des mots finis sur l'alphabet infini $\Sigma = \mathbb{Z}$ et pour des mots x, y et z de Σ^*, on a :

1. $x \prec y$ si, et seulement si, x est un préfixe strict de y,
2. $\mathfrak{clone}(x) \Leftrightarrow x = uaa$ avec $u \in \Sigma^*$ et $a \in \Sigma$,
3. $\mathfrak{less}(x) \Leftrightarrow x = uab$ avec $u \in \Sigma^*$, $a, b \in \Sigma$ et $a < b$,
4. pour tout $p, q \in \mathbb{N}$: $\mathfrak{mod}_{p,q}(x) \Leftrightarrow x = uab$ avec $u \in \Sigma^*$, $a, b \in \Sigma$ et $(b - a) \equiv q[p]$.

Nous avons d'abord démontré que la théorie du premier ordre de la structure $\mathfrak{S}_0$ est décidable. Ce résultat est un corollaire de l'automaticité de $\mathfrak{S}_0$. Nous avons ensuite démontré que la théorie monadique du second ordre de $\mathfrak{S}_0$ est également décidable par interprétation dans la théorie S3S décidable d'après Rabin [Rab69].

Nous avons ensuite considéré la structure

$$\mathfrak{S}_1 = (\Sigma^*; \prec, \mathfrak{clone}, \mathfrak{less}, \{\mathfrak{mod}_{p,q} \mid p, q \in \mathbb{N}\}, \sim, \oplus)$$

qui étend la structure $\mathfrak{S}_0$ en ajoutant les prédicats :

1. $x \sim y \Leftrightarrow |x| = |y|$,
2. le prédicat $\oplus(x, y, z)$ est vrai si, et seulement si, x, y et z ont la même longueur et la somme de x et y, lettre par lettre, est égal à z.

Nous avons démontré la $\mathfrak{M}$-automaticité de $\mathfrak{S}_1$, ce qui nous a permis de déduire la décidabilité de sa théorie du premier ordre.

Nous avons par ailleurs étudié des structures qui comportent le prédicat $\mathfrak{diff}$. En particulier nous avons démontré la décidabilité de la théorie du premier ordre et de la théorie monadique du second ordre de la structure $\mathfrak{S}_4 = (\Sigma^*; \prec, \mathfrak{clone}, \mathfrak{diff})$.

Nous avons également obtenu des résultats de décidabilité et des résultats d'indécidabilité pour plusieurs variantes des structures $\mathfrak{S}_0$ et $\mathfrak{S}_4$, ainsi que pour des familles de structures appelées *structure d'applications exclusives* et *structure de décomposition*. On trouvera au début du chapitre 2 un tableau récapitulatif de tous les résultats obtenus (tableau 5.1).

Passons maintenant à la présentation du deuxième chapitre théorique qui concerne la réduction du nombre de transitions dans un automate fini.

1.1.2 Minimisation du nombre de transitions

Dans plusieurs domaines d'application seuls les petits automates permettent d'offrir des solutions réalisables et efficaces aux problèmes pratiques. Ainsi, la réduction des automates finis est importante à la fois pour des questions théoriques et aussi pratiques.

Minimiser le nombre d'états d'un automate fini est un sujet qui a été largement étudié depuis les années 1950, à la fois dans le cas déterministe et le cas non-déterministe [Moo56, Hop71a]. Toutefois, les travaux sur la minimisation du nombre de transitions sont apparus récemment.

En 1997, Hromkovič *et al.* [HSW01] ont donné un algorithme, fondé sur le concept de *Common Follow Sets* (CFS) d'une expression rationnelle, qui convertit une expression rationnelle de taille n en un automate fini non déterministe sans ε-transition ayant $O(n)$ états, $\Omega(n \ \log_2 n)$ transitions comme borne inférieure et $O(n(\log_2 n)^2)$ transitions comme borne supérieure. La borne inférieure du nombre de transitions a été améliorée par Lifshits [Lif03] en $\Omega(\frac{n(\log_2 n)^2}{\log_2 \log_2 n})$. Puis Schnitger [Sch06] l'a améliorée en $\Omega(n(\log_2 n)^2)$. Hagenah et Muscholl [HM00] ont montré que l'algorithme proposé par Hromkovič *et al.* peut être implémenté en temps $O(n(\log_2 n)^2)$. Ouardi et Ziadi [OZ08], en se basant sur la structure ZPC [COZ09], ont donné une construction quasi-linéaire de $O(n(\log_2 n)^2)$ pour convertir une expression rationnelle avec multiplicité de taille n en un automate fini avec multiplicité ayant $O(n)$ états et $O(n(\log_2 n)^2)$ transitions. Geffert[Gef03] a montré que chaque expression rationnelle de taille n sur un alphabet de s symboles peut être convertie en un automate fini non déterministe sans ε-transition avec $O(s \ n \log_2 n)$ transitions.

La plupart des résultats de complexité mentionnés pour le problème de la minimisation du nombre de transitions sont obtenus à partir de l'étude d'une classe spéciale de langages, plus précisément la classe des langages $L(E_n)$. Cette classe de langages correspond à une classe d'automates finis simples qu'on appelle les automates triangulaires. Ainsi, on s'intéresse aux automates triangulaires et à la minimisation de leur nombre de transitions non seulement pour des études de complexité mais aussi pour les raisons suivantes :

1. L'automate triangulaire $\mathcal{A}_n$ contient un nombre quadratique de transitions dont la minimisation n'est pas évidente.
2. L'automate triangulaire $\mathcal{A}_n$ reconnaît exactement l'ensemble de toutes les sous - séquences du mot $1.2.3\dots n$. Minimiser le nombre de transitions de $\mathcal{A}_n$ peut être utile dans le domaine de la recherche de motifs ou encore en bio-informatique [CHL01].
3. La minimisation du nombre de transitions des automates triangulaires peut être utilisée pour concevoir une heuristique pour la réduction du nombre de transitions d'un automate homogène quelconque $\mathcal{A}$, tout en minimisant les fragments triangulaires de cet automate $\mathcal{A}$ (voir section 3.4).

Cox [Cox07] s'est basé sur une recherche exhaustive pour minimiser les automates triangulaires de $n = 1$ jusqu'à $n = 7$. Au-delà, l'espace de recherche est gigantesque et Cox a utilisé une approche heuristique qui construit des automates réduits en nombre de transitions pour $n > 7$. Les automates produits par Cox sont des automates finis non déterministes sans ε-transition ayant $n + 1$ états dont un seul initial.

Nous montrons dans le chapitre 3 comment établir une liaison assez directe entre des systèmes de CFS spécifiques associés aux langages $L(E_n)$ et les arbres binaires complets. Ce lien est prouvé en utilisant un objet combinatoire appelé *triangle d'Ératosthène - Pascal*. Cette correspondance nous permet de transformer la valeur qui nous intéresse (le nombre de transitions) en une valeur assez naturelle associée aux arbres (le poids d'un arbre). En effet, construire un automate ayant un minimum de transitions revient à trouver un arbre de poids minimal.

Nous avons montré, d'une part, que ce nombre de transitions est asymptotiquement équivalent à $n(\log_2 n)^2$ (la borne inférieure). D'autre part, les tests expérimentaux montrent que pour les petites valeurs de n, les automates minimaux en nombre de transitions coïncident (en nombre et en taille) avec ceux obtenus par notre construction. Cela nous mène à suggérer que notre réduction est finalement une minimisation pour les automates triangulaires.

Passons maintenant à l'introduction de la partie pratique. Commençons par présenter le chapitre concernant les automates à trous.

1.1.3 Recherche approchée de motif

Le monde d'aujourd'hui est inondé par une immense quantité d'information et spécialement par de l'information textuelle due à de grandes et croissantes collections de bases de données, articles, livres, etc. Il est essentiel de pouvoir accéder à cette information avec des requêtes très rapides et efficaces. Bien que les données puissent être mémorisées à l'aide de divers supports, le texte demeure la forme principale pour échanger l'information où

les données sont constituées par de gigantesques corpus et dictionnaires de données. Ceci s'applique aussi à l'informatique et à la bio-informatique où une grande quantité de données est enregistrée dans des fichiers linéaires. En outre, la quantité de données disponible dans plusieurs domaines tend à doubler tous les dix-huit mois [Lec00]. L'exploitation de ces textes passe nécessairement par la recherche d'un ou de plusieurs mots dans ces textes. C'est le problème de la *recherche de motif*. Par conséquent, les algorithmes de recherche d'informations doivent être efficaces même si la vitesse et la capacité de mémoire des ordinateurs augmentent régulièrement.

Le problème de la *recherche de motif* consiste à localiser une ou, plus généralement, toutes les occurrences d'un motif dans un ensemble de mots (texte ou dictionnaire). Un motif peut être un mot, un ensemble fini de mots ou un ensemble infini de mots exprimé sous la forme d'une expression rationnelle. Ce problème important apparaît dans plusieurs domaines de l'informatique, notamment dans les traitements de textes, en analyse lexicale et en biologie dans l'analyse de l'ADN et des séquences de protéines. Ainsi, améliorer les techniques utilisées en recherche de motif augmente considérablement le rendement de ces programmes. les techniques utilisées en recherche de motif servent généralement de bases aux autres types de recherche.

Résoudre un problème algorithmique se ramène souvent à trouver une structure de données permettant de résoudre efficacement ce problème. Un exemple typique est celui des moteurs de recherche. Les moteurs de recherche stockent des index sophistiqués qui permettent l'exécution d'une variété de requêtes :
- quels documents contiennent le mot *logique* ?
- quel est le nombre d'occurrences du mot *automate* dans le document `dz.txt` ?

Contrairement à la plupart des problèmes algorithmiques, un temps de réponse linéaire est trop lent, on ne peut pas balayer tout le web pour répondre à une telle question, il faut trouver une solution dont le temps de recherche est logarithmique ou même constant. D'un autre coté, l'espace tend à devenir une question plus importante, parce que le corpus de données est habituellement grand et on ne veut pas vraiment une structure de données qui occupe trop d'espace.

Le problème basique de la *recherche de motif* est la recherche d'un mot w dans un dictionnaire D sans tolérance d'erreurs. Celui-là possède deux variantes :

1. Lorsque le dictionnaire est connu à l'avance, un prétraitement est effectué sur celui-ci, pour construire une structure d'index linéaire en temps et en espace. Ainsi la localisation d'un motif w dans D peut alors s'effectuer en temps et espace linéaire. La structure de données la plus courante pour représenter un index est l'arbre des suffixes [Ukk95, CLS$^+$10, Far97, McC76, Ukk92, Wei73]. Elle permet de stocker un ensemble fini et non vide de mots (un dictionnaire). L'arbre des suffixes est un arbre enraciné dans lequel chaque branche est étiquetée par une lettre. Chaque chemin de la racine vers une feuille est associé à un suffixe d'un mot du dictionnaire.
2. Lorsque le mot est connu à l'avance, un prétraitement est alors effectué sur celui-ci, pour construire une structure de données linéaire en temps et en espace, ainsi, la localisation de w dans D peut alors s'effectuer en temps linéaire. Beaucoup de travaux ont été consacrés à cet effet [KR87, BM77, KJHMP77].

Le problème de la *recherche approchée de motif* est défini comme suit : étant donnés une tolérance d'erreur d'ordre k et un dictionnaire D, concevoir un algorithme ou construire une

structure de données qui peut répondre à la requête suivante : trouver les occurrences d'un motif w dans D tout en tolérant k erreurs. Il y a différentes manières de mesurer l'erreur, comme :

– la distance de Hamming : recherche approchée du mot w dans D avec au plus k substitutions.
– la distance d'édition (distance de Levenstein) : recherche approchée du mot w dans D avec au plus k insertions, substitutions ou suppressions.

Afin d'accélérer la recherche approchée, Cole *et al.* [CGL04] donnent des solutions pour les problèmes des systèmes d'indexation, entre autre le problème de l'indexation approchée. Cela se fait en utilisant une structure de données particulière appelée LCP (*Longest Common Prefix*). Chan *et al.* [CLS$^+$10] ainsi que Cole *et al.* [CGL04] utilisent cette LCP comme noyau pour leur indexation approchée. Ils résolvent le problème de la *recherche approchée de motif* dans un espace de $O(T(\log T)^k)$ et en un temps de recherche de $O(|w| + 2^k \log\log T)$ avec $T = \sum_{v\in D} |v|$.

Karch *et al.* [KLS10] proposent un automate particulier basé sur ce qu'ils appellent le voisinage des délétions pour effectuer des recherches approchées dans des dictionnaires de mots. Ils proposent aussi une implémentation pratique de leur structure de recherche. Les indexes de recherche produits par cet outil est de l'ordre de quelques centaines de mégaoctets même pour une tolérance d'erreur $k < 5$ et pour un dictionnaire de taille relativement moyenne (dictionnaire des mots du français par exemple).

Un exemple typique d'application de la recherche approchée est le correcteur orthographique. Actuellement, la majorité des outils de traitement de texte proposent un correcteur orthographique automatique. L'idée est simple, l'utilisateur tape son texte, et en même temps, le correcteur vérifie si les mots tapés existent dans un dictionnaire bien précis (le dictionnaire d'hébreu par exemple). Un mot est considéré comme faux s'il n'existe pas dans le dictionnaire ; dans ce cas le correcteur orthographique propose à l'utilisateur une liste de mots proches du mot erroné. Vous avez sans doute remarqué que dans plusieurs situations le correcteur orthographique propose une liste qui ne contient pas forcément le mot juste qu'on voulait taper, même parfois pour une différence d'une seule lettre ! Pourquoi ? Tout simplement parce que ces correcteurs utilisent des algorithmes de recherche heuristiques pour engendrer cette liste. Pourquoi des heuristiques ? Car retrouver efficacement la liste complète des mots voisins d'un autre mot par rapport à un dictionnaire n'est pas du tout une tâche facile. On a besoin soit de beaucoup d'espace mémoire pour engendrer cette liste rapidement, soit de beaucoup de temps pour avoir cette liste tout en utilisant peu d'espace mémoire. Les méthodes heuristiques permettent de construire une liste pas forcement complète mais en peu de temps et en consommant peu d'espace.

L'exemple cité du correcteur orthographique , n'est qu'une application possible parmi d'autres. La recherche des voisins d'un mot précis dans un dictionnaire précis trouve son application non seulement en linguistique mais aussi en bio-informatique et en recherche documentaire.

Dans le chapitre 4 on propose une structure de données particulière, appelée *automate à trous binaire* permettant d'engendrer la liste des mots voisins en un temps et un espace expé-

rimentalement linéaires. Théoriquement et dans des cas extrêmes [1], cette structure peut être lourde en recherche ou grande en volume (voir théorèmes 20 et 21). Les indexes de recherche produits par notre algorithme sont de l'ordre de quelques kilo-octets même pour une tolérance d'erreur $k > 4$ et pour un dictionnaire de taille relativement moyenne (dictionnaire des mots du français par exemple).

Les résultats de ce chapitre ont été obtenus suite à une collaboration avec SAÏD ABDEDDAÏM, de l'université de Rouen, France.

Le dernier chapitre du manuscrit décrit *womb*, un outil graphique de manipulation des automates finis, développé dans des buts pédagogiques d'enseignement et aussi pour faciliter les tâches de manipulation et de dessin des automates pour la rédaction de ce manuscrit de thèse.

1.1.4 Outil graphique de manipulation des automates finis

Un graphe, un schéma ou un simple dessin peuvent compacter énormément d'informations dans un espace très réduit. D'autre part, les couleurs, les formes et leurs emplacements reflètent les propriétés des concepts et les liens entre eux. Visualiser un concept facilite considérablement sa compréhension. Ainsi, une bonne conception et une meilleure compréhension d'un automate passent forcément par la fameuse représentation graphique de l'automate, qu'il soit représenté en mémoire machine par des tableaux, des listes ou des matrices.

L'outil *womb* a été développé en C++ et est déjà fonctionnel sur les systèmes Windows [2]. Il est téléchargeable en version binaire depuis le lien :

`https://www.dropbox.com/s/7azngj2dzwp89ay/womb.1.0.0.zip?dl=0`

L'utilitaire *womb* permet, d'une part, de créer facilement avec quelques clics de souris et avec une grande souplesse toutes sortes d'automates finis, d'autre part, cet outil propose quelques fonctionnalités élémentaires de transformation d'automates tels que l'élimination des transitions spontanées (ε-transitions), la déterminisation, la transposition (l'automate miroir), la minimisation et bien sûr l'affichage des automates résultants. Enfin il permet d'exporter les résultats au format *.dot*, qui est un langage de description de graphe dans un format texte. Ce format peut être utilisé notamment pour engendrer du code LaTeX.

Initialement *womb* a été développé pour un but plutôt pédagogique. Durant mes enseignements j'étais amené dans plusieurs cas et dans plusieurs modules à aborder les automates finis, et de montrer à mes étudiants l'intérêt pratique des notions théoriques liées aux automates comme celle de la déterminisation ou de la minimisation. Cet outil m'a permis à la fois de montrer les algorithmes les plus basiques pour la déterminisation et la minimisation des automates et de les appliquer dans des exercices pratiques comme en compilation (mini interpréteur automatique) ou dans le développement de mini protocoles réseau (FTP simplifié ou messagerie instantanée), dans lesquels les automates manipulés ne dépassent pas

1. Pire des cas.
2. Nous envisageons de développer une version JAVA

quelque dizaines d'états. Plus tard j'ai profité de cet outil pour dessiner les automates dont j'avais besoin dans ce manuscrit ou dans mes articles de recherche. Il existe plusieurs outils et bibliothèques pour la manipulation des automates. Par exemple l'outil Vaucanson [3] ou l'utilitaire Semigroupe [4] sont des outils très généraux et efficaces incluant des implémentations optimisées des algorithmes classiques sur les automates. L'outil que nous proposons est caractérisé par une implémentation très basique permettant de mettre en évidence quelques notions fondamentales de la théorie des automates.

1.2 Définitions et notations

Nous présentons dans ce qui suit les objets et les notions de base de la théorie des langages, des automates finis et de la logique mathématique dans lesquels s'inscrit ce travail.

1.2.1 Théorie des automates

Dans ce qui suit on aborde une partie introductive concernant les objets et les notions de base de la théorie des langages et des automates finis. Pour plus de détails, le lecteur se reportera à [BBC92, HMU06, Sak03].

1.2.1.1 Mots et langages

On désigne par *alphabet* un ensemble A non vide, de symboles appelés *lettres*. Dans notre étude on considère des alphabets finis mais également des alphabet infinis. Un *mot* sur l'alphabet A est une suite finie $(a_1, a_2, \ldots, a_n)$ de lettres a_i et sera écrit sous la forme $a_1 a_2 \cdots a_n$. L'ensemble de tous les mots sur l'alphabet A est noté A^*.

La longueur d'un mot u, notée $|u|$, est le nombre de lettre composant le mot. On notera par ailleurs u_i la $i^{\text{ème}}$ lettre du mot u. Il existe un unique mot de longueur égale à zéro : *le mot vide*, noté ε.

Si u et v sont des mots, alors uv est la *concaténation* de u et v. Le mot vide est l'élément neutre de cette opération : $\varepsilon w = w\varepsilon = w$, pour tout mot w.

Définition 1. Soient u et v des mots sur A.

1. u est un *préfixe* de v, s'il existe w de A^* tel que $v = uw$.
2. u est un *suffixe* de v, s'il existe w de A^* tel que $v = wu$.
3. u est un *facteur* de v, s'il existe w et t de A^* tels que $v = tuw$.

Un mot est à la fois préfixe, facteur, et suffixe de lui-même. Le mot vide ε est à la fois préfixe, facteur et suffixe de tout mot.

3. http://www.infres.telecom-paristech.fr/~jsaka/PROJ-VAUC/proj-vauc.html.
4. http://www.liafa.jussieu.fr/~jep/semigroupes.html.

L'opération de concaténation, associative, fait de l'ensemble des mots sur A, noté A^*, un monoïde : *le monoïde libre engendré par* A. On notera A^+ l'ensemble des mots de A^* différents du mot vide. Tout mot de A^* possède une écriture unique comme produit de lettres de A.

Un sous-ensemble L de A^* est appelé *un langage*. Nous notons $\emptyset$ *le langage vide*. Pour un langage L, on note $|L|$ la cardinalité de L.

Étant donnés les langages L_1 et L_2, définissons les opérations suivantes :

1. L'extension de l'opération de concaténation aux langages donne ce qu'on appelle le *produit de langages* : $L_1L_2 = \{w_1w_2 \mid w_1 \in L_1 \text{ et } w_2 \in L_2\}$.
2. *L'union* : $L_1 \cup L_2 = \{w \mid w \in L_1 \text{ ou } w \in L_2\}$.
3. *L'intersection* : $L_1 \cap L_2 = \{w \mid w \in L_1 \text{ et } w \in L_2\}$.
4. *Le résiduel* d'un langage L par rapport à un mot u est défini par : $u^{-1}L = \{v \mid uv \in L\}$.
5. La répétition, un nombre fini de fois, du produit d'un langage par lui-même donne ce qu'on appelle *l'itération bornée* :
   (a) $L^0 = \{\varepsilon\}$,
   (b) $L^n = L^{n-1}L$, pour tout $n > 0$.
6. *L'étoile* (ou fermeture de Kleene) d'un langage L, notée L^*, est l'ensemble : $L^* = \bigcup_{i=0}^{\infty} L^i$.

   De manière similaire, nous définissons L^+ comme : $L^+ = \bigcup_{i=1}^{\infty} L^i$.

Exemple 3. Avec les langages $L_1 = \{aa, bb\}$ et $L_2 = \{a, aa\}$, on a :

$$\begin{aligned} L_1 \cup L_2 &= \{aa, bb, a, aa\}, \\ L_1L_2 &= \{aaa, aaaa, bba, bbaa\}, \\ L_1^* &= \{\varepsilon, aa, bb, aabb, bbaa, aaaa, bbbb, \ldots\}. \end{aligned}$$

Ces opérations élémentaires permettent de définir une famille relativement riche de langages, dits *rationnels*.

Définition 2. L'ensemble $\mathrm{rat}(A^*)$ des *langages rationnels* sur A est la plus petite famille de langages sur A vérifiant les conditions suivantes :

1. $\mathrm{rat}(A^*)$ contient les ensembles $\emptyset$ et $\{a\}$, pour tout $a \in A$,
2. $\mathrm{rat}(A^*)$ est clos pour les opérations d'union finie, de produit fini et d'étoile.

1.2.1.2 Expressions rationnelles

Dans le but de simplifier les notations ensembliste, Kleene [Kle56] a utilisé, pour décrire les langages rationnels, les *expressions rationnelles*

Définition 3. Les expressions rationnelles sur un alphabet A sont définies récursivement comme suit :

1. 0 est l'expression rationnelle qui dénote le langage vide,
2. 1 est l'expression rationnelle qui dénote le langage $\{\varepsilon\}$,

3. a est l'expression rationnelle qui dénote le langage $\{a\}$ où $a \in A$,
4. si E et F sont des expressions rationnelles dénotant les langages $L(E)$ et $L(F)$, alors les expressions suivantes sont rationnelles :
   (a) $E + F$ dénote le langage $L(E) \cup L(F)$,
   (b) $E \cdot F$ dénote le langage $L(E) \cdot L(F)$,
   (c) E^* dénote le langage $(L(E))^*$,
   (d) (E) dénote le langage $L(E)$.

Les symboles 0 et 1 sont des constantes, $+$ et $\cdot$ sont des opérateurs binaires, $*$ est un opérateur unaire. On note $\exp(A)$ l'ensemble des expressions rationnelles sur l'alphabet A. Cet ensemble sera considéré modulo les règles suivantes :

$$\begin{array}{ccccccccc} 0 & + & E & = & E & + & 0 & = & E \\ 1 & \cdot & E & = & E & \cdot & 1 & = & E \\ 0 & \cdot & E & = & E & \cdot & 0 & = & 0 \end{array}$$

La *taille d'une expression rationnelle* E, notée $|E|$, est le nombre d'opérateurs et de lettres de E.

Exemple 4. Considérons l'alphabet $A = \{a, b\}$:
– L'alphabet A peut être représenté par l'expression rationnelle $a + b$.
– L'expression ba^* désigne l'ensemble des mots commençant par la lettre b et suivis d'un nombre quelconque (éventuellement nul) de a.
– L'expression $(bba)^*$ désigne l'ensemble des mots constitués d'une suite quelconque de facteurs bba.
– L'expression A^*a désigne l'ensemble des mots se terminant par la lettre a.
– L'expression $(b + ab)^*(1 + a)$ désigne l'ensemble des mots ne comprenant jamais deux a consécutifs.

1.2.1.3 Automates

Nous allons maintenant définir une structure permettant de représenter de façon finie certains ensembles éventuellement infinis de mots.

Définition 4. Un *automate*[5] $\mathcal{A}$ sur un alphabet A est un quintuplet $\langle Q, A, I, \delta, F\rangle$ tel que :
– Q est un ensemble fini d'éléments appelés *états*,
– A est un alphabet,
– $I \subseteq Q$ est un ensemble d'états dits *initiaux*,
– $\delta \ : \ Q \times A \longrightarrow 2^Q$ est la *fonction de transition* permettant de passer d'un état à un ensemble d'états en lisant une lettre de A,
– $F \subseteq Q$ est un ensemble d'états, qu'on appelle *états finaux*.

On remarque que la notion $q' \in \delta(q, a)$ est équivalente à $q \xrightarrow{a} q'$.

Un automate sera représenté graphiquement en adoptant les conventions suivantes :

5. On dit aussi *automate fini non-déterministe*.

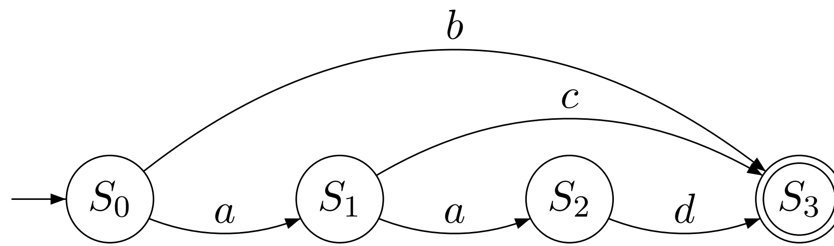


FIGURE 1.1 – Représentation graphique d'un automate.

- chaque état est matérialisé par un nœud ; à l'intérieur duquel on mettra éventuellement le nom de l'état.
- une transition est vue comme une flèche d'un état à un autre, à côté de laquelle on note son étiquette. Si plusieurs transitions vont d'un état q à un état p alors on ne tracera généralement qu'une seule flèche et les étiquettes seront séparées par des virgules ",",
- un état initial est marqué par l'indice zéro, ou une flèche entrante sans origine,
- un état final est marqué par un double cercle.

Le terme *automate* suggère une structure plus mécanique que cette combinaison de ronds et de flèches. Un automate tel qu'on l'a défini peut être vu comme une machine dont la mémoire est constituée d'un nombre fini d'états. Lorsque la mémoire est dans un état donné, la machine peut effectuer un nombre fini d'actions, l'état de la mémoire étant alors modifié en fonction de l'action choisie.

La description formelle du *fonctionnement* d'un automate nécessite l'introduction de plusieurs notions.

Définition 5. Étant donné un automate fini $\mathcal{A} = \langle Q, A, I, \delta, F \rangle$, un *chemin* fini $C = c_0 c_1 \ldots c_n$ dans $\mathcal{A}$ est une suite finie de transitions c_i telles que, pour tout entier $0 < i \leq n$, la transition c_i part de l'état dans lequel arrive la transition c_{i-1}. L' *étiquette* du chemin C, notée $m(C)$, est alors la concaténation des étiquettes de toutes les transitions qui le composent. Formellement, on notera : $C = q_0 \xrightarrow{a_0} q_1 \xrightarrow{a_1} \ldots q_n \xrightarrow{a_n} q_{n+1}$ où $q_i \xrightarrow{a_i} q_{i+1} = c_i$.

Les états q_0 et q_{n+1} sont appelés respectivement l'*origine* et l'*extrémité* du chemin C. L'étiquette du chemin C est $m(C) = a_0 a_1 a_2 \cdots a_n \in A^*$.

On dit qu'un chemin fini est *réussi* si son origine est un état initial et son extrémité un état final. L'ensemble de tous les chemins réussis d'un automate $\mathcal{A}$ sera noté par $C_{\mathcal{A}}$. Un mot $u \in A^*$ est dit *reconnu* ou *accepté* par l'automate $\mathcal{A}$ s'il existe un chemin réussi d'étiquette u. L'ensemble des mots reconnus par $\mathcal{A}$ est appelé le *langage reconnu* par $\mathcal{A}$ et noté $\mathcal{L}(\mathcal{A})$. On a donc : $\mathcal{L}(\mathcal{A}) = \{u \in A^* \mid \exists C \in C_{\mathcal{A}}$ tel que $m(C) = u\}$.

Un langage reconnu par un automate fini est dit *reconnaissable par automate fini* . L'ensemble des langages reconnaissables sur A est une partie de $\mathcal{P}(A^*)$ notée $Rec(A^*)$.

Deux automates sont *équivalents* s'ils reconnaissent le même langage.

Un état q d'un automate fini est dit *accessible* s'il existe un chemin d'un état initial vers q ; il est *co-accessible* s'il existe un chemin de q vers un état final. Un *état puits* est un état non final q tel que toute transition d'origine q a aussi q pour extrémité.

Un automate est dit *accessible* (resp. *co-accessible*) si tous ses états sont accessibles (resp.

co-accessibles).

On ne change pas le langage reconnu par un automate en supprimant un ou plusieurs états non accessibles et/ou non co-accessible. Un automate est dit *émondé* si tous ses états sont à la fois accessibles et co-accessibles.

Un automate est *standard* s'il possède un unique état initial sans transition entrante.

Un automate est *normalisé* s'il est standard et possède un unique état final différent de l'état initial sans transition sortante.

Un automate est *acyclique* s'il ne contient pas de circuit ; autrement dit, il n'existe pas de suite de transitions dans l'automate qui permette de passer deux fois par le même état.

1.2.1.3.1 Automate complet Un automate est *complet* si, pour chaque état $p \in Q$ et pour chaque lettre $a \in A$, il existe au moins une transition étiquetée par a sortante de p. Si un automate n'est pas complet, on peut le compléter sans changer le langage reconnu en ajoutant un état puits s, et en ajoutant les transitions $\delta(p, a) = \{s\}$ pour tout couple (p, a) tel que $\delta(p, a)$ n'est pas défini.

1.2.1.3.2 Élimination des ε-transitions Il est souvent avantageux d'autoriser les transitions entre deux états distincts étiquetées par le mot vide ε, afin de construire facilement des automates reconnaissant certains langages, comme le montre l'exemple suivant :

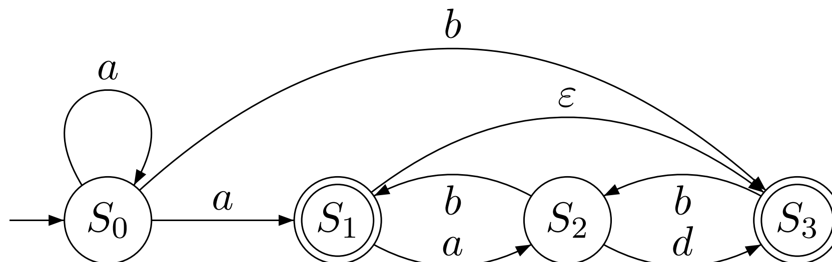


FIGURE 1.2 – Un automate avec des ε transitions.

Définition 6. Une *ε-transition* est une transition de p vers q, $p \neq q$, étiquetée par ε. On dit que $\mathcal{A} = \langle Q, A \cup \{\varepsilon\}, I, \delta, F \rangle$ est un ε-automate.

L'utilisation des ε- transitions facilite la description des langages mais pose quelques problèmes calculatoires. En effet, si après la lecture d'un mot, un automate se trouve dans un état q d'où part une ε-transition, lorsque la lettre suivante est lue, il est nécessaire d'examiner tous les états atteignables à partir de q, par une ou plusieurs ε-transitions successives, afin de savoir si ces états sont l'origine d'une transition étiquetée par cette lettre. D'un point de vue pratique, les calculs effectués par un automate sans ε-transitions sont donc plus efficaces.

Proposition 1. *Pour tout ε-automate $\mathcal{A}$, on peut construire un automate équivalent sans ε-transition ayant le même ensemble d'états.*

Pour les automates finis, on peut construire un automate sans ε-transition $\langle Q, A, I, \delta, F \rangle$ à partir d'un ε-automate $\mathcal{A} = \langle Q_\varepsilon, A \cup \{\varepsilon\}, I_\varepsilon, \delta_\varepsilon, F_\varepsilon \rangle$ comme suit :

- On ordonne les états de manière quelconque (on peut choisir un ordre topologique par exemple).
- Tant qu'il existe au moins une ε-transition, on applique la procédure suivante :
  Soit k le plus grand état (suivant l'ordre choisi) dans lequel aboutit au moins à une ε-transition : pour tout $q \in Q_\varepsilon \setminus \{k\}$ tel que $\delta_\varepsilon(q, \varepsilon) = k$, et pour tous $q' \in Q_\varepsilon$, $a \in A \cup \{\varepsilon\}$, tels que $\delta_\varepsilon(k, a) = q'$, on ajoute une transition $\delta(q, a) = q'$.
- On supprime les transitions $\delta_\varepsilon(q, \varepsilon) = k$.
- Si k est un état final, on ajoute q à l'ensemble des états finaux.
- Si k n'est plus accessible, on le supprime.

Plus généralement le problème calculatoire se pose lorsqu'il y a ambiguïté sur le chemin à parcourir dans un automate. Les ε-transitions causent des ambiguïtés en cours de calcul, sur l'état dans lequel se trouve l'automate. Des ambiguïtés peuvent également venir de l'existence de deux transitions étiquetées par une même lettre et partant d'un même état ; c'est la notion du non-déterminisme.

1.2.1.3.3 Déterminisme Une propriété remarquable des automates est le déterminisme. On remarque que le test d'appartenance d'un mot au langage reconnu par un automate déterministe est linéaire en fonction de la longueur du mot.

Définition 7. On dit qu'un automate fini est *déterministe* s'il satisfait les conditions suivantes :
- l'ensemble des états initiaux est réduit à un seul élément,
- aucune transition n'est étiquetée par le mot vide,
- à partir d'un état donné, il part au plus une seule transition étiquetée par une lettre donnée.

Le déterminisme d'un automate permet de n'avoir besoin de parcourir qu'un seul chemin pour savoir si un mot est reconnu.

Théorème 1. *(cf.*[RS59]*)Pour tout automate fini, on peut construire un automate déterministe reconnaissant le même langage en un temps* $O(2^{|Q|})$.

En pratique, pour déterminiser un automate, on ne construit pas en entier l'automate ensembliste qui est décrit dans la preuve de la déterminisation, mais on fait plutôt une recherche des descendants de l'état regroupant les états initiaux de l'automate. En maintenant la liste des états déjà construits de l'automate résultant plus l'ensemble des transitions qui reste à explorer et qui sont composées d'un état déjà construit et une lettre de l'alphabet. À chaque étape, on choisit un couple (état, lettre) et on construit l'état successeur, si ce dernier n'existe pas dans la liste des états déjà construits on doit l'ajouter sinon on passe à la transition suivante.

1.2.1.3.4 Automate homogène Une autre propriété remarquable dans certains automates est l'homogénéité.

Définition 8. Soit $\mathcal{A}$ un automate fini. Si, pour chaque état q de $\mathcal{A}$, toutes les transitions entrantes de q ont la même étiquette alors on dit que $\mathcal{A}$ est un automate *homogène*. Dans ce

cas on peut étiqueter chaque état par l'étiquette de ses transitions entrantes et on note $h(q)$ la fonction qui retourne l'étiquette des transitions entrantes de l'état q.

1.2.1.3.5 Automate quotient Soient $\mathcal{A} = \langle Q, A, I, \delta, F\rangle$ un automate et $\sim$ une relation d'équivalence sur l'ensemble des états Q. On dit que $\sim$ est régulière à droite sur $\mathcal{A}$ si, et seulement si :

- $\sim \subseteq (Q \setminus F)^2 \cup F^2$, c'est-à-dire les états finaux et non-finaux ne sont pas $\sim$-équivalents,
- pour tout $p, q \in Q$, $a \in A$, si $p \sim q$, alors $\delta(p, a)/_\sim = \delta(q, a)/_\sim$.

Si $\sim$ est régulière à droite sur $\mathcal{A}$, l' *automate quotient* $\mathcal{A}_\sim = \langle Q_\sim, A, I_\sim, \delta_\sim, F_\sim\rangle$ est défini comme suit :

- $F_\sim = \{[p]_\sim \mid p \in F\}$,
- $Q_\sim = (Q \setminus F)_\sim \cup F_\sim$,
- pour tout $[p]_\sim \in Q$, $a \in A$, on a $\delta_\sim([p]_\sim, a) = \{[q]_\sim \mid q \in \delta(p, a)\}$.

Corollaire 1. *Si $\sim$ est une relation d'équivalence régulière à droite sur $\mathcal{A}$ alors $\mathcal{L}(\mathcal{A}_\sim) = \mathcal{L}(\mathcal{A})$.*

1.2.1.3.6 Automate minimal L'ensemble des automates déterministes reconnaissant un langage donné admet un plus petit élément par rapport au nombre d'états. Ce plus petit automate, appelé *automate déterministe minimal*, est unique à un isomophisme près. La représentation par l'automate déterministe minimal peut être extrêmement efficace comme méthode de compression.

Soient A un alphabet fini et $L \subseteq A^*$ un langage rationnel. Afin de calculer l'unique automate déterministe minimal en nombre d'états reconnaissant L, Myhill et Nerode [Myh57, Ner58] ont défini indépendamment la relation d'équivalence suivante : $u \sim v \iff u^{-1}L = v^{-1}L$. La relation $\sim$ est régulière à droite ; autrement dit, si $u \sim v$, alors pour tout $w \in L$, $uw \sim vw$.

Le théorème suivant est une variante du théorème de Myhill et Nerode [Myh57, Ner58]. Un résultat similaire sur les expressions rationnelles est montré par Brzozowski [Brz64].

Théorème 2. [HMU06] *Un langage $L \subseteq A^*$ est rationnel si, et seulement si, la relation d'équivalence $\sim$ est d'index fini.*

Soit $\mathcal{A} = \langle Q, A, I, \delta, F\rangle$ un automate fini. Le *langage droit* d'un état $q \in Q$, noté $\vec{\mathcal{L}}_q(\mathcal{A})$, est l'ensemble $\{u \in A^* \mid \delta(q, u) \in F\}$. Le *langage gauche* d'un état $q \in Q$, noté $\overleftarrow{\mathcal{L}}_q(\mathcal{A})$, est l'ensemble $\{u \in A^* \mid \exists i \in I \text{ t.q. } \delta(i, u) = q\}$.

Corollaire 2. *Soit $\mathcal{A} = \langle Q, A, I, \delta, F\rangle$ un automate fini. $\mathcal{A}$ est l'automate déterministe et complet ayant le moins d'états reconnaissant $\mathcal{L}(\mathcal{A})$ si, et seulement si, les langages droits des états de $\mathcal{A}$ sont distincts.*

On peut voir la relation d'équivalence $\sim$ sur les états de l'automate $\mathcal{A}$ comme suit : $p \sim q \iff \vec{\mathcal{L}}_p(\mathcal{A}) = \vec{\mathcal{L}}_q(\mathcal{A})$. Comme conséquence, pour un automate déterministe complet $\mathcal{A}$, l'automate quotient $\mathcal{A}_\sim$ est l'automate déterministe complet minimal en nombre d'états reconnaissant le langage $\mathcal{L}(\mathcal{A})$.

D'autres variantes d'algorithmes de minimisation ont été mises au point [Moo56, Rev92, Hop71a, Brz64].

1.2.2 Logique

Nous examinons dans ce travail plusieurs formalismes logiques traitant différentes structures relationnelles. Examinons maintenant brièvement les notions essentielles de ces formalismes logiques (pour plus de détails, voir par exemple [Tho97]).

1.2.2.1 Formalismes logiques

Rappelons que pour un alphabet fini ou infini A, *un mot fini* est une séquence finie de symboles (lettres) de l'alphabet A. On note A^* l'ensemble des mots finis sur A. Nous appelons *mot fini sur les entiers naturels* un mot de A^* où $A = \mathbb{N}$. Nous appelons *mot fini sur les entiers* un mot de A^* où $A = \mathbb{Z}$. Dans ce qui suit, *mot* désigne toujours *mot fini*. On note ε le mot vide.

Définition 9. Étant donnée une signature $\mathfrak{L}$, une $\mathfrak{L}$-*structure* relationnelle $\mathfrak{A} = (A, \dots, R_i, \dots)$ est un uplet, où A est un ensemble non vide, appelé *domaine* de $\mathfrak{A}$, et où chaque $R_i \subseteq A^{r_i}$ est une relation d'arité r_i sur A.

Par FO, *first order*, nous entendons *la logique du premier ordre égalitaire*. La logique du premier ordre permet de quantifier sur des éléments du domaine de la structure. Étant donnée une signature $\mathfrak{L}$, on peut définir l'ensemble des formules FO sur $\mathfrak{L}$ sous forme de formules bien formées qui contiennent :

1. des symboles de variables du premier ordre $x, y, \dots$ interprétés comme des éléments du domaine de la structure,
2. des symboles de $\mathfrak{L}$.

On notera $\models$ la relation de satisfaction d'une formule dans une structure. Un *énoncé* est une formule sans variable libre. Nous identifierons souvent les symboles logiques avec leur interprétation. Étant données une signature $\mathfrak{L}$ et une $\mathfrak{L}$-structure $\mathfrak{A}$ sur le domaine A, on dit qu'une relation $R \subseteq A^m$ est FO-*définissable* dans $\mathfrak{A}$ si et seulement s'il existe une formule $\varphi(x_1, \dots, x_m)$ sur $\mathfrak{L}$ telle que $\mathfrak{A} \models \varphi[a_1, \dots, a_m]$ si, et seulement si, on a $R(a_1, \dots, a_m)$. On note FO($\mathfrak{A}$) la théorie du premier ordre de $\mathfrak{A}$.

Nous considèrerons aussi MSO, *monadic second order logic*, en français *logique monadique du second ordre*. La logique monadique du second ordre est une extension de la logique du premier ordre qui permet de quantifier sur des éléments ainsi que sur des sous-ensembles du domaine de la structure. À partir d'une signature $\mathfrak{L}$, on peut définir l'ensemble des formules MSO sur $\mathfrak{L}$ sous forme de formules bien formées qui peuvent contenir :

1. des symboles de variables du premier ordre $x, y, \dots$ interprétés comme des éléments du domaine de la structure,
2. des symboles de variables monadiques du second ordre $X, Y, \dots$ interprétés comme des sous-ensembles du domaine,
3. des symboles de $\mathfrak{L}$,
4. et un nouveau prédicat binaire $x \in X$ interprété comme "x appartient à X".

Étant données une signature $\mathfrak{L}$ et une $\mathfrak{L}$-structure $\mathfrak{A}$ sur le domaine A, on dit qu'une relation $R \subseteq A^m \times (2^A)^n$ est MSO-*définissable* dans $\mathfrak{A}$ si et seulement s'il existe une formule $\varphi(x_1, \ldots, x_m, X_1, \ldots, X_n)$ sur $\mathfrak{L}$ telle que $\mathfrak{A} \models \varphi[a_1, \ldots, a_m, E_1, \ldots, E_n]$ si, et seulement si, on a $R(a_1, \ldots, a_m, E_1, \ldots, E_n)$. On note $\mathrm{MSO}(\mathfrak{A})$ la théorie monadique du second ordre de $\mathfrak{A}$.

La théorie logique (FO ou MSO) d'une structure relationnelle est dites *décidable*, s'il existe un *algorithme* qui détermine si un énoncé donné φ est vrai dans cette structure.Pour un alphabet infini A, nous nous intéressons à des logiques sur A^* afin d'exprimer l'égalité entre les symboles de A et des propriétés plus complexes comme celles de l'arithmétique (par exemple $<$ et $+$) pour $A = \mathbb{N}$ ou $\mathbb{Z}$.

1.2.2.2 Structures automatiques

Nous rappelons ici des notions et des résultats utiles portant sur les structures automatiques [Hod83, KN94, BG00].

Dans toutes les preuves présentées dans ce travail on aura besoin de coder chaque n-uplet de mots selon un codage approprié pour établir les résultats d'automaticité et/ou de décidabilité. Ceci conduit à l'utilisation des automates synchrones multi-bandes.

Soit A un alphabet fini. Étant donné $(v_1, \ldots, v_n)$ un n-uplet de mots sur A, on introduit un symbole de remplissage $\#$, et nous complétons (si nécessaire) chaque mot v_i avec un nombre suffisant de # afin d'avoir des mots de même longueur. Ce faisant, nous obtenons n mots sur $A \cup \{\#\}$ qui ont la même longueur. Ceci peut être vu comme un seul mot sur l'alphabet $(A \cup \{\#\})^n$ (c'est-à-dire l'alphabet des n-uplets d'éléments de $A \cup \{\#\}$). Ce mot sera noté $\langle v_1, \ldots, v_n \rangle$.

Définition 10. Soit A un alphabet fini. Une structure relationnelle $\mathfrak{A} = (A, \ldots, R_i, \ldots)$ est *automatique* s'il existe une application injective $\mu : A \to (A \cup \{\#\})^*$ telle que les images par μ de A et toutes les R_i sont régulières :

1. $\mu(A) = \{\mu(x) \mid x \in A\}$ est un langage rationnel sur l'alphabet A,
2. $\mu(R_i) = \{\langle \mu(w_1), \ldots, \mu(w_{r_i}) \rangle \mid (w_1, \ldots, w_{r_i}) \in R_i\}$ est un langage rationnel sur l'alphabet $(A \cup \{\#\})^{r_i}$.

Cela signifie qu'on peut trouver une fonction injective, appelée *codage*, qui transforme le domaine et chaque relation de la structure, en un langage rationnel, c'est-à-dire reconnaissable par un automate fini. La relation entre l'automaticité d'une structure et la décidabilité de sa théorie est énoncée par le théorème suivant.

Théorème 3 ([BG00]). *Si une structure relationnelle $\mathfrak{A}$ est automatique alors $\mathrm{FO}(\mathfrak{A})$ est décidable.*

1.2.2.3 Interprétation logique du premier ordre

Soient $\mathfrak{L}$ et $\mathfrak{L}'$ deux signatures, $\mathfrak{A}$ une $\mathfrak{L}$-structure sur le domaine A et $\mathfrak{A}'$ une $\mathfrak{L}'$-structure sur le domaine A'.

On dit que $\mathfrak{A}$ est FO-*interprétable* dans $\mathfrak{A}'$ s'il existe un sous-ensemble D de A' et une application bijective $\mu : A \to D$ avec :

1. D est FO-définissable dans $\mathfrak{A}'$,
2. Pour chaque symbole m-aire R de $\mathfrak{L}$, il existe une $\mathfrak{L}'$-formule φ_R de FO telle que, pour tous $a_1, \ldots, a_m \in A$:

$$\mathfrak{A} \models R[a_1, \ldots, a_m] \Leftrightarrow \mathfrak{A}' \models \varphi_R[\mu(a_1), \ldots, \mu(a_m)]$$

La propriété d'interprétation suivante est bien connue [Rab77] :

Lemme 1. *Si* $\mathrm{FO}(\mathfrak{A}')$ *est décidable et si* $\mathfrak{A}$ *est* FO-*interprétable dans* $\mathfrak{A}'$ *alors* $\mathrm{FO}(\mathfrak{A})$ *est décidable.*

1.2.2.4 Interprétation logiques du second ordre

Soient $\mathfrak{L}$ et $\mathfrak{L}'$ deux signatures, $\mathfrak{A}$ une $\mathfrak{L}$-structure sur le domaine A et $\mathfrak{A}'$ une $\mathfrak{L}'$-structure sur le domaine A'.

On dit que $\mathfrak{A}$ est MSO-*interprétable* dans $\mathfrak{A}'$ s'il existe un sous-ensemble D de A' et une application bijective $\mu : A \to D$ tels que :

1. D est MSO-définissable dans $\mathfrak{A}'$,
2. Pour chaque symbole m-aire R de $\mathfrak{L}$, il existe une $\mathfrak{L}'$-formule φ_R de MSO telle que, pour tous $a_1, \ldots, a_m \in A$:

$$\mathfrak{A} \models R[a_1, \ldots, a_m] \Leftrightarrow \mathfrak{A}' \models \varphi_R[\mu(a_1), \ldots, \mu(a_m)]$$

La propriété d'interprétation suivante est bien connue [Rab77] :

Lemme 2. *Si* $\mathrm{MSO}(\mathfrak{A}')$ *est décidable et si* $\mathfrak{A}$ *est* MSO-*interprétable dans* $\mathfrak{A}'$ *alors* $\mathrm{MSO}(\mathfrak{A})$ *est décidable.*

1.2.2.5 Structures universelles

Si une structure $\mathfrak{A}$ est automatique alors toute relation FO-définissable dans $\mathfrak{A}$ est rationnelle. *Que peut-on dire sur l'inverse ?*

Si toute relation rationnelle est FO-définissable dans $\mathfrak{A}$ on dit que $\mathfrak{A}$ est *universelle*.

Un exemple de structure automatique universelle est la structure $(\mathbb{N}; +, \epsilon_k(x, y))$ où le prédicat $\epsilon_k(x, y)$ est vrai si, et seulement si, x est une puissance de k qui apparaît dans l'écriture en base k de y. La théorie logique du premier ordre de cette structure est décidable tandis que sa théorie monadique du second ordre ne l'est pas [BG00, Büc60].

La structure $\mathfrak{P} = (P_f(\mathbb{N}); \subseteq, <)$ est définie comme suit :

1. $P_f(\mathbb{N})$ est l'ensemble de toutes les parties finies de $\mathbb{N}$.
2. $x < y$ si $x = \{n\}$ et $y = \{m\}$ (x et y sont des singletons) et $n < m$.

La structure $\mathfrak{P}$ est universelle. On peut démontrer que la structures $(\mathbb{N}; +, \epsilon_2(x, y))$ et la structure $(P_f(\mathbb{N}); \subseteq, <)$ sont interprétables les unes dans les autres. Il suffit de coder chaque partie finie e de $P_f(\mathbb{N})$ par un entier naturel n dont la représentation binaire donne les éléments de

e. Le $i^{\text{ème}}$ bit (de gauche à droit) égale "un" si l'entier i appartient à e, et "zéro" sinon. Ce codage est une bijection : le codage permet d'établir l'interprétation dans un sens et la bijection inverse permet d'établir l'interprétation dans l'autre sens (voir [Vil92]).

La double interprétation entre ces deux structures permet à la fois de déduire la décidabilité de la théorie logique du premier ordre de la structure $\mathfrak{P}$ et l'indécidabilité de sa théorie monadique de second ordre.

La propriété de la double interprétation suivante sera utile dans la suite :

Lemme 3. *Soient $\mathfrak{A}$ et $\mathfrak{A}'$ des structures relationnelles. Si $\mathfrak{A}$ est universelle, $\mathfrak{A}$ est* FO*-interprétable dans $\mathfrak{A}'$ par la bijection b, et $\mathfrak{A}'$ est* FO*-interprétable dans $\mathfrak{A}$ par b^{-1}, alors $\mathfrak{A}'$ est universelle.*

Idée de preuve. En passant par la double interprétation (codage bijectif) on peut sans difficulté montrer que chaque relation de $\mathfrak{A}$ est interprétable par une formule de $\mathfrak{A}'$ définissant une relation rationnelle. Inversement chaque relation de $\mathfrak{A}'$ est interprétable par une formule de $\mathfrak{A}$ définissant une relation rationnelle. □

1.2.2.6 $\mathfrak{M}$-automates

Pour prouver d'autres résultats de décidabilité nous utilisons la notion de $\mathfrak{M}$-*automate* introduite par Bès dans [Bès08]. Le concept de $\mathfrak{M}$-automate est une notion naturelle d'automates pour les mots finis sur un alphabet infini. Nous introduisons dans ce qui suit la notion de relation $\mathfrak{M}$-*reconnaissable* ainsi que la relation entre la $\mathfrak{M}$-*automaticité* et la décidabilité.

Comme dans le paragraphe 1.2.2.2, on représentera un n-uplet $(w_1, \ldots, w_n)$ de mots sur A par un mot sur l'alphabet $(A \cup \{\#\})^n$ noté $\langle w_1, \ldots, w_n \rangle$, où # est un symbole de remplissage.

Considérons un langage $\mathfrak{L}$ et une $\mathfrak{L}$-structure $\mathfrak{M}$ avec A comme domaine. Nous allons associer à $\mathfrak{M}$ la structure $\mathfrak{M}_\#$ dans le langage étendu $\mathfrak{L}_\# = \mathfrak{L} \cup \{P_\#\}$, telle que :

1. le domaine de $\mathfrak{M}_\#$ est $A \cup \{\#\}$,
2. pour chaque symbole relationnel R de $\mathfrak{L}$, les interprétations de R dans $\mathfrak{M}$ et $\mathfrak{M}_\#$ sont les mêmes,
3. le prédicat $P_\#(x)$ est vérifié dans $\mathfrak{M}_\#$ si, et seulement si, $x = \#$.

Un $\mathfrak{M}$-automate est un automate fini synchrone non déterministe à n bandes, qui lit des mots finis sur A. Les règles de transition sont des triplets de la forme (q, φ, q'), où q, q' sont des états de l'automate, et $\varphi(x_1, \ldots, x_n)$ est une formule du premier ordre dans le langage $\mathfrak{L}_\#$ de $\mathfrak{M}_\#$. La transition (q, φ, q') peut être franchie si le n-uplet des symboles lus par les n têtes satisfait φ dans $\mathfrak{M}_\#$.

Définition 11. Soient A un alphabet et $\mathfrak{M}$ une $\mathfrak{L}$-structure ayant A comme domaine. Un $\mathfrak{M}$-*automate* est défini par un 7-uplet $\mathcal{A} = (Q, n, A, \mathfrak{M}, E, I, T)$ tel que :

1. Q est un ensemble fini (d'états),
2. $n \in \mathbb{N}^+$ est le nombre de bandes,
3. $E \subseteq Q \times \mathcal{F}_n \times Q$ est l'ensemble des transitions, où $\mathcal{F}_n$ représente l'ensemble des $\mathfrak{L}_\#$-formules avec n variables libres,

4. $I \subseteq Q$ est l'ensemble des états initiaux,
5. $T \subseteq Q$ est l'ensemble des états terminaux.

Étant donné un n-uplet $w = (w_1, \ldots, w_i, \ldots, w_n)$ de mots sur A, un *chemin* λ dans $\mathcal{A}$ étiqueté par $\langle w \rangle$ est une séquence d'états $\lambda = (q_0, \ldots, q_i, \ldots, q_m)$, où $m = |\langle w \rangle|$, $q_0 \in I$, et pour chaque $0 \leq i < m$ il existe une $\mathfrak{L}_\#$-formule $\varphi(x_1, \ldots, x_n)$ avec $(q_i, \varphi, q_{i+1}) \in E$ et

$$\mathfrak{M}_\# \models \varphi(\pi_1(\langle w \rangle)[i], \ldots, \pi_n(\langle w \rangle)[i])$$

où $\pi_j(\langle w \rangle)$ désigne la j-ième composante de $\langle w \rangle$. Le chemin λ est *accepté* si $q_m \in T$. On dit que w est accepté par $\mathcal{A}$ si $\langle w \rangle$ est l'étiquette d'un chemin accepté. On note par $L(\mathcal{A})$ l'ensemble des mots $w \in (A^*)^n$ qui sont acceptés par $\mathcal{A}$.

Définition 12. Soit $n \geq 1$. Une relation $X \subseteq (A^*)^n$ est dite $\mathfrak{M}$-*reconnaissable* si et seulement s'il existe un $\mathfrak{M}$-automate $\mathcal{A}$ à n bandes tel que $X = L(\mathcal{A})$.

Exemple 5. Pour $\mathfrak{M} = (\mathbb{Z}; <, 0)$, l'ensemble des mots formés de lettres positives, et finissant par une lettre négative, est $\mathfrak{M}$-reconnaissable. L'automate suivant est un $\mathfrak{M}$-automate reconnaissant cet ensemble. Il est composé de deux états q_0, q_1, tel que q_0 est initial et q_1 est terminal, et dont l'ensemble des transitions est $\{(q_0, \varphi_0, q_0), (q_0, \varphi_1, q_1)\}$ tel que φ_0 est la formule $x \geq 0$, et φ_1 exprime que $x \leq 0$.

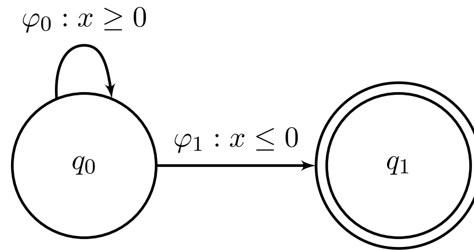


FIGURE 1.3 – Une exemple d'un $\mathfrak{M}$-automate.

Cet automate accepte le mot $[2, 7, 9, 12, 5, 2, -5]$, mais pas $[7, -8, 9, 9, 7, -1]$.

Définition 13. Soient $\mathfrak{M} = (A; ...)$ et $\mathfrak{N} = (N; \ldots, R_i, \ldots)$ deux structures. On dit que $\mathfrak{N}$ est $\mathfrak{M}$-*automatique* s'il existe une application injective $c : N \to A^*$ telle que les images par c de N et toutes les relations R_i sont $\mathfrak{M}$-reconnaissables.

Théorème 4 ([Bès08]). *Soit $\mathfrak{N}$ une structure relationnelle $\mathfrak{M}$-automatique. Si* FO($\mathfrak{M}$) *est décidable alors* FO($\mathfrak{N}$) *est décidable.*

On a présenté dans ce chapitre les objets et les notions nécessaires de la logique, de la théorie des langages et des automates finis pour la bonne compréhension et formalisation du reste du travail.

Première partie

Partie théorique

CHAPITRE 2

AUTOMATICITÉ ET DÉCIDABILITÉ POUR DES LOGIQUES DES MOTS SUR UN ALPHABET INFINI

Le but de ce chapitre est de montrer l'automaticité de plusieurs structures logiques du premier ordre dont le domaine est l'ensemble des mots finis écrits sur un alphabet infini dénombrable. Nous sommes intéressés, en particulier, par les structures contenant la relation du préfixe, le prédicat $\mathfrak{clone}$, qui est vrai lorsqu'un mot se termine par deux lettres identiques, et le prédicat $\mathfrak{diff}$, qui est lui vrai quand toutes les lettres d'un mot sont (deux à deux) distinctes. Comme corollaire de l'automaticité, nous obtenons la décidabilité des théories du premier ordre correspondantes. D'autre résultats d'indécidabilité, et des résultats de décidabilité pour les théories monadiques du second ordre, sont également présentés :

Alphabet Σ	**Structure**	**Théorie** FO	**Théorie** MSO	**Automaticité**
$\mathbb{Z}$	$\mathfrak{S}_0 = (\Sigma^*; \prec, \mathfrak{clone}, \mathfrak{less}, \{\mathfrak{mod}_{p,q} \mid p, q \in \mathbb{N}\})$	Décidable	Décidable	Automatique
	$\mathfrak{S}_1 = (\Sigma^*; \prec, \mathfrak{clone}, \mathfrak{less}, \{\mathfrak{mod}_{p,q} \mid p, q \in \mathbb{N}\}, \sim, \oplus)$	Décidable	?	$\mathfrak{M}$-automatique
	$\mathfrak{S}_9 = (\Sigma^*; \prec, \mathfrak{clone}, \mathfrak{lastNew}, \mathfrak{firstZero}, \sim, \ominus)$	Indécidable	Indécidable	Non automatique
Infini dénombrable	$\mathfrak{S}_2 = (\Sigma^*; \prec, \mathfrak{clone}, \{\mathfrak{diff}_i \mid i > 2\})$	Décidable	Décidable	Automatique (i borné)
	$\mathfrak{S}_3 = (\Sigma^*; \prec, \mathfrak{clone}, \{\mathfrak{diff}_i \mid i > 2\}, \sim)$	Décidable	?	$\mathfrak{M}$-automatique (i borné)
	$\mathfrak{S}_4 = (\Sigma^*; \prec, \mathfrak{clone}, \mathfrak{diff})$	Décidable	Décidable	Automatique
	$\mathfrak{S}_5 = (\Sigma^*; \prec, \mathfrak{clone}, \mathfrak{diff}, \sim)$	Décidable	?	$\mathfrak{M}$-automatique
	$\mathfrak{S}_6 = (\Sigma^*; \prec, \{R^{\oplus}_{f_i} \mid i > 0\}, \sim)$	Décidable	?	$\mathfrak{M}$-automatique (i borné)
	$\mathfrak{S}_7 = (\Sigma^*; \prec, <, \mathfrak{dec}_\star)$	Décidable	Indécidable	Universelle
	$\mathfrak{S}_8 = (\Sigma^*; \prec, <, \mathfrak{dec}_\star, \sim)$	Indécidable	Indécidable	Non automatique

TABLE 2.1 – Principaux résultats

Pour montrer l'intérêt des résultats de décidabilité obtenus nous allons revenir sur l'exemple 2 présenté dans l'introduction. Considérons un système de gestion de processus qui stocke périodiquement une liste des processus qui ont accédé à une zone mémoire bien précise. Montrons comment on peut exprimer formellement certaines propriétés du système en utilisant les prédicats logiques étudiés dans le chapitre.

Soit $\Sigma = \mathbb{N}$:

1. Pour des mots x et y de Σ^* :
   - On note $x \preceq y$ si, et seulement si, x est un préfixe de y.
   - On note $x \sim y$ si, et seulement si, x et y ont la même longueur : $x \sim y \Leftrightarrow |x| = |y|$.
2. Pour un mot x de Σ^* :
   - Le prédicat $\mathfrak{clone}(x)$ est vrai si, et seulement si, x se termine par deux lettres identiques : $\mathfrak{clone}(x) \Leftrightarrow x = uaa$ avec $u \in \Sigma^*$ et $a \in \Sigma$.
   - Le prédicat $\mathfrak{less}(x)$ est défini par : $\mathfrak{less}(x) \Leftrightarrow x = uab$ avec $u \in \Sigma^*$, $a, b \in \Sigma$ et $a < b$.
   - Le prédicat $\mathfrak{diff}(x)$ est vrai si, et seulement si, toutes les lettres de x sont distinctes $\mathfrak{diff}(x) \Leftrightarrow x = x_1 x_2 \ldots x_n$ avec $x_1, x_2, \ldots, x_n \in \Sigma$ et $\forall i \forall j \; i \neq j \Rightarrow x_i \neq x_j$.

Soit $l \in \Sigma^*$ une liste d'identifiants de processus. Prenons la liste suivante :

$$l = [3, 4, 2, 1, 1, 2, 0, 7, 7]$$

Plusieurs propriétés intéressantes peuvent être considérées :

1. *Les deux derniers accès (à la zone mémoire) ont été effectués par deux processus différents.* Cette propriété n'est pas vérifiée pour l. Elle peut être exprimée par la formule logique qui dit que les deux derniers éléments de la liste sont distincts :

$$\neg\mathfrak{clone}(l)$$

2. *La liste contient deux accès consécutifs du même processus.* Cette propriété est vérifiée pour l. Elle peut être exprimée par la formule logique qui dit que le prédicat $\mathfrak{clone}$ est vrai pour au moins un préfixe de la liste :

$$\exists p \; (p \preceq l) \wedge \mathfrak{clone}(p)$$

3. *Tous les accès sont effectués par le même processus.* Cette propriété n'est pas vérifiée pour l. Elle peut être exprimée par la formule logique qui dit que le prédicat $\mathfrak{clone}$ est vrai pour tous les préfixes (de la liste) de longueur supérieure à 2 :

$$\forall p \; ((p \preceq l) \wedge (|p| > 1)) \rightarrow \mathfrak{clone}(p)$$

   avec : $|p| > 1$ qui peut être exprimé comme suit :

$$\exists q (q \neq \varepsilon \wedge q \neq p \wedge q \preceq p)$$

4. *Chaque processus de la liste a effectué un seul accès.* Cette propriété n'est pas vérifiée pour l. Elle peut être exprimée par la formule logique qui dit que le prédicat $\mathfrak{diff}$ est vrai pour la liste :

$$\mathfrak{diff}(l)$$

5. *Les trois derniers accès sont effectués par le même processus*. Cette propriété n'est pas vérifiée pour l. Elle peut être exprimée par la formule logique qui dit que le prédicat $\mathfrak{clone}$ est vrai pour la liste et pour le préfixe strict le plus long de la liste :

$$\mathfrak{clone}(l) \wedge \exists p(p \neq l \wedge p \preceq l \wedge \mathfrak{clone}(p) \wedge (\forall q(q \preceq l) \rightarrow (q \preceq p \vee q = l)))$$

6. *La liste est triée (tri décroissant)*. Cette propriété n'est pas vérifiée pour l. Elle peut être exprimée par la formule logique qui dit que le prédicat $\mathfrak{less}$ est faux pour tous les préfixes de la liste :

$$\forall p \ (p \preceq l) \rightarrow \neg\mathfrak{less}(p)$$

Dans la suite on détaille les résultats obtenus. Les définitions et les notations suivantes seront utilisées dans tout le chapitre.

Pour un alphabet fini ou infini Σ, *un mot fini* est une séquence finie de symboles (lettres) de l'alphabet Σ. On note Σ^* l'ensemble des mots finis sur Σ. Nous appelons *mot fini sur les entiers naturels* un mot de Σ^* où $\Sigma = \mathbb{N}$. Nous appelons *mot fini sur les entiers* un mot de Σ^* où $\Sigma = \mathbb{Z}$. Dans ce qui suit, *mot* désigne toujours *mot fini*. On note ε le mot vide.

2.1 Structure de clone

Nous définissons dans cette section la structure de clone, nous montrons son automaticité, et par conséquent, la décidabilité de sa théorie du premier ordre. On montre aussi la décidabilité de sa théorie monadique du second ordre.

Cette structure fait intervenir des prédicats arithmétiques qui conduisent à travailler avec $\Sigma = \mathbb{Z}$.

Définition 14. On note $\mathfrak{S}_0 = (\Sigma^*; \prec, \mathfrak{clone}, \mathfrak{less}, \{\mathfrak{mod}_{p,q} \mid p, q \in \mathbb{N}\})$ la *structure de clone* :

1. Σ^* désigne l'ensemble des mots finis sur l'alphabet infini $\Sigma = \mathbb{Z}$.
2. Pour des mots x et y de Σ^*, on a $x \prec y$ si, et seulement si, x est un préfixe strict de y.
3. Pour un mot x de Σ^*, le prédicat $\mathfrak{clone}(x)$ est vrai si, et seulement si, x se termine par deux lettres identiques : $\mathfrak{clone}(x) \Leftrightarrow x = uaa$ avec $u \in \Sigma^*$ et $a \in \Sigma$.
4. Pour un mot x de Σ^*, le prédicat $\mathfrak{less}(x)$ est défini par : $\mathfrak{less}(x) \Leftrightarrow x = uab$ avec $u \in \Sigma^*$, $a, b \in \Sigma$ et $a < b$.
5. Pour tout $p, q \in \mathbb{N}$, $\mathfrak{mod}_{p,q}$ est le prédicat défini comme suit : $\mathfrak{mod}_{p,q}(x) \Leftrightarrow x = uab$ avec $u \in \Sigma^*$, $a, b \in \Sigma$ et $(b - a) \equiv q[p]$.

Exemple 6. Soient $x, y \in \Sigma^*$ les mots $y = [-1, 2, 5, 2, -3, 4, 4]$ et $x = [-1, 2, 5]$. On peut vérifier que $x \prec y$, $\mathfrak{less}(x)$, $\mathfrak{clone}(y)$ sont vrais, et que $\mathfrak{mod}_{2,1}(x)$ est aussi vrai parce que $(5 - 2) \equiv 1[2]$.

2.1.1 Théorie du premier ordre de la structure de clone

Nous montrons dans cette section, l'automaticité de la structure de clone, d'où la décidabilité de sa théorie du premier ordre.

Théorème 5. *La structure de clone $\mathfrak{S}_o$ est automatique.*

Avant de détailler la preuve de ce théorème, introduisons quelques définitions.

Définition 15. Soit $x = x_1x_2\dots x_n \in \Sigma^*$ un mot de longueur n. Soit $\sigma_3 = \{a, b, c\}$. Nous appelons :

1. *différentiel* de x le mot noté par $d(x) = x_1d_2d_3\dots d_n \in \Sigma^*$ tel que $d_k = x_k - x_{k-1}$,
2. *représentation ternaire* de x le mot $t(x) = t_1t_2\dots t_n \in \sigma_3^*$ tel que :

$$t_i = \begin{cases} a^{x_i+1}b \text{ si } x_i \geq 0, \\ a^{|x_i|}c \text{ sinon.} \end{cases}$$

Par convention $d(\varepsilon) = t(\varepsilon) = \varepsilon$.

Exemple 7. Soit $y \in \Sigma^*$ avec $y = [-1, 5, 2, 2, -3, 4, 4]$. On a :

1. $d(y) = [-1, 6, -3, 0, -5, 7, 0]$,
2. $t(d(y)) = \underbrace{ac}_{-1}\underbrace{aaaaaaab}_{+6}\underbrace{aaac}_{-3}\underbrace{ab}_{0}\underbrace{aaaaac}_{-5}\underbrace{aaaaaaaab}_{+7}\underbrace{ab}_{0}.$

Démonstration. Pour prouver que la structure de clone est automatique, nous devons définir un codage μ qui établit la rationalité du domaine de la structure, du préfixe, du $\mathfrak{clone}$, du $\mathfrak{less}$ et de chaque prédicat $\mathfrak{mod}_{p,q}$ pour $p, q \in \mathbb{N}$. Ce codage renvoie la représentation ternaire du différentiel de chaque mot de Σ^*.

$$\begin{aligned} \mu : \Sigma^* &\longrightarrow \sigma_3^* \\ x &\longmapsto \mu(x) = t(d(x)) \end{aligned}$$

Premièrement, nous calculons le différentiel de x pour obtenir un mot à partir duquel nous pouvons facilement savoir si le mot x contient des séquences de lettres identiques. Cela est possible car les lettres nulles dans $d(x)$ correspondent aux lettres identiques dans x. En outre, le différentiel préserve le prédicat de préfixe et conserve des informations qui nous permettent de retrouver le mot x. C'est essentiellement pour cette raison que nous calculons la représentation ternaire du différentiel de x. Nous avons :

1. Le codage $\mu(\Sigma^*)$ du domaine de $\mathfrak{S}_o$ est rationnel : en effet, chaque mot non vide x est codé par un mot ternaire $\mu(x)$ débutant par a et se terminant par b ou c.

$$\mu(\Sigma^*) = (\{a\}^+\{b, c\})^*$$

2. Le langage $\mu(\prec)$ est rationnel : on peut voir qu'un mot est préfixe d'un autre si, et seulement si, le codage du premier mot est préfixe du codage du second.

$$x, y \in \Sigma^* \; : \; x \prec y \Leftrightarrow \mu(x) \prec \mu(y)$$

3. Le langage $\mu(\mathfrak{clone})$ est rationnel : en effet $\mathfrak{clone}(x)$ est vrai si, et seulement si, $\mu(x)$ se termine par bab ou cab.

$$\begin{aligned} \mathfrak{clone}(x) &\Leftrightarrow d(x) \text{ se termine par un zéro} \\ &\Leftrightarrow t(d(x)) \text{ se termine par } bab \text{ ou } cab \\ &\Leftrightarrow \mu(x) \in \mu(\Sigma^*) \cap \sigma_3^*\{bab, cab\} \end{aligned}$$

4. Le langage $\mu(\mathfrak{less})$ est rationnel : en effet $\mathfrak{less}(x)$ est vrai si, et seulement si, $\mu(x)$ se termine par c.

$$\begin{aligned} \mathfrak{less}(x) &\Leftrightarrow d(x) \text{ se termine par un entier négatif} \\ &\Leftrightarrow t(d(x)) \text{ se termine par } c \\ &\Leftrightarrow \mu(x) \in \mu(\Sigma^*) \cap \sigma_3^*\{c\} \end{aligned}$$

5. Pour $p, q \in \mathbb{N}$, $\mu(\mathfrak{mod}_{p,q})$ est rationnel.

$$\begin{aligned} \mathfrak{mod}_{p,q}(x) &\Leftrightarrow d_n \equiv q[p] \\ &\Leftrightarrow t(d(x)) \text{ se termine par un mot de} \\ & \left\{ \begin{array}{l} \{b\}\{a^p\}^*\{a^q ab\} \text{ ou } \{c\}\{a^p\}^*\{a^q ab\} \text{ pour } d_n \geq 0 \\ \{b\}\{a^p\}^*\{a^{p-q} c\} \text{ ou } \{c\}\{a^p\}^*\{a^{p-q} c\} \text{ pour } d_n < 0 \end{array} \right. \\ &\Leftrightarrow \mu(x) \in \mu(\Sigma^*) \cap \sigma_3^*\{b, c\}\{a^p\}^*\{a^q ab, a^{p-q} c\} \end{aligned}$$

□

Les théorèmes 3 et 5 impliquent le résultat suivant.

Corollaire 3. *La théorie du premier ordre de la structure du clone* $\mathrm{FO}(\mathfrak{S}_o)$ *est décidable.*

2.1.2 Théorie monadique du second ordre de la structure de clone

Nous montrons dans cette section que la théorie $\mathrm{MSO}(\mathfrak{S}_o)$ est décidable par réduction à MSO(S3S), qui est décidable d'après [Rab69]. On note $\mathrm{S3S} = (\sigma_3^*; \prec, S_a, S_b, S_c)$ la *structure des trois successeurs* de Rabin telle que :

1. $\sigma_3 = \{a, b, c\}$ est un alphabet ternaire.
2. Pour des mots w_1 et w_2 de σ_3^*, on a $w_1 \prec w_2$ si, et seulement si, w_1 est un préfixe strict de w_2.
3. Pour un mot w de σ_3^*, nous avons $S_a(w) = wa$, $S_b(w) = wb$ et $S_c(w) = wc$.

Théorème 6. *La structure* $\mathfrak{S}_o$ *est* MSO-*interprétable dans la structure* S3S.

Démonstration. On reprend le codage μ défini dans la section précédente. On a montré dans cette section que $\mu(\Sigma^*)$, $\mu(\prec)$, $\mu(\mathfrak{clone})$, $\mu(\mathfrak{less})$ et $\mu(\mathfrak{mod}_{p,q\in\mathbb{N}})$ sont des langages rationnels, donc définissables dans la théorie monadique du second ordre de S3S. Ceci est vrai car tout langage rationnel sur l'alphabet ternaire est définissable dans la théorie monadique du second ordre des trois successeurs (voir [Tho97]). □

Une conséquence directe de l'interprétation de la structure $\mathfrak{S}_o$ dans S3S [Rab69] est la décidabilité de sa théorie logique.

Corollaire 4. *La théorie monadique du second ordre de la structure du clone est décidable.*

Afin d'exprimer plus de propriétés sur les éléments de Σ^*, nous introduisons une logique décidable étendue avec des prédicats plus expressifs.

2.2 Structure étendue du clone

Nous définissons dans cette section la structure étendue du clone et nous montrons la décidabilité de sa théorie du premier ordre.

Définition 16. Nous appelons *structure étendue de clone* la structure

$$\mathfrak{S}_1 = (\Sigma^*; \prec, \mathfrak{clone}, \mathfrak{less}, \{\mathfrak{mod}_{p,q} \mid p, q \in \mathbb{N}\}, \sim, \oplus)$$

telle que :

1. Σ^* désigne l'ensemble des mots finis sur l'alphabet infini $\Sigma = \mathbb{Z}$.
2. Pour des mots x et y de Σ^*, nous avons $x \sim y$ si, et seulement si, x et y ont la même longueur, c'est-à-dire $x \sim y \Leftrightarrow |x| = |y|$.
3. Pour des mots x, y et $z \in \Sigma^*$, le prédicat $\oplus(x, y, z)$ est vrai si, et seulement si, x, y et z ont la même longueur et la somme de x et y, lettre par lettre, est égal à z.

$$\oplus(x, y, z) \Leftrightarrow \begin{cases} x = x_1 x_2 \dots x_i \dots x_n \text{ et} \\ y = y_1 y_2 \dots y_i \dots y_n \text{ et} \\ z = (x_1 + y_1)(x_2 + y_2) \dots (x_i + y_i) \dots (x_n + y_n). \end{cases}$$

Exemple 8. Soient $w, x, y, z \in \Sigma^*$ avec $w = [-1, 5, 2]$, $x = [1, 3, 0]$, $y = [0, 8, 2]$ et $z = [6, 0, 9]$. Nous pouvons vérifier que $\oplus(w, x, y)$ est vrai mais pas $\oplus(x, y, z)$.

2.2.1 Théorie du premier ordre de la structure étendue du clone

Dans ce qui suit, nous montrons la $\mathfrak{M}$-automaticité de la structure étendue de clone pour $\mathfrak{M} = (\mathbb{Z}; 0, <, +)$, d'où la décidabilité de sa théorie logique du premier ordre.

Théorème 7. *La structure étendue de clone $\mathfrak{S}_1$ est $\mathfrak{M}$-automatique.*

Démonstration. Pour prouver que la structure étendue de clone est $\mathfrak{M}$-automatique, nous utilisons le codage d. Ce codage retourne le différentiel de chaque mot de Σ^*. On note $\top$ la formule "vrai". Dans ce qui suit l'état S_0 représentante l'état initial de chaque automate. On a :

1. L'image $d(\Sigma^*)$ du domaine de $\mathfrak{S}_1$ est $\mathfrak{M}$-reconnaissable : en effet $\mathcal{A}_{d(\Sigma^*)}$ est un $\mathfrak{M}$-automate qui reconnaît $d(\Sigma^*)$.

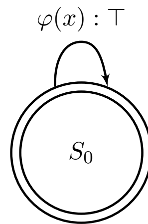


FIGURE 2.1 – L'automate $\mathcal{A}_{d(\Sigma^*)}$.

2. Le langage $d(\prec)$ est $\mathfrak{M}$-reconnaissable : il est facile de voir qu'un mot est préfixe d'un autre si, et seulement si, le code du premier mot est préfixe du code du second. Pour $u, v \in \Sigma^*$, nous avons $u \prec v \Leftrightarrow d(u) \prec d(v)$. Le $\mathfrak{M}$-automate $\mathcal{A}_{d(\prec)}$ est en mesure de vérifier si un mot est le préfixe d'un autre.

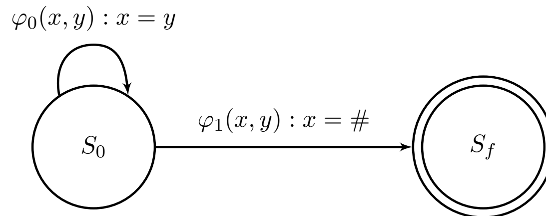


FIGURE 2.2 – L'automate $\mathcal{A}_{d(\prec)}$.

3. Le langage $d(\sim)$ est $\mathfrak{M}$-reconnaissable : il est également aisé de vérifier que deux mots ont la même longueur si, et seulement si, le code du premier et du second ont la même longueur. Pour $u, v \in \Sigma^*$, nous avons $u \sim v \Leftrightarrow d(u) \sim d(v)$. Le $\mathfrak{M}$-automate $\mathcal{A}_{d(\sim)}$ est capable de vérifier si deux mots ont la même longueur.

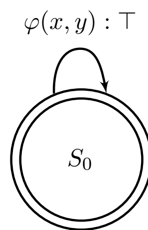


FIGURE 2.3 – L'automate $\mathcal{A}_{d(\sim)}$.

4. Le langage $d(\oplus)$ est $\mathfrak{M}$-reconnaissable : en effet $\mathcal{A}_{d(\oplus)}$ est un $\mathfrak{M}$-automate en mesure de vérifier que la somme lettre par lettre est correcte. Pour $u, v, w \in \Sigma^*$, nous avons $\oplus(u, v, w) \Leftrightarrow \oplus(d(u), d(v), d(w))$.

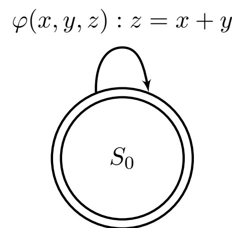


FIGURE 2.4 – L'automate $\mathcal{A}_{d(\oplus)}$.

5. Le langage $d(\mathfrak{clone})$ est $\mathfrak{M}$-reconnaissable : la propriété d'être clone peut être vérifiée par le $\mathfrak{M}$-automate $\mathcal{A}_{d(\mathfrak{clone})}$.

$$\mathfrak{clone}(u) \Leftrightarrow d(u) \text{ se termine par } 0$$

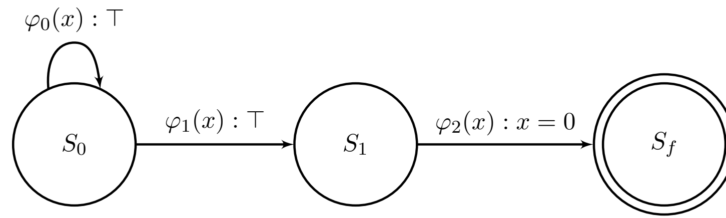


FIGURE 2.5 – L'automate $\mathcal{A}_{d(\mathfrak{clone})}$.

6. Le langage $d(\mathfrak{less})$ est $\mathfrak{M}$-reconnaissable : en effet $\mathcal{A}_{d(\mathfrak{less})}$ est un $\mathfrak{M}$-automate qui vérifie si le code d'un mot se termine par un entier négatif.

$$\mathfrak{less}(u) \Leftrightarrow d(u) \text{ se termine par } a \text{ telle que } a \in \Sigma \text{ et } a < 0$$

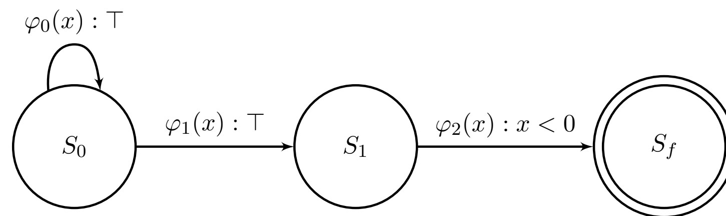


FIGURE 2.6 – L'automate $\mathcal{A}_{d(\mathfrak{less})}$.

7. Pour $p, q \in \mathbb{N}$, le langage $d(\mathfrak{mod}_{p,q})$ est $\mathfrak{M}$-reconnaissable : le $\mathfrak{M}$-automate suivant est capable de vérifier le prédicat $\mathfrak{mod}_{p,q}$.

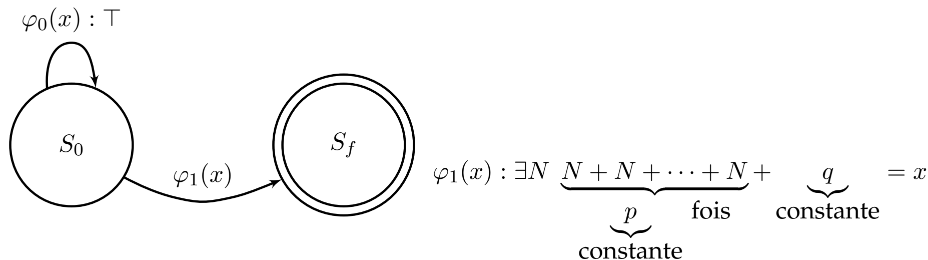


FIGURE 2.7 – L'automate $\mathcal{A}_{d(\mathfrak{mod}_{p,q})}$.

□

Les théorèmes 4, 7 et la décidabilité de FO($\mathfrak{M}$) [Pre30, PJ91] impliquent le résultat suivant.

Corollaire 5. *La théorie du premier ordre de la structure étendue de clone est décidable.*

2.3 Structure de *differentia*

Pour un alphabet infini dénombrable Σ, et afin d'exprimer des propriétés plus naturelles sur les éléments de Σ^*, nous introduisons la structure de *differentia* et nous montrons son automaticité et la décidabilité de sa théorie. Pour mieux comprendre la preuve, des exemples pour $\Sigma = \mathbb{N}$ sont donnés au fur et à mesure.

Définition 17. On note $\mathfrak{S}_4 = (\Sigma^*; \prec, \mathfrak{clone}, \mathfrak{diff})$ la *structure de differentia* telle que :

1. Σ désigne un ensemble infini dénombrable.
2. Pour un mot x de Σ^*, le prédicat $\mathfrak{diff}(x)$ est vrai si, et seulement si, toutes les lettres de x sont distinctes $\mathfrak{diff}(x) \Leftrightarrow x = x_1 x_2 \dots x_n$ avec $x_1, x_2, \dots, x_n \in \Sigma$ et $\forall i \forall j \; i \neq j \Rightarrow x_i \neq x_j$.

Exemple 9. Soient $x, y \in \Sigma^*$ les mots $x = [1, 5, 2]$ et $y = [1, 5, 2, 2, 3, 4, 4]$. Nous pouvons vérifier que $x \prec y$, $\mathfrak{clone}(y)$ et $\mathfrak{diff}(x)$ sont vrais, mais $y \prec x$, $\mathfrak{clone}(x)$ et $\mathfrak{diff}(y)$ sont faux.

2.3.1 Théorie du premier ordre de la structure de *differentia*

Dans ce qui suit, nous montrons l'automaticité de la structure de *differentia* et, par conséquent, la décidabilité de sa théorie logique du premier ordre.

Théorème 8. *La structure de* differentia, $\mathfrak{S}_4 = (\Sigma^*; \prec, \mathfrak{clone}, \mathfrak{diff})$, *est automatique.*

Pour prouver que la structure de *differentia* est automatique, nous allons définir un codage ν qui établit la rationalité du domaine de la structure, du préfixe, du $\mathfrak{clone}$ et du prédicat *differentia*. Avant de détailler la preuve, nous allons montrer comment nous pouvons coder n'importe quel mot sur les entiers naturels dont les lettres sont distinctes, par un mot sur les entiers naturels.

Définition 18. Pour un ensemble ordonné $S = \{x_1 < x_2 < \dots < x_i < \dots\} \subseteq \Sigma$, on note $I_S^{x_i}$ l'indice de l'élément x_i dans S. Nous avons $I_S^{x_i} = i$.

Définition 19. Soit $D = \{x \in \Sigma^* \mid \mathfrak{diff}(x)\}$. Soit $\delta : D \longrightarrow (\mathbb{N} \setminus \{0\})^*$ la bijection qui fait correspondre chaque mot de D à un mot sur les entiers naturels dans $(\Sigma \setminus \{0\})^*$ telle que

$$
\begin{array}{rcl}
\delta : D & \longrightarrow & (\mathbb{N} \setminus \{0\})^* \\
x = x_1 x_2 \dots x_i \dots x_n & \longmapsto & \delta(x) = y_1 y_2 \dots y_i \dots y_n \quad \text{avec} \\
y_1 & = & I_{\Sigma}^{x_1} \\
y_2 & = & I_{\Sigma \setminus \{x_1\}}^{x_2} \\
 & \vdots & \\
y_i & = & I_{\Sigma \setminus \{x_1, x_2, \dots, x_{i-1}\}}^{x_i} \\
 & \vdots & \\
y_n & = & I_{\Sigma \setminus \{x_1, x_2, \dots, x_i, \dots, x_{n-1}\}}^{x_n}
\end{array}
$$

$$\begin{array}{rcl}
\delta^{-1} : (\mathbb{N} \setminus \{0\})^* & \longrightarrow & D \\
y = y_1 y_2 \dots y_i \dots y_n & \longmapsto & \delta^{-1}(y) = x_1 x_2 \dots x_i \dots x_n \quad \text{avec} \\
x_1 & = & \text{le } {y_1}^{\text{ième}} \text{ élément dans } \Sigma \\
x_2 & = & \text{le } {y_2}^{\text{ième}} \text{ élément dans } \Sigma \setminus \{x_1\} \\
& \vdots & \\
x_i & = & \text{le } {y_i}^{\text{ième}} \text{ élément dans } \Sigma \setminus \{x_1, x_2, \dots, x_{i-1}\} \\
& \vdots & \\
x_n & = & \text{le } {y_n}^{\text{ième}} \text{ élément dans } \Sigma \setminus \{x_1, x_2, \dots, x_i, \dots, x_{n-1}\}
\end{array}$$

Par convention $\delta(\varepsilon) = \varepsilon$.

Étant donné un mot dont les lettres sont distinctes, on code la première lettre par l'indice de cette lettre dans Σ, la deuxième lettre par l'indice de cette lettre dans Σ moins la valeur de la première lettre qui a été déjà codée. La $i^{ème}$ lettre est codée par l'indice de cette lettre dans Σ moins toutes les lettres qui la précédent. Ce mécanisme fonctionne parce que nous ne rencontrons pas deux fois la même lettre.

Exemple 10. Soit $x = [5, 3, 4, 2, 1]$ un mot sur les entiers naturels dont les lettres sont distinctes. Nous pouvons vérifier que :

$$\begin{array}{rcl}
\delta(x) & = & \delta([5, 3, 4, 2, 1]) \\
& = & [I^5_{\{0,1,2,3,4,5,6,\dots\}}, I^3_{\{0,1,2,3,4,6,\dots\}}, I^4_{\{0,1,2,4,6,\dots\}}, I^2_{\{0,1,2,3,6,\dots\}}, I^1_{\{0,1,3,6,\dots\}}] \\
& = & [6, 4, 4, 3, 2]
\end{array}$$

Inversement, nous pouvons obtenir x depuis $[6, 4, 4, 3, 2]$:

1. le $6^{\text{ième}}$ élément dans $\Sigma = \{0, 1, 2, 3, 4, 5, 6, \dots\}$ est $5 = x_1$,
2. le $4^{\text{ième}}$ élément dans $\Sigma \setminus \{x_1\} = \{0, 1, 2, 3, 4, 6, \dots\}$ est $3 = x_2$,
3. le $4^{\text{ième}}$ élément dans $\Sigma \setminus \{x_1, x_2\} = \{0, 1, 2, 4, 6, \dots\}$ est $4 = x_3$,
4. le $3^{\text{ième}}$ élément dans $\Sigma \setminus \{x_1, x_2, x_3\} = \{0, 1, 2, 6, \dots\}$ est $2 = x_4$,
5. le $2^{\text{ième}}$ élément dans $\Sigma \setminus \{x_1, x_2, x_3, x_4\} = \{0, 1, 6, \dots\}$ est $1 = x_5$.

Ainsi $\delta^{-1}([6, 4, 4, 3, 2]) = [5, 3, 4, 2, 1]$.

Nous allons montrer maintenant comment nous procédons pour coder un mot quelconque sur les entiers naturels. Tout d'abord, on note que tout mot sur les entiers naturels peut être "divisé" en un nombre minimal de mots de D dits sous-mots. En avançant dans le mot donné, sa première lettre fera partie du premier sous-mot, la lettre suivante est attribuée au sous-mot courant si cette lettre n'est pas dans ce sous-mot courant. On continue ainsi jusqu'à ce qu'on rencontre une lettre qui est déjà dans le sous-mot courant, ce sous-mot en cours sera achevé, et cette lettre sera la première lettre du prochain sous-mot. Et ainsi de suite jusqu'à la fin du mot.

Exemple 11. Voici un exemple.

$$[3,0,6,0,1,2,5,2,4,5,5,5,3,4,8,9] \xrightarrow{s} [3,0,6][0,1,2,5][2,4,5][5][5,3,4,8,9]$$
$$[5,2,6,8,3,6,4,8,5,4,4,4,5,6,8,5] \xrightarrow{s} [5,2,6,8,3][6,4,8,5][4][4][4,5,6,8][5]$$
$$[1,3,5,7,8,0,9,6,4,2] \xrightarrow{s} [1,3,5,7,8,0,9,6,4,2] \text{ un seul sous-mot}$$

Définition 20. On note s la fonction qui *divise* un mot sur les entiers naturels en une séquence de sous-mots de D :

$$\begin{array}{rcl} s : \Sigma^* & \longrightarrow & D^* \\ x = x_1x_2\dots x_n & \longmapsto & s(x) = u_1u_2\dots u_m \end{array}$$

telle que :

1. pour tout i entre 1 et m, $\mathfrak{diff}(u_i)$ est vérifié,
2. pour tout i entre 2 et m, la première lettre de u_i apparaît déjà dans u_{i-1}.

L'algorithme suivant permet d'obtenir cette division :

Algorithme 1 : Division

```
Données : x = x1 x2 ... xn ∈ Σ*                      1
Résultat : s(x) = u1 u2 ... um ∈ D*                  2
m ← 1                                                3
pour j ← 1 à n faire                                 4
    si xj ∉ um alors                                 5
        um ← um · xj                                 6
    fin                                              7
    sinon                                            8
        m++                                          9
        um ← xj                                      10
    fin                                              11
fin                                                  12
```

Il est clair que :

– $m \leq n$.
– s coïncide avec l'identité pour un élément de D : $x \in D \Leftrightarrow s(x) = x$.

Notre codage final, appelé $\nu : \Sigma^* \longrightarrow \sigma_3 = \{a,b,c\}^*$, est obtenu en plusieurs étapes. La première consiste à diviser le mot donné en sous-mots de D en utilisant la fonction s. La deuxième étape consiste à coder ces sous-mots en utilisant la fonction δ. Le résultat est une suite de mots sans zéro, donc nous utilisons le symbole zéro, afin de séparer les sous-mots. Il est important de noter que si un sous-mot v suit un sous-mot u, le second sous-mot v doit commencer par une lettre qui se trouve dans le sous-mot u ; on marquera alors cette lettre par le symbole du signe négatif dans le codage de u et par zéro dans le codage de v. Le résultat est un mot dont les lettres sont marquées par le symbole du signe positif ou négatif [1]. Enfin, on code chaque lettre du résultat par un certain nombre de a suivi par b ou c en fonction du signe de cette lettre : la lettre $+n$ est codée par a^nb et la lettre $-n$ est codée par a^nc. Ainsi $+0$ et -0, codés respectivement par b et c, sont deux lettres distinctes.

1. Même le zéro peut être marqué par un symbole de signe positif ou négatif.

Exemple 12. Voici un exemple pour le codage du mot

$$[5,2,6,8,3,6,4,8,5,4,4,4,5,6,8,5]$$

$$\begin{array}{ll}
\stackrel{s}{\to} & [5,2,6,8,3][6,4,8,5][4][4][4,5,6,8][5] \\
\stackrel{\delta}{\to} & [6,3,5,6,3][7,5,7,5][5][5][5,5,5,6][6] \\
\stackrel{-/+}{\to} & [+6,+3,-5,+6,+3][+7,-5,+7,+5][-5][-5][+5,-5,+5,+6][+6] \\
\stackrel{séparateur}{\to} & [+6,+3,-5,+6,+3,+0,-5,+7,+5,-0,-0,+0,-5,+5,+6,+0] \\
\stackrel{\sigma_3=\{a,b,c\}}{\to} & [a^6ba^3ba^5ca^6ba^3bba^5ca^7ba^5bccba^5ca^5ba^6bb]
\end{array}$$

Exemple 13. Voici un autre exemple, pour le codage du mot

$$[1,3,5,7,8,0,9,6,4,2]$$

$$\begin{array}{ll}
\stackrel{s}{\to} & [1,3,5,7,8,0,9,6,4,2] \\
\stackrel{\delta}{\to} & [2,3,4,5,5,1,4,3,2,1] \\
\stackrel{-/+}{\to} & [+2,+3,+4,+5,+5,+1,+4,+3,+2,+1] \\
\stackrel{séparateur}{\to} & [+2,+3,+4,+5,+5,+1,+4,+3,+2,+1] \\
\stackrel{\sigma_3=\{a,b,c\}}{\to} & [a^2ba^3ba^4ba^5ba^5ba^1ba^4ba^3ba^2ba^1b]
\end{array}$$

Lemme 4. *Les langages $\nu(\Sigma^*)$, $\nu(\prec)$, $\nu(\mathfrak{clone})$ et $\nu(\mathfrak{diff})$ sont rationnels.*

Démonstration. L'utilisation du codage ν permet d'obtenir la rationalité du domaine de notre structure ainsi que ses relations :

1. Nous pouvons vérifier qu'un mot sur l'alphabet $\sigma_3 = \{a,b,c\}$ correspond à l'image d'un élément de Σ^* par ν si, et seulement si, ce mot appartient au langage rationnel suivant :

$$\nu(\Sigma^*) = P^*NP^*(\{b\}P^*NP^* \cup \{c\}P^*)^*\{b\}P^* \text{ tel que } P = \{a\}^+\{b\} \text{ et } N = \{a\}^+\{c\}$$

P code ce qui est marqué par un symbole de signe **p**ositif, et N code ce qui est marqué par un symbole de signe **n**égatif. Étant donné un mot x de Σ^*, si x appartient à D alors $\nu(x)$ appartient à P^*. Si maintenant x n'appartient pas à D alors $\nu(x)$ est une suite de sous-mots qui ont la propriété suivante : chaque sous-mot x_i, sauf le dernier, se termine par un zéro positif ou négatif, et le reste du sous-mot x_i est une suite de nombres tous marqués par un signe positif sauf un seul nombre marqué par un signe négatif. Le dernier sous-mot se termine par un zéro positif. On a ainsi :

$$\nu(\Sigma^*) = \overbrace{P^* \underbrace{N}_{\text{strictement négatif}} P^*}^{\text{le premier sous-mot}} \overbrace{(\{\underbrace{b}_{+0}\}P^*NP^* \cup \{\underbrace{c}_{-0}\}P^*)^*}^{\text{la suite de sous-mots}} \overbrace{\{b\}P^*}^{\text{le dernier sous-mot}} .$$

2. Remarquons dans un premier temps que si on ne prend pas en considération les marques de signe, le codage ν respecte la propriété "être préfixe". Plus précisément pour $x, y \in \Sigma^*$, on a $x \prec y$ si, et seulement si, $\delta(s(x)) \prec \delta(s(y))$ (δ et s étant les fonctions intermédiaires utilisées dans la définition de ν). D'autre part, si $x \prec y$ alors sur $\nu(x)$ et $\nu(y)$ les marques sont exactement les mêmes sauf pour un seul élément parmi les derniers éléments marqués positivement dans $\nu(x)$. Cet élément est positif dans $\nu(x)$ mais peut être négatif dans $\nu(y)$ selon que cet élément va réapparaître ou non plus loin dans le mot y. Ceci conduit au langage rationnel suivant pour $\nu(\prec)$:

$$\left\{\binom{a}{a},\binom{b}{b},\binom{c}{c}\right\}^*\left\{\binom{b}{b},\binom{c}{b}\right\}\left(\left\{\binom{a}{a}\right\}^*\left\{\binom{b}{b}\right\}\right)^*\left\{\binom{a}{\#},\binom{b}{\#},\binom{c}{\#}\right\}^*.$$

3. Le langage $\nu(\mathfrak{clone})$ est rationnel : en effet $\mathfrak{clone}(x)$ est vrai si, et seulement si, $\nu(x)$ se termine par cb. On a :

$$\mathfrak{clone}(x) \Leftrightarrow \nu(x) \in \nu(\Sigma^*) \cap {\sigma_3}^*\{cb\}$$

4. Le langage $\nu(\mathfrak{diff})$ est rationnel : en effet $\mathfrak{diff}(x)$ est vrai si, et seulement si, $\nu(x)$ est une suite de codage de nombres positifs. Ainsi :

$$\nu(\mathfrak{diff}) = \nu(D) = P^* = (\{a\}^+\{b\})^*.$$

$\square$

Les théorèmes 3 et 8 impliquent le résultat suivant.

Corollaire 6. *La théorie du premier ordre de la structure de* differentia *est décidable.*

2.3.2 Théorie monadique du second ordre de la structure de *differentia*

Nous montrons dans cette section la décidabilité de la théorie monadique du second ordre de la structure de *differentia*. Nous allons prouver que la théorie $\mathrm{MSO}(\mathfrak{S}_4)$ est décidable par réduction à MSO(S3S), qui est décidable d'après [Rab69].

Théorème 9. *La structure* $\mathfrak{S}_4$ *est* MSO-*interprétable dans la structure* S3S.

Démonstration. On reprend le codage ν défini dans la section précédente. On a montré dans cette section que $\nu(\Sigma^*)$, $\nu(\prec)$, $\nu(\mathfrak{clone})$ et $\nu(\mathfrak{diff})$ sont des langages rationnels, donc définissables dans la théorie monadique du second ordre de S3S. Ceci est vrai car tout langage rationnel sur l'alphabet ternaire est définissable dans la théorie monadique du second ordre des trois successeurs (voir [Tho97]). $\square$

Une conséquence directe de l'interprétation de la structure $\mathfrak{S}_4$ dans S3S [Rab69] est la décidabilité de sa théorie logique.

Corollaire 7. *La théorie monadique du second ordre de la structure de* differentia *est décidable.*

Remarque 1. Considérons la structure $\mathfrak{S}_5 = (\Sigma^*; \prec, \mathfrak{clone}, \mathfrak{diff}, \sim)$ obtenue à partir de la structure $\mathfrak{S}_4$ en ajoutant le prédicat $\sim$. On peut obtenir la décidabilité de sa théorie du premier ordre à partir de sa $\mathfrak{M}$-automaticité et de la décidabilité de la théorie du premier ordre de la structure $(\mathbb{Z}; 0, <)$. En effet La structure $\mathfrak{S}_5$ est $\mathfrak{M}$-automatique pour $\mathfrak{M} = (\mathbb{Z}; 0, <)$ selon un codage ν' similaire au codage ν. Le codage $\nu' : \Sigma^* \longrightarrow \{0, 1, 2, \ldots\}^*$ est obtenu comme ν en plusieurs étapes. La première consiste à diviser le mot donné en sous-mots de D en utilisant la fonction s. La deuxième étape consiste à coder ces sous-mots en utilisant la fonction δ. Le résultat est une suite de mots sans zéro, donc là aussi nous utilisons le symbole zéro, afin de séparer les sous-mots.

2.4 Structure des différences

Pour un alphabet infini dénombrable Σ, et afin d'exprimer des propriétés plus naturelles sur les éléments de Σ^*, nous introduisons *la structure des différences* et nous montrons la décidabilité de ses théories du premier ordre. Pour mieux comprendre la preuve, des exemples pour $\Sigma = \mathbb{N}$ sont donnés au fur et à mesure.

À partir du prédicat $\mathfrak{clone}$ on peut facilement définir le prédicat $\mathfrak{diff}_2$ qui exprime qu'un mot se termine par deux lettres différentes comme suit :

$$\text{Pour } x \in \Sigma^* \ : \ \mathfrak{diff}_2(x) \Leftrightarrow \neg\mathfrak{clone}(x)$$

Cependant, pour $i > 2$, on veut exprimer le prédicat $\mathfrak{diff}_i$ qui exprime qu'un mot se termine par i lettres différentes, ce qui ne semble pas possible d'exprimer dans les structures étudiées précédemment. Ainsi, afin d'étendre la structure du clone, tout en maintenant la décidabilité de sa théorie, nous introduisons la structure des différences :

$$\mathfrak{S}_3 = (\Sigma^*; \prec, \mathfrak{clone}, \{\mathfrak{diff}_i \mid i > 2\}, \sim)$$

caractérisée par la généralisation du prédicat $\mathfrak{clone}$ par la famille des prédicats $\{\mathfrak{diff}_i \mid i > 2\}$.

Nous montrons dans cette section, la $\mathfrak{M}$-automaticité de la structure des différences puis la décidabilité de sa théorie logique du premier ordre.

Définition 21. On note $\mathfrak{S}_3 = (\Sigma^*; \prec, \mathfrak{clone}, \{\mathfrak{diff}_i \mid i > 2\}, \sim)$ la *structure des différences*, telle que :

1. Σ^* désigne un ensemble infini dénombrable,
2. le prédicat $\mathfrak{diff}_i(x)$ est vrai si, et seulement si, x se termine par i lettres distinctes :

$$\mathfrak{diff}_i(x) \Leftrightarrow x = ua_1a_2\ldots a_i \text{ avec } u \in \Sigma^* \text{ et } a_1, a_2, \ldots, a_i \in \Sigma \text{ et } \forall p \forall q \ : \ p \neq q \Rightarrow a_p \neq a_q$$

Exemple 14. Soient x, y des mots de Σ^*.

1. Pour $x = [1, 2, 5, 2, 3]$ et $y = [1, 2, 5, 2, 3, 4, 4]$:
2. $x \prec y$, $\mathfrak{diff}_3(x)$ et $\mathfrak{clone}(y)$ sont vrais.
3. $x \sim y$, $\mathfrak{diff}_4(x)$ et $\mathfrak{diff}_3(y)$ sont faux.

Définition 22. Nous appelons *structure des différences d'ordre* k la structure définie comme suit :

$$\mathfrak{D}_k = (\Sigma^*; \prec, \mathfrak{clone}, \{\mathfrak{diff}_i \mid 2 < i \leq k\}, \sim)$$

Théorème 10. *La structure* $\mathfrak{D}_k$ *est* $\mathfrak{M}$*-automatique avec* $\mathfrak{M} = (\mathbb{Z}; -1, -2, \ldots, -(k-1))$.

Avant de détailler la preuve de ce théorème, nous introduisons un codage μ_k qui établit la $\mathfrak{M}$-automaticité du domaine de la structure et la $\mathfrak{M}$-automaticité des prédicats : clone, même longueur et des différences.

Définition 23. Soit $x = x_0 x_1 \ldots x_{n-1}$ un mot sur Σ^* de longueur n. L'ensemble des k *voisins* de la lettre x_i est l'ensemble des k lettres qui précèdent x_i s'il existe au moins k lettres avant x_i, autrement, s'il y a moins de k lettres avant x_i, l'ensemble des k voisins de la lettre x_i est l'ensemble de toutes les lettres qui précèdent x_i :

$$V_k(x_i) = \bigcup_{j=1}^{\min(i,k)} \{x_{i-j}\} = \bigcup_{j=1}^{\min(i,k)} \{v_j\} \text{ avec } v_j = x_{i-j}$$

Nous appelons $j^{\text{ième}}$ voisin de la lettre x_i et on note v_j la lettre qui correspond à la $j^{\text{ième}}$ lettre qui précède x_i. S'il y a moins de k lettres avant la lettre x_i, le nombre de v_j est égal à k, le cardinal de l'ensemble $V_k(x_i)$, est au plus k, et atteint k lorsque les v_j sont distincts.

Notre codage est décrit comme suit. Pour chaque lettre x_i de x nous regardons l'ensemble des $k-1$ lettres qui précèdent x_i, si x_i est égal à son premier voisin, nous codons x_i par un -1, sinon, si x_i est égal à son deuxième voisin, nous codons x_i par un -2, sinon, si x_i est égal à son troisième voisin, nous codons x_i par un -3, etc. Sinon, si, x_i est différent de tous ses $k-1$ voisins, dans ce cas, x_i est codé par son indice dans l'ensemble $\Sigma \setminus V_{k-1}(x_i)$. Voici la définition formelle du codage.

Définition 24. Le codage μ_k d'un mot $x = x_0 x_1 \ldots x_{n-1}$ est aussi un mot, défini par :

$$\begin{array}{rcl} \mu_k : \Sigma^* & \rightarrow & \Sigma^* \\ \mu_k(x) & = & x_0 y_1 \ldots y_i \ldots y_{n-1} \text{ avec } y_i \in \Sigma \text{ et :} \end{array}$$

$$y_i = \begin{cases} -\min\{j \mid x_i = v_j, 1 \leq j \leq \min(i, k-1)\} & \text{si } x_i \in V_{k-1}(x_i), \\ I^{x_i}_{\Sigma \setminus V_{k-1}(x_i)} & \text{sinon.} \end{cases}$$

Par convention $\mu_k(\varepsilon) = \varepsilon$. Voir la définition 18 pour la fonction I.

Exemple 15. Voici un exemple pour $k = 4$ et un mot $w \in \mathbb{N}$.

$$w = \left[3, 0, 6, 0, 1, 2, 5, 2, 4, 5, 5, \overbrace{5}^{b}, 3, 4, \overbrace{8}^{a}\right] \xrightarrow{\mu_k} [4, 1, 5, -2, 1, 1, 3, -2, 4, -3, -1, -1, 4, 4, 6]$$

Considérons a l'avant dernière lettre de w qui est de valeur 8. Le voisinage de cette lettre est les trois lettres qui la précèdent $\{5, 3, 4\}$. La lettre a est donc codée par l'indice du nombre 8 dans l'ensemble $\mathbb{N} \setminus \{5, 3, 4\}$, donc 6.

Considérons maintenant b la dernière lettre de w de valeur 5. Le voisinage de cette lettre est les trois lettres qui la précèdent dont deux qui sont égaux. Donc le voisinage de b est l'ensemble $\{4, 5\}$. La lettre b est égal à la fois à son premier et à son deuxième voisin. Notre codage code cette lettre par -1 qui veut dire que b fait référence à son première voisin. Nous interdisons que la lettre b soit codée par -2 qui veux dire que b fait référence à son deuxième voisin.

Chaque mot de Σ^* est codé par un unique mot de Σ^* mais pas l'inverse. Ils existent des mots Σ^* qui sont des mots non valides au sens de notre codage. Ces mots interdits sont facilement reconnaissables. Expliquons : dans l'exemple 15 la lettre b est égal à son premier et son deuxième voisin, implicitement son premier voisin est égal à son deuxième voisin. Si le codage a été bien calculé jusqu'au premier voisin de b alors le premier voisin est nécessairement codé par -1. Ceci veut dire qu'une lettre codée par -1 interdit le codage de la lettre suivante par -2. De même une lettre codée par -2 interdit le codage de la lettre suivante par -3. Pour $k = 4$ on peut facilement voir qu'un mot correspond à un code si et seulement s'il ne contient aucun des motifs de facteurs suivants :

1. $[-1, -2]$ ce facteur est interdit car il veut dire qu'une lettre est égal à son premier et à son deuxième voisin mais elle fait référence à son deuxième voisin.

2. $[-2, -3]$ ce facteur est interdit car il veut dire qu'une lettre est égal à son premier et à son troisième voisin mais elle fait référence à son troisième voisin.

3. $[-1, x, -3]$ pour x quelconque, ce motif est interdit car il veut dire qu'une lettre est égal à son deuxième et à son troisième voisin mais elle fait référence à son troisième voisin.

De même pour un k quelconque on peut facilement construire la liste de tous les des motifs interdits. Il est claire que le nombre de motifs interdits est fini, car k est borné. Ainsi on peut construire pour un k donné un $\mathfrak{M}$-automate permettant de reconnaître les mots qui ne contiennent aucun des facteurs interdits.

Démonstration. Pour prouver que la structure des différences d'ordre k est $\mathfrak{M}$-automatique, nous utilisons le codage μ_k. L'image $\mu_k(\Sigma^*)$ du domaine de $\mathfrak{D}_k$ est $\mathfrak{M}$-reconnaissable car, comme expliqué dans l'exemple précédent, k est borné et le nombre de motifs de facteurs inadéquats avec notre codage sont finis et reconnaissable par un $\mathfrak{M}$-automate avec $\mathfrak{M} = (\mathbb{Z}; -1, -2, \ldots, -(k-1))$.

Avec le codage μ_k nous pouvons aussi construire des $\mathfrak{M}$-automates qui peuvent vérifier si un mot est clone, si deux mots ont la même longueur, et encore si les prédicats des différences sont vérifiés ou pas pour un mot donné. On donne par exemple les automates $\mathcal{A}_{\mu(\mathfrak{clone})}$, $\mathcal{A}_{\mu(\mathfrak{diff}_3)}$, $\mathcal{A}_{\mu(\mathfrak{diff}_4)}$ et $\mathcal{A}_{\mu(\mathfrak{diff}_i)}$.

L'automate $\mathcal{A}_{\mu(\mathfrak{clone})}$ vérifie que la dernière lettre est codée par -1 qui veut dire qu'elle est égal à son premier voisin.

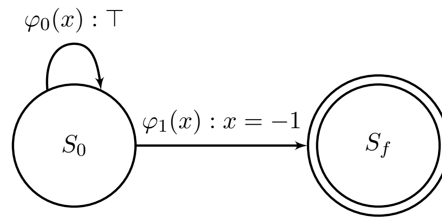


FIGURE 2.8 – L'automate $\mathcal{A}_{\mu(\mathfrak{clone})}$.

L'automate $\mathcal{A}_{\mu(\mathfrak{diff}_3)}$ vérifie que la dernière lettre n'est pas codée par -1 ni par -2 qui veut dire qu'elle est différente de ses deux premiers voisins. Il vérifie aussi que l'avant dernière lettre n'est pas codée par -1 qui veut dire que l'avant dernière lettre est différente de son premier voisin (ce voisin est aussi le deuxième voisin de la dernière lettre). Ces deux vérifications permettent de savoir si le mot se termine par trois lettres différentes.

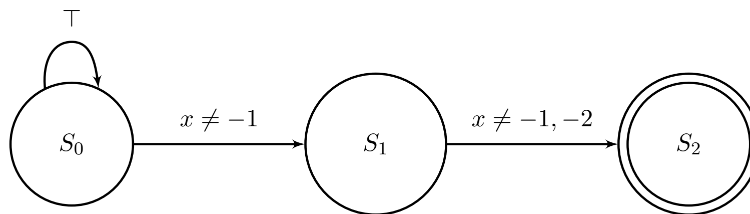


FIGURE 2.9 – L'automate $\mathcal{A}_{\mu(\mathfrak{diff}_3)}$.

L'automate $\mathcal{A}_{\mu(\mathfrak{diff}_4)}$ vérifie que la dernière lettre n'est pas codée par -1 ni par -2 ni par -3 qui veut dire qu'elle est différente de ses trois premiers voisins. Il vérifie aussi que l'avant dernière lettre n'est pas codée par -1 ni par -2 qui veut dire que l'avant dernière lettre est différente de ses deux premiers voisins. Il vérifie aussi que l'avant avant dernière lettre n'est pas codée par -1 qui veut dire que l'avant avant dernière lettre est différente de son premier voisin. Ces trois vérifications permettent de savoir si le mot se termine par quatre lettres différentes.

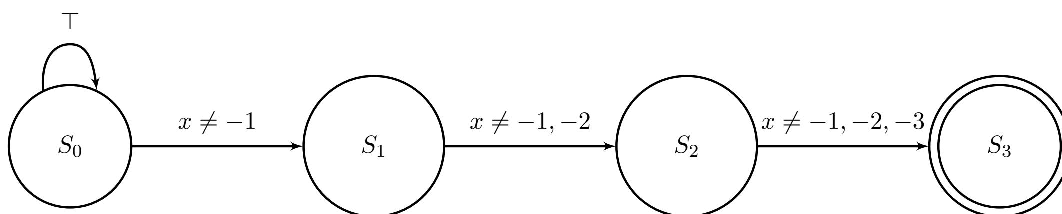


FIGURE 2.10 – L'automate $\mathcal{A}_{\mu(\mathfrak{diff}_4)}$.

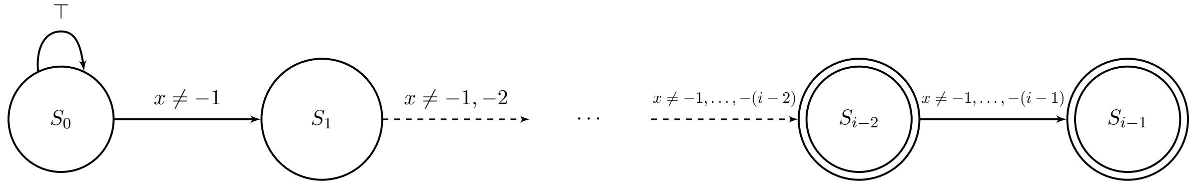


FIGURE 2.11 – L'automate $\mathcal{A}_{\mu(\mathfrak{diff}_i)}$.

□

On a le corollaire suivant.

Corollaire 8. *La théorie du premier ordre de la structure des différences d'ordre k,* $\mathrm{FO}(\mathfrak{D}_k)$ *est décidable.*

Démonstration. On peut montrer sans difficulté que $\mathfrak{M}$ est automatique et donc sa théorie de premier ordre $\mathrm{FO}(\mathfrak{M})$ est décidable. Le théorème 4 et la $\mathfrak{M}$-automaticité de a structure $\mathfrak{D}_k$ démontrée dans le théorème 10 permettent de conclure. □

Corollaire 9. *La théorie du premier ordre de la structure $\mathfrak{S}_3$ est décidable.*

Démonstration. Pour montrer que la théorie de la structure $\mathfrak{S}_3$ est décidable, il suffit de remarquer que tout énoncé φ dans la signature de $\mathfrak{S}_3$ contient un nombre fini d'occurrences de prédicats de différences :

$$\mathfrak{diff}_{i_1}, \mathfrak{diff}_{i_2}, \ldots, \mathfrak{diff}_{i_r}$$

Par conséquent, décider φ dans $\mathfrak{S}_3$ revient à décider φ dans $\mathfrak{D}_k$ avec $k = \max(i_1, i_2, \ldots, i_r)$. Or la théorie du premier ordre de $\mathfrak{D}_k$ est décidable d'après le corollaire 8. □

2.5 Structure d'applications exclusives

On donne dans cette section, un résultat de décidabilité lié à certaines fonctions spécifique que nous appelons applications exclusives.

Soit $J \subseteq \mathbb{N} \setminus \{0\}$ non vide.

Définition 25. Soient $f, g : \Sigma^* \to \Sigma$ des applications. On dit que f et g sont *exclusives* si, et seulement si : $\forall w \in \Sigma^* \ : \ f(w) \neq g(w)$.

Définition 26. Soit $(f_i)_{i \in J}$ de Σ^* dans Σ une famille finie ou infinie d'applications. On dit que $(f_i)_{i \in J}$ est *exclusive* si, et seulement si : $\forall (i \neq j) \ : \ f_i$ et f_j sont exclusives. On définit la famille de *relations exclusives* $(R^{\oplus}_{f_i})_{i \in J}$ comme suit :

$$\text{pour } w \in \Sigma^* \text{ et } a \in \Sigma,\ R^{\oplus}_{f_i}(wa) \text{ est vrai si, et seulement si, } f_i(w) = a.$$

Exemple 16. Dans cette exemple nous allons proposer des applications exclusives définies à partir de l'opération d'addition et de multiplication. Pour $\Sigma = \mathbb{N} \setminus \{0, 1, 2\}$, soient les applications suivantes :

$$\begin{array}{rcl} f_1 = f_+ : \Sigma^* & \longrightarrow & \Sigma \\ \varepsilon & \longmapsto & f_+(\varepsilon) = 3 \\ a & \longmapsto & f_+(a) = 3 \text{ tel que } a \in \Sigma \\ wab & \longmapsto & f_+(wab) = a + b \text{ tel que } w \in \Sigma^* \text{ et } a, b \in \Sigma \end{array}$$

$$\begin{array}{rcl} f_2 = f_\times : \Sigma^* & \longrightarrow & \Sigma \\ \varepsilon & \longmapsto & f_\times(\varepsilon) = 4 \\ a & \longmapsto & f_\times(a) = 4 \text{ tel que } a \in \Sigma \\ wab & \longmapsto & f_\times(wab) = a \times b \text{ tel que } w \in \Sigma^* \text{ et } a, b \in \Sigma \end{array}$$

Pour que les fonctions f_+ et $f_\times$ soit exclusives nous avons exclu les éléments $\{0, 1, 2\}$ car $0 + 0 = 0 \times 0$ et $2 + 2 = 2 \times 2$.

Théorème 11. *Soit Σ un alphabet infini. Soit $(R^{\oplus}_{f_i})_{i \in J}$ une famille infinie de relations exclusives définie à partir de la famille d'applications exclusives $(f_i)_{i \in J} : \Sigma^* \to \Sigma$.*

La structure $\mathfrak{S}_6 = (\Sigma^*; \prec, \{R^{\oplus}_{f_i} \mid i > 0\}, \sim)$ est $(\mathbb{Z}; 0, <)$-automatique.

Avant de détailler la preuve de ce théorème, nous introduisons un codage λ qui établit la $\mathfrak{M}$-automaticité du domaine de la structure et la $\mathfrak{M}$-automaticité des relations exclusives et le prédicat "même longueur".

Définition 27. Pour prouver la $\mathfrak{M}$-automaticité de la structure $\mathfrak{S}_6$ nous passerons par le codage $\lambda : \Sigma^* \longrightarrow (\mathbb{Z})^*$ défini comme suit (le point c'est la concaténation) :

$$\text{pour } a \in \Sigma \text{ et } w \in \Sigma^* : \ \lambda(wa) = \lambda(w) \cdot \begin{cases} -i & \text{si } \exists i \in J \ : \ a = f_i(w), \\ I^a_{\Sigma - \{f_i(w) \mid i \in J\}} & \text{sinon.} \end{cases}$$

Par convention $\lambda(\varepsilon) = \varepsilon$.

Supposons qu'on a utilisé λ pour coder un mot w. Pour coder le mot wa (w concaténé à la lettre a) il faut d'abord coder le mot w ce qui donne $\lambda(w)$. Ensuite on cherche à savoir s'il y a une application exclusive f_i telle que l'application de cette fonction sur w donne a. Si une telle application existe elle sera unique car par définition l'ensemble des applications considérées sont exclusives. Dans ce cas la lettre a sera codée par l'entier $-i$ pour garder cette information. Sinon, si la valeur de a est différente de $f_i(w)$ pour toute application f_i considérée, dans ce cas la lettre a est codée par l'indice de cette lettre dans l'ensemble Σ qu'on lui retranche les valeurs $f_i(w)$ pour toute application f_i considérée.

Exemple 17. Soit $\Sigma = \mathbb{N} \setminus \{0, 1, 2\}$. Soit la structure $\mathfrak{X} = (\Sigma^*; \prec, R_{f_+}, R_{f_\times}, \sim)$.

Pour $x = [8, 9, 20, 29, 5, 6, 30, 14]$ un mot de Σ^*, on a $\lambda(x) = [4, 5, 17, -1, 3, 4, -2, 12]$.

Notons que la suite $[3, 5, 8, 13, \ldots, F_i]$, toute suite finie de Fibonacci qui commence par l'élément 3 est définissable dans la structure $\mathfrak{X} = (\Sigma^*; \prec, R_{f_+}, R_{f_\times}, \sim)$. Il suffit d'écrire la formule qui exprime qu'un mot x de Σ^* commence par $[3, 5]$ est que la relation R_{f_+} est vraie pour tout préfixe de x de longueur supérieur strictement à 2.

Démonstration. Pour prouver que la structure des applications exclusives est $\mathfrak{M}$-automatique, nous utilisons le codage λ. Sous ce codage λ, on a :

1. L'image $\lambda(\Sigma^*)$ du domaine de $\mathfrak{S}_6$ est $\mathfrak{M}$-reconnaissable : en effet $\mathcal{A}_{\lambda(\Sigma^*)}$ est un $\mathfrak{M}$-automate qui reconnaît $\lambda(\Sigma^*)$.

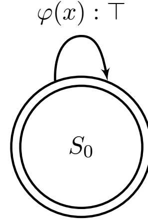


FIGURE 2.12 – L'automate $\mathcal{A}_{\lambda(\Sigma^*)}$.

2. Le langage $\lambda(\prec)$ est $\mathfrak{M}$-reconnaissable : il est facile de voir qu'un mot est préfixe d'un autre si, et seulement si, le code du premier mot est préfixe du code du second. Pour $u, v \in \Sigma^*$, nous avons $u \prec v \Leftrightarrow \lambda(u) \prec \lambda(v)$. Le $\mathfrak{M}$-automate $\mathcal{A}_{\lambda(\prec)}$ est en mesure de vérifier si un mot est le préfixe d'un autre.

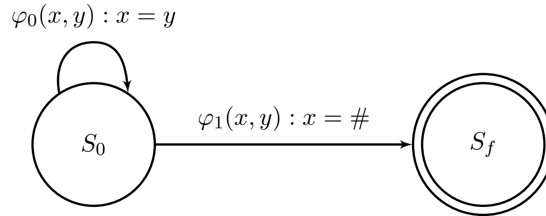


FIGURE 2.13 – L'automate $\mathcal{A}_{\lambda(\prec)}$.

3. Le langage $\lambda(\sim)$ est $\mathfrak{M}$-reconnaissable : il est également aisé de vérifier que des mots ont la même longueur si, et seulement si, le code du premier et du second ont la même longueur. Pour $u, v \in \Sigma^*$, nous avons $u \sim v \Leftrightarrow \lambda(u) \sim \lambda(v)$. Le $\mathfrak{M}$-automate $\mathcal{A}_{\lambda(\sim)}$ est capable de vérifier si deux mots ont la même longueur.

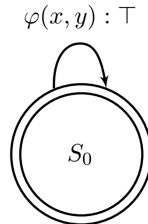


FIGURE 2.14 – L'automate $\mathcal{A}_{\lambda(\sim)}$.

4. Le langage $\lambda(R^{\oplus}_{f_j})$ est $\mathfrak{M}$-reconnaissable : chaque relation $R^{\oplus}_{f_j}$ peut être vérifiée par le $\mathfrak{M}$-automate $\mathcal{A}_{\lambda(R^{\oplus}_{f_j})}$.

$$R^{\oplus}_{f_j}(u) \Leftrightarrow \lambda(u) \text{ se termine par l'entier négatif } -j$$

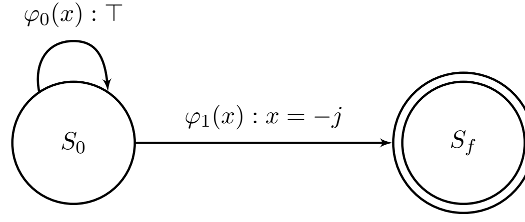


FIGURE 2.15 – L'automate $\mathcal{A}_{\lambda(R^{\oplus}_{f_j})}$.

□

Les théorèmes 4 et 11 impliquent le résultat suivant.

Corollaire 10. *La théorie du premier ordre de la structure* $\mathfrak{S}_6 = (\Sigma^*; \prec, \{R^{\oplus}_{f_i} \mid i > 0\}, \sim)$ *est décidable.*

2.6 Structure de décomposition

Nous définissons dans cette section la structure de décomposition, nous montrons son automaticité et, par conséquent, la décidabilité de sa théorie du premier ordre. On montre son universalité et aussi l'indécidabilité de sa théorie monadique du second ordre.

Soit Σ un alphabet infini dénombrable muni d'une relation d'ordre $<$ et d'une opération binaire interne $\star$. On note :

$$\star_{j=i}^{n} x_j = (\cdots((x_i \star x_{i+1}) \star x_{i+2}) \star \cdots) \star x_n)$$

Définition 28. Une *structure de décomposition* est un quadruplet $\mathfrak{S}_7 = (\Sigma^*; \prec, <, \mathfrak{dec}_\star)$ tel que pour des mots x et y de Σ^*, on dit que *x décompose y relativement à $\star$* et on note $\mathfrak{dec}_\star(x, y)$ si, et seulement si :

$$\begin{cases} |x| \leq |y| \text{ et} \\ x = x_1 \dots x_n \text{ et} \\ y = y_1 \dots y_n \dots y_m \text{ et} \\ \text{il existe } d_0, d_1, \dots, d_n \in \mathbb{N} \text{ avec } 1 = d_0 < d_1 < \cdots < d_n \leq m \text{ tels que :} \\ x_i = \star_{j=d_{i-1}}^{d_i - 1} y_j \text{ pour } 1 \leq i \leq n \end{cases}$$

Exemple 18. La somme est une opération binaire interne sur $\Sigma = \mathbb{N}$. On a $\mathfrak{dec}_+(x, y)$ si, et seulement si :

$$x_i = \sum_{j=d_{i-1}}^{d_i - 1} y_j \text{ pour } 1 \leq i \leq n$$

Soient $x, y, z \in \Sigma^*$ des mots avec $x = [5, 7, 4]$, $y = [2, 3, 7, 2, 1, 1, 2, 4, 1]$ et $z = [16, 7]$. Nous pouvons vérifier que $\mathfrak{dec}_+(x, y)$ et $\mathfrak{dec}_+(z, y)$ sont vrais :

$$[\underbrace{2+3}_{5}, \underbrace{7}_{7}, \underbrace{2+1+1}_{4}, 2, 4, 1]$$

$$[\underbrace{2+3+7+2+1+1}_{16}, \underbrace{2+4+1}_{7}]$$

Exemple 19. Le produit est une opération binaire interne sur $\Sigma = \mathbb{Q}$. On a $\mathfrak{Dec}_\times(x, y)$ si, et seulement si :

$$x_i = \prod_{j=d_{i-1}}^{d_i - 1} y_j \text{ pour } 1 \leq i \leq n$$

Soient $x, y \in \Sigma^*$ les mots $x = [\frac{35}{9}, 20]$ et $y = [\frac{7}{4}, \frac{10}{9}, 2, \frac{16}{5}, \frac{25}{4}, \frac{17}{43}]$. Nous pouvons vérifier que $\mathfrak{Dec}_\times(x, y)$ est vrai :

$$[\underbrace{\frac{7}{4} \times \frac{10}{9} \times 2}_{\frac{35}{9}}, \underbrace{\frac{16}{5} \times \frac{25}{4}}_{20}, \frac{17}{43}]$$

Dans ce qui suit on donnera un théorème qui énonce un critère lié à l'opération interne définissant la structure de décomposition pour aboutir à l'universalité de cette dernière et, par conséquent :

1. l'automaticité de la structure,
2. la décidabilité de sa théorie logique du premier ordre,
3. l'indécidabilité de sa théorie monadique du second ordre.

Donnons d'abord quelques définitions.

Définition 29. Étant donnée $\star$ une opération interne sur $(\Sigma, <)$, on dit que $\star$ est *strictement croissante* si, et seulement si :

$$\forall x, y, z \in \Sigma \; x \star y = z \Rightarrow (z > x \wedge z > y)$$

Définition 30. Étant donnée $\star$ une opération interne sur $(\Sigma, <)$, on dit que $\star$ est *strictement décroissante* si, et seulement si :

$$\forall x, y, z \in \Sigma \; x \star y = z \Rightarrow (z < x \wedge z < y)$$

Définition 31. Étant donnée $\star$ une opération interne sur Σ, on dit que $\star$ est *inversible à gauche* si, et seulement si :

$$\forall x, y \in \Sigma : x < y \Rightarrow (\exists z \in \Sigma : y = z \star x)$$

Définition 32. Étant donnée $\star$ une opération interne sur Σ, on dit que $\star$ est *inversible à droite* si, et seulement si :

$$\forall x, y \in \Sigma : x > y \Rightarrow (\exists z \in \Sigma : y = z \star x)$$

Définition 33. Étant donnée $\star$ une opération interne sur Σ, on dit que $\star$ est *monotonement inversible* si, et seulement si :

1. soit $\star$ est strictement croissante et inversible à droite,
2. soit $\star$ est strictement décroissante et inversible à gauche.

Dans les deux cas on note $\star^{-1}$ l'opération *inverse* de $\star$.

Exemple 20. Voici quelques exemples :

1. L'opération d'addition est monotonement inversible dans $\mathbb{N} \setminus \{0\}$ car :

   (a) elle est strictement croissante sur $\mathbb{N} \setminus \{0\}$,
   (b) elle est inversible à droite dans $\mathbb{N} \setminus \{0\}$ (son inverse est la soustraction).
2. L'opération d'addition est monotonement inversible dans $\mathbb{Q}^+ \setminus \{0\}$ car :
   (a) elle est strictement croissante sur $\mathbb{Q}^+ \setminus \{0\}$,
   (b) elle est inversible à droite dans $\mathbb{N} \setminus \{0\}$ et $\mathbb{Q}^+ \setminus \{0\}$ (son inverse est la soustraction).
3. La multiplication est monotonement inversible dans l'intervalle $]0,1[$ de $\mathbb{Q}$:
   (a) elle est strictement décroissante sur l'intervalle $]0,1[$ de $\mathbb{Q}$,
   (b) elle est inversible à gauche dans l'intervalle $]0,1[$ de $\mathbb{Q}$ (son inverse est la division).

Théorème 12. *Soit $\star$ une opération interne sur un alphabet infini Σ. Si $\star$ est monotonement inversible sur Σ alors la structure de décomposition, $\mathfrak{S}_7 = (\Sigma^*; \prec, <, \mathfrak{Dec}_\star)$, est universelle.*

Démonstration. On va procéder par double interprétation, i.e. on montrera l'interprétation de la structure $\mathfrak{S}_7$ dans une structure universelle puis l'interprétation d'une structure universelle dans $\mathfrak{S}_7$.

Pour un ordre sur un alphabet infini dénombrable Σ, la structure $\mathfrak{P} = (P_f(\Sigma); \subseteq, <)$ est définie comme suit :

1. $P_f(\Sigma)$ est l'ensemble de toutes les parties finies de Σ.
2. $x < y$ si $x = \{n\}$ et $y = \{m\}$ (x et y sont des singletons) et $n < m$.

La structure $\mathfrak{P}$ est universelle (voir section 1.2.2.5). La double interprétation est établie selon la bijection κ suivante (et son inverse) :

$$\begin{array}{rcl} \kappa : \Sigma^* & \longrightarrow & P_f(\Sigma) \\ x = x_1 \dots x_i \dots x_n & \longmapsto & \kappa(x) = \{\star_{j=1}^{i} x_j \mid 1 \leq i \leq n\} \end{array}$$

$$\begin{array}{rcl} \kappa^{-1} : P_f(\Sigma) & \longrightarrow & \Sigma^* \\ e = \{e_1 < \dots < e_i < \dots < e_n\} & \longmapsto & \kappa^{-1}(e) = \\ & & e_1.(e_2 \star^{-1} e_1). \dots .(e_i \star^{-1} e_{i-1}). \dots .(e_n \star^{-1} e_{n-1}) \end{array}$$

Par convention $\kappa(\varepsilon) = \emptyset$.

La monotonie de l'opération $\star$ permet de construire, à partir d'un mot x sur Σ, en utilisant la bijection κ, un ensemble e dont le nombre d'éléments est égal au nombre de lettres de x. L'inversibilité de l'opération $\star$ permet de construire un mot de Σ à partir de n'importe quel ensemble fini de Σ.

1. La structure $\mathfrak{S}_7 = (\Sigma^*; \prec, <, \mathfrak{Dec}_\star)$ est FO-interprétable dans la structure $\mathfrak{P} = (P_f(\Sigma); \subseteq, <)$ qui est universelle (voir section1.2.2.5). Ceci est établi grâce au codage κ :
   (a) Chaque élément de Σ^* est codé par un et un seul élément de $P_f(\Sigma)$. L'application κ est une bijection, ainsi $\kappa(\Sigma^*)$ l'image du domaine de $\mathfrak{S}_7$ est égale à $P_f(\Sigma)$, qui est évidemment définissable dans $\mathfrak{P}$.

(b) L'ensemble $\kappa(\prec)$ image du prédicat $\prec$, est définissable dans $\mathfrak{P}$. Soient u, v des mots de Σ^* et soient $x = \kappa(u)$, $y = \kappa(v)$ leurs codes respectifs. Alors pour que u soit un préfixe de v, les deux ensembles x et y doivent vérifier les trois conditions suivantes (voir l'exemple 21) :

i. L'ensemble x est inclus dans y :

$$x \subseteq y$$

ii. Le plus petit élément de x est égal au plus petit élément de y. Il existe un singleton m tel que :

$$(m \subseteq x) \wedge (m \subseteq y) \wedge (\forall z \ : \ z < m \rightarrow \neg(z \subseteq x) \wedge \neg(z \subseteq y))$$

iii. Tous les éléments de y qui sont entre le plus petit élément et le plus grand élément de x doivent être dans x. Pour les singletons $y_1, y_2, y_3 \subseteq y$ on a :

$$((y_1 \leq y_2 \leq y_3) \wedge (y_1 \subseteq x) \wedge (y_3 \subseteq x)) \rightarrow (y_2 \subseteq x)$$

(c) Par définition l'ensemble $\kappa(<)$, image du prédicat $<$, est définissable dans $\mathfrak{P}$ relativement au prédicat $<$:

$$a, b \in \Sigma \ : \ a < b \Leftrightarrow \kappa(a) < \kappa(b)$$

(d) Aussi, par définition, l'ensemble $\kappa(\mathfrak{Dec}_\star)$ image du prédicat $\mathfrak{Dec}_\star$, est définissable dans $\mathfrak{P}$ relativement au prédicat $\subseteq$:

$$u, v \in \Sigma^* \ : \ \mathfrak{Dec}_\star(u, v) \Leftrightarrow \kappa(u) \subseteq \kappa(v)$$

2. La structure universelle $\mathfrak{P} = (P_f(\Sigma); \subseteq, <)$ est FO-interprétable dans la structure $\mathfrak{S}_7 = (\Sigma^*; \prec, <, \mathfrak{Dec}_\star)$. Ceci est établi grâce au codage κ :

(a) Chaque élément de $P_f(\Sigma)$ est codé par un et un seul élément de Σ^*. L'application κ^{-1} est une bijection, ainsi $\kappa^{-1}(P_f(\Sigma))$ l'image du domaine de $\mathfrak{P}$ est égal à Σ^*, qui est évidemment définissable dans $\mathfrak{S}_7$.

(b) Par définition l'ensemble $\kappa^{-1}(<)$, image du prédicat $<$, est définissable dans $\mathfrak{S}_7$ relativement au prédicat $<$.

(c) Aussi, par définition, l'ensemble $\kappa^{-1}(\subseteq)$ image du prédicat $\subseteq$, est définissable dans $\mathfrak{S}_7$ relativement au prédicat $\mathfrak{Dec}_\star$.

□

Exemple 21. Pour $\star = +$ et $v = [1, 11, 5, 5, 7, 11, 5]$, on a $y = \kappa(v) = \{1, 12, 17, 22, 29, 38, 43\}$:

u	$x = \kappa(u)$	
$[12, 5, 5]$	$\{12, 17, 22\}$	$x \subseteq y$ mais u n'est pas un préfixe de v
$[1, 21, 5]$	$\{1, 22, 29\}$	$x \subseteq y$ et $\min(x) = \min(y)$ mais u n'est pas un préfixe de v
$[1, 11, 5]$	$\{1, 12, 17\}$	x et y vérifient les trois conditions : u est un préfixe de v.

On a le corollaire suivant.

Corollaire 11. *Soit $\star$ une opération interne sur un alphabet infini Σ. Si $\star$ est monotonement inversible alors la structure de décomposition $\mathfrak{S}_7 = (\Sigma^*; \prec, <, \mathfrak{Dec}_\star)$ est automatique, sa théorie du premier ordre est décidable tandis que sa théorie monadique du second ordre ne l'est pas.*

Démonstration. C'est une conséquence directe de :
- la double interprétation (lemme 3) entre la structure $\mathfrak{S}_7 = (\Sigma^*; \prec, <, \mathfrak{Dec}_\star)$ et la structure $\mathfrak{P} = (P_f(\Sigma); \subseteq, <)$,
- la décidabilité de la théorie logique du premier ordre de la structure $\mathfrak{P}$,
- l'indécidabilité de la théorie monadique de second ordre de la structure $\mathfrak{P}$.

□

Considérons la structure $\mathfrak{S}_8 = (\Sigma^*; \prec, <, \mathfrak{Dec}_\star, \sim)$ obtenue à partir de la structure $\mathfrak{S}_7$ en ajoutant le prédicat $\sim$. Pour des ensembles finis A et B, notons $A \approx B$ la relation binaire qui est vraie si, et seulement si, A et B ont même cardinalité. En utilisant le codage κ, il est évident que des mots u et v ont la même longueur si, et seulement si, les ensembles $\kappa(u)$ et $\kappa(v)$ ont la même cardinalité. Ceci nous permet d'interpréter dans la structure $\mathfrak{S}_8$ la structure $(P_f(\Sigma); \subseteq, <, \approx)$ dont la théorie du premier ordre est indécidable [Rob58]. On a ainsi le corollaire suivant.

Corollaire 12. *Soit $\star$ une opération interne sur un alphabet infini Σ. Si $\star$ est monotonement inversible alors la structure de décomposition étendue $\mathfrak{S}_8 = (\Sigma^*; \prec, <, \mathfrak{Dec}_\star, \sim)$ a une théorie de premier ordre (et du second ordre monadique) indécidable. Par conséquent cette structure n'est pas automatique.*

2.7 Autres résultats

Nous montrons dans cette section l'indécidabilité de la théorie du premier ordre de la structure $\mathfrak{S}_9 = (\Sigma^*; \prec, \mathfrak{clone}, \mathfrak{lastNew}, \mathfrak{firstZero}, \sim, \ominus)$ telle que :

1. Σ^* est l'ensemble des mots finis sur les entiers.
2. Pour un mot x de Σ^*, le prédicat $\mathfrak{lastNew}(x)$ est vrai si, et seulement si, x se termine par une nouvelle lettre.

$$\mathfrak{lastNew}(x) \Leftrightarrow x = a_1 a_2 \ldots a_n \text{ avec } a_1, a_2, \ldots, a_n \in \Sigma \text{ et } \forall i < n \; a_n \neq a_i$$

3. Pour un mot x de Σ^*, le prédicat $\mathfrak{firstZero}(x)$ est vrai si, et seulement si, x débute par un zéro. $\mathfrak{firstZero}(x) \Leftrightarrow x = 0u$ avec $u \in \Sigma^*$.
4. Pour des mots x, y et $z \in \Sigma^*$, le prédicat $\ominus(x, y, z)$ est vrai si, et seulement si, l'opération

de soustraction de y par x, lettre par lettre, est égal à z.

$$\ominus(x,y,z) \Leftrightarrow \begin{cases} \text{si } |x| > |y| \\ \begin{cases} x = x_1 \dots x_i \dots x_n x_{n+1} \dots x_m \text{ et} \\ y = y_1 \dots y_i \dots y_n \text{ et} \\ z = (x_1 - y_1) \dots (x_i - y_i) \dots (x_n - y_n)(x_{n+1} - 0) \dots (x_m - 0) \end{cases} \\ \text{sinon} \\ \begin{cases} x = x_1 \dots x_i \dots x_n \text{ et} \\ y = y_1 \dots y_i \dots y_n y_{n+1} \dots y_m \text{ et} \\ z = (x_1 - y_1) \dots (x_i - y_i) \dots (x_n - y_n)(0 - y_{n+1}) \dots (0 - y_m) \end{cases} \end{cases}$$

Nous pouvons vérifier que le prédicat $\mathfrak{diff}$ est définissable dans la théorie de $\mathfrak{S}_9$. Ce prédicat est vrai pour un mot donné si, et seulement si, le prédicat $\mathfrak{lastNew}$ est vérifié pour chaque préfixe de mot donné.

Exemple 22. Soient x, y, z des mots de Σ^*. Pour $x = [1,7,8,8,3,7]$, $y = [0,5,5,4]$ et $z = [1,2,3,4,3,7]$:

1. $\mathfrak{lastNew}(y)$, $\mathfrak{firstZero}(y)$ et $\ominus(x,y,z)$ sont vrais.
2. $\mathfrak{lastNew}(x)$ et $\mathfrak{firstZero}(x)$ sont faux.

On démontre notre résultat d'indécidabilité par l'interprétation de la structure :

$$\mathfrak{C} = (\Sigma^*; \prec, \varepsilon, \mathfrak{pred}, \mathfrak{eqLast})$$

dans $\mathfrak{S}_9$. L'indécidabilité de la théorie du premier ordre de $\mathfrak{C}$ été prouvée par Choffrut & Grigorieff dans [CG09a, CG09b].

Théorème 13. *La structure* $\mathfrak{C} = (\Sigma^*; \prec, \varepsilon, \mathfrak{pred}, \mathfrak{eqLast})$ *est* FO*-interprétable dans :*

$$\mathfrak{S}_9 = (\Sigma^*; \prec, \mathfrak{clone}, \mathfrak{lastNew}, \mathfrak{firstZero}, \sim, \ominus)$$

Démonstration. Nous établissons l'interprétabilité grâce au codage suivant.

Soit $\nu : \Sigma^* \to \Sigma^*$ la fonction qui associe chaque mot $x = x_1 x_2 x_3 \dots x_n$ le mot $\nu(x) = (x_n)(x_n - x_1)(x_n - x_2)(x_n - x_3) \dots (x_n - x_{n-1})$. Par convention $\nu(\varepsilon) = \varepsilon$.

Exemple 23. En continuant le dernier exemple, nous avons :

1. $\nu(x) = [7,6,0,-1,-1,4]$,
2. $\nu(y) = [4,-1,-1]$,
3. $\nu(z) = [7,5,4,3,4]$.

Afin de simplifier la preuve, nous introduisons le prédicat $<_\sim$ qui est vrai pour des mots x et y de Σ^* si, et seulement si, x est plus long que y, c'est-à-dire : $x <_\sim y \Leftrightarrow |x| < |y|$. Il est clair que ce prédicat est définissable dans $\mathfrak{S}_9$ en combinant les prédicats $\prec$ et $\sim$.

Nous allons maintenant prouver que ν fournit une interprétation de la structure $\mathfrak{C}$ dans $\mathfrak{S}_9$.

1. La fonction ν est une bijection, ainsi l'image $\nu(\Sigma^*)$ du domaine de $\mathfrak{C}$ est égal à Σ^*, qui est évidemment définissable dans $\mathfrak{S}_9$.

2. L'image $\nu(\mathfrak{eqLast})$ du prédicat $\mathfrak{eqLast}$, est définissable dans $\mathfrak{S}_9$ par la formule :

$$\exists z \ \ominus(\nu(x), \nu(y), z) \wedge \mathfrak{firstZero}(z)$$

En effet, par définition, le codage ν consiste à déplacer la dernière lettre d'un mot à la première position donc si deux mots se terminent par la même lettre, la soustraction des codes respectifs de chaque mot donne zéro à la première position.

3. L'ensemble $\nu(\prec)$, image du prédicat $\prec$, est définissable dans $\mathfrak{S}_9$ par la formule :

$$(\exists w \exists z \ \ominus(\nu(y), \nu(x), z) \wedge (w \prec z) \wedge (\nu(x) <_{\sim} w) \wedge (\forall p \ (p \prec w) \Rightarrow \mathfrak{clone}(p)))$$

Pour des mots x et y, si x est un préfixe strict de y alors la soustraction du code de y depuis le code de x est un mot dont les première $(|x| + 1)$ lettres sont identiques.

On remarque que ε et $\mathfrak{pred}$ sont déjà définissables dans $\mathfrak{C}$.

□

Corollaire 13. *La théorie du premier ordre de la structure $\mathfrak{S}_9$ est indécidable.*

Démonstration. Cela découle de l'indécidabilité de la théorie du premier ordre de la structure $\mathfrak{C} = (\Sigma^*; \prec, \varepsilon, \mathfrak{pred}, \mathfrak{eqLast})$ [CG09a, CG09b], et du théorème 13. □

2.8 Conclusion

Nous avons prouvé dans ce chapitre l'automaticité et la décidabilité de la théorie de certaines structures portant sur un alphabet infini dénombrable, en développant des codages de plus en plus raffinés, qui conservent la décidabilité et l'automaticité de ces structures tout en ajoutant des prédicats plus complexes et des relations plus expressives.

D'autres questions restent pour l'instant sans réponse comme celui de l'automaticité de la structure $(\Sigma^*; \prec, \mathfrak{clone}, \sim)$ ou encore la décidabilité de la théorie logique du premier ordre de la structure $(\Sigma^*; \prec, \mathfrak{clone}, \{\mathfrak{diff}_i \mid i > 2\}, \mathfrak{diff})$ et la théorie monadique du seconde ordre des structures :

- $\mathfrak{S}_1 = (\Sigma^*; \prec, \mathfrak{clone}, \mathfrak{less}, \{\mathfrak{mod}_{p,q} \mid p, q \in \mathbb{N}\}, \sim, \oplus)$,
- $\mathfrak{S}_3 = (\Sigma^*; \prec, \mathfrak{clone}, \{\mathfrak{diff}_i \mid i > 2\}, \sim)$,
- $\mathfrak{S}_5 = (\Sigma^*; \prec, \mathfrak{clone}, \mathfrak{diff}, \sim)$,
- $\mathfrak{S}_6 = (\Sigma^*; \prec, \{R^{\oplus}_{f_i} \mid i > 0\}, \sim)$.

CHAPITRE 3

RÉDUCTION DU NOMBRE DE TRANSITIONS POUR LES AUTOMATES FINIS TRIANGULAIRES

NOUS généralisons, dans ce chapitre, le concept de *Common Follow Sets* [HSW01] d'une expression rationnelle aux automates finis homogènes. En se basant sur cette généralisation et en utilisant des arbres binaires particuliers, nous proposons un algorithme quasi-linéaire pour réduire le nombre de transitions des *automates triangulaires*, à savoir les automates reconnaissant le langage $L(E_n)$ dénoté par l'expression rationnelle $E_n = (1 + \varepsilon) \cdot (2 + \varepsilon) \cdot (3 + \varepsilon) \cdots (n + \varepsilon)$. Nous montrons en effet que l'automate fini non-déterministe sans ε-transition ainsi produit est asymptotiquement minimal, dans le sens où son nombre de transitions est équivalent à $n(\log_2 n)^2$, qui correspond à la fois à la borne inférieure et supérieure. Ce travail a été réalisé en collaboration avec DJELLOUL ZIADI de l'université de Rouen, France.

Ce chapitre est organisé comme suit. Dans un premier temps, dans la section 3.1, nous montrons comment nous pouvons étendre le concept de CFS aux automates homogènes. Puis, dans la section 3.2.2, nous introduisons CFSZ, une famille spécifique de systèmes CFS associés aux langages $L(E_n)$, et nous mettons en évidence une bijection adéquate avec l'ensemble des arbres binaires complets à $n + 2$ feuilles. Ensuite, nous introduisons la notion de *Z-arbre* qui nous permet de concevoir un algorithme de complexité d'exécution $O(n \log_2 n)$ pour engendrer les Z-arbres minimaux ainsi que les automates minimisés associés. Dans la section 3.2.6, nous montrons des résultats expérimentaux. Dans la section 3.2.5 nous montrons que le nombre de transitions dans les automates produit par notre méthode de réduction est asymptotiquement minimal. Dans la section 3.3 on présente une application particulière des P-arbres permettant d'engendrer la suite des nombres premiers en utilisant les arbres binaires. Enfin, la section 3.4 conclut le chapitre en discutant une méthode heuris-

tique possible, qui peut être utilisée pour réduire le nombre de transitions dans un automate homogène quelconque.

3.1 CFS pour les automates homogènes

Hromkovič *et al.* [HSW01] ont donné un algorithme basé sur la notion de *Common Follow Sets* pour convertir une expression rationnelle de taille n en un automate fini non-déterministe sans ε-transition ayant $O(n)$ états (au plus $2n-1$) et $O(n(\log_2 n)^2)$ transitions (au plus $\frac{4}{\log_2(\frac{3}{2})}n(\log_2(n))^2$). Hromkovič *et al.* [HSW01] partent d'une expression rationnelle et non pas d'un automate mais, au cœur de leur construction, il y a l'automate de Glushkov [Glu61] qui est un automate homogène (tout automate de Glushkov est homogène mais pas l'inverse). Montrons dans cette section comment cette notion peut être facilement étendue aux automates homogènes.

Soient $n \in \mathbb{N}$ un entier naturel et $\Sigma = \{1, 2, \ldots, n\}$ un alphabet de n symboles.

Soit $\mathcal{A} = (Q, \Sigma, \{q_0\}, \delta, F)$ un automate homogène. Afin de distinguer les états finaux de $\mathcal{A}$, nous introduisons un *état artificiel* noté # qui n'est pas dans Q. On définit sur Q la fonction *follow* :

$$\text{follow}(q) = \begin{cases} \{p \mid \exists a \in \Sigma \text{ tel que } (q, a, p) \in \delta\} \cup \{\#\} \text{ si } q \in F, \\ \{p \mid \exists a \in \Sigma \text{ tel que } (q, a, p) \in \delta\} \text{ sinon.} \end{cases}$$

On associe à chaque état $q \in Q$ de l'automate $\mathcal{A}$ un recouvrement de l'ensemble $\text{follow}(q)$, appelé *décomposition* et noté $\text{dec}(q) = \{Q_1, Q_2, \ldots, Q_k\}$ telle que $Q_i \subseteq \text{follow}(q)$. Dans le cas où $\text{dec}(q)$ est une partition de l'ensemble $\text{follow}(q)$, la décomposition $\text{dec}(q)$ sera appelée *décomposition de partitions*. La figure 3.1 donne des exemples de décompositions.

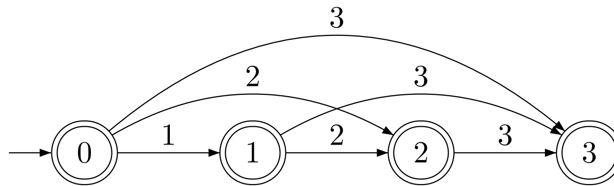


FIGURE 3.1 – On a $\text{follow}(0) = \{1, 2, 3, \#\}$. Voici trois exemples de décompositions de $\text{follow}(0)$, les deux premières sont des décompositions de partitions : (i) $\text{dec}(0) = \{\{1, 2\}, \{3, \#\}\}$ (ii) $\text{dec}(0) = \{\{1\}, \{2\}, \{3, \#\}\}$ (iii) $\text{dec}(0) = \{\{1, 2\}, \{2, 3, \#\}\}$.

En se basant sur les décompositions, nous définissons le système et l'automate de *Common Follow Sets* :

Définition 34 (Système de *Common Follow Sets*)**.** Soit $\mathcal{A}$ un automate homogène. Étant donnée une décomposition de l'ensemble follow de chaque état de $\mathcal{A}$, un système CFS pour $\mathcal{A}$ est défini comme étant $S(\mathcal{A}) = (\text{dec}(q))_{q \in Q}$ tel que $\text{dec}(q)$ est une décomposition de $\text{follow}(q)$.

Définition 35 (Automate de *Common Follow Sets*)**.** Soient $\mathcal{A} = (Q, \Sigma, \{q_0\}, \delta, F)$ un automate homogène et $S(\mathcal{A})$ un système de *Common Follow Sets* qui lui est associé. L'automate de *Common Follow Sets* associé à $S(\mathcal{A})$ est défini par $\mathcal{C}_{S(\mathcal{A})} = (Q', \Sigma, I', \delta', F')$ tel que :
- $Q' = \bigcup_{q \in Q} \mathrm{dec}(q)$,
- $I' = \mathrm{dec}(q_0)$,
- $F' = \{Q_1 \mid Q_1 \in Q' \text{ et } \# \in Q_1\}$,
- $\delta' = \{(Q_1, a, Q_2) \mid \exists q \in Q_1 \text{ tel que } h(q) = a \text{ et } Q_2 \in \mathrm{dec}(q)\}$.

On rappelle que $h(q)$ est la fonction qui retourne l'étiquette des transitions entrantes de l'état q (voir section 1.2.1.3.4). Les figures 3.2 et 3.3 illustrent la définition par des exemples.

Par définition, la façon dont l'automate CFS est construit permet de retrouver chaque chemin acceptant de l'automate $\mathcal{C}_{S(\mathcal{A})}$ dans l'automate $\mathcal{A}$ et inversement. Ainsi on a le théorème suivant.

Théorème 14. *Soient $\mathcal{A}$ un automate homogène, $S(\mathcal{A})$ un système de* Common Follow Sets *qui lui est associé et $\mathcal{C}_{S(\mathcal{A})}$ son automate de* Common Follow Sets. *Les automates $\mathcal{C}_{S(\mathcal{A})}$ et $\mathcal{A}$ reconnaissent le même langage.*

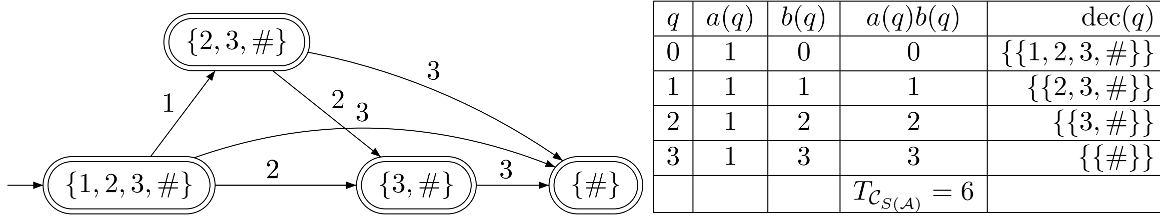
Pour évaluer le nombre de transitions dans l'automate $\mathcal{C}_{S(\mathcal{A})}$, on définit sur l'ensemble des états de $\mathcal{A}$ deux fonctions :
- $a(q) = |\,\mathrm{dec}(q)|$, la taille de la décomposition de l'ensemble $\mathrm{follow}(q)$,
- $b(q) = |\{Q_1 \in Q' \mid q \in Q_1\}|$, le nombre d'états de Q' qui contiennent l'état q.

Lemme 5. *Le nombre de transitions $T_{\mathcal{C}_{S(\mathcal{A})}}$ de $\mathcal{C}_{S(\mathcal{A})}$ vérifie $T_{\mathcal{C}_{S(\mathcal{A})}} \leq \sum_{q \in Q} a(q)b(q)$.*

Démonstration. De chaque état $Q_1 \in Q'$, contenant l'élément q, sortent des transitions étiquetées par $h(q)$ vers les éléments constituant la décomposition de q. Donc on compte $a(q)b(q)$ transitions pour l'élément q. Le nombre total de transitions est donc majoré par $\sum_{q \in Q} a(q)b(q)$. Si l'automate homogène initial contient au moins deux états distincts p et q tels que $h(p) = h(q)$ alors la formule proposée dans le lemme compte deux fois certaines transitions. Ainsi, on remarque que si on a $\forall\, p, q \in Q \setminus \{q_0\} : p \neq q \Rightarrow h(p) \neq h(q)$ alors on a l'égalité : $T_{\mathcal{C}_{S(\mathcal{A})}} = \sum_{q \in Q} a(q)b(q)$. □

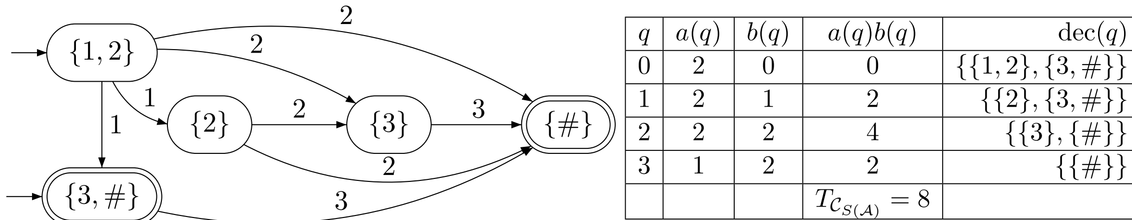
Les figures 3.2 et 3.3 montrent deux automates CFS associés à l'automate $\mathcal{A}_3$.

D'après le lemme 5, nous pouvons voir que le nombre de transitions dans un automate CFS dépend du système de décomposition. Aussi, en général, une décomposition qui n'est pas une partition induira plus de transitions qu'une décomposition de partitions. Par conséquent, dans ce qui suit, nous nous intéressons principalement à des décompositions de partitions. Comme mentionné dans l'introduction, notre étude se concentrera sur les automates CFS associés à la famille d'automates triangulaires $(\mathcal{A}_n)_{n \geq -1}$, eux-même associés aux langages $L(E_n)$.



q	$a(q)$	$b(q)$	$a(q)b(q)$	$\mathrm{dec}(q)$
0	1	0	0	$\{\{1, 2, 3, \#\}\}$
1	1	1	1	$\{\{2, 3, \#\}\}$
2	1	2	2	$\{\{3, \#\}\}$
3	1	3	3	$\{\{\#\}\}$
			$T_{\mathcal{C}_{S(\mathcal{A})}} = 6$	

FIGURE 3.2 – Un automate CFS associé à l'automate triangulaire $\mathcal{A}_3$ ayant un état initial et 6 transitions.



q	$a(q)$	$b(q)$	$a(q)b(q)$	$\mathrm{dec}(q)$
0	2	0	0	$\{\{1, 2\}, \{3, \#\}\}$
1	2	1	2	$\{\{2\}, \{3, \#\}\}$
2	2	2	4	$\{\{3\}, \{\#\}\}$
3	1	2	2	$\{\{\#\}\}$
			$T_{\mathcal{C}_{S(\mathcal{A})}} = 8$	

FIGURE 3.3 – Un automate CFS associé à l'automate triangulaire $\mathcal{A}_3$ ayant deux états initiaux et 8 transitions.

3.2 Automates triangulaires $\mathcal{A}_n$

On introduira dans cette section une classe particulière d'automates, appelés *automates triangulaires*.

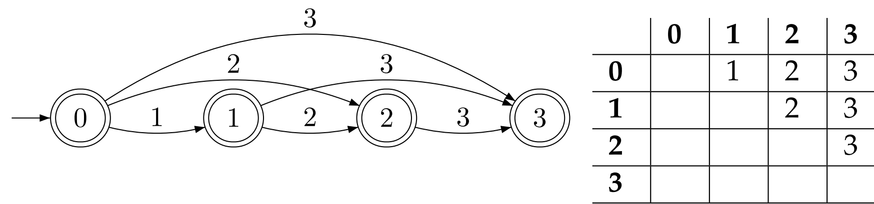
Soient un entier naturel $n \in \mathbb{N}$ et un alphabet $\Sigma = \{1, 2, \ldots, n\}$ de n symboles.

Définition 36. Considérons l'expression rationnelle $E_n = (1 + \varepsilon) \cdot (2 + \varepsilon) \cdot (3 + \varepsilon) \cdots (n + \varepsilon)$ pour $n > 0$. On note $L(E_n)$ le *langage triangulaire* dénoté par l'expression rationnelle E_n. Par convention on note $L(E_0) = \{\varepsilon\}$ et $L(E_{-1}) = \emptyset$.

Définition 37. Soit $n \geq 1$. L'*automate triangulaire* $\mathcal{A}_n = (Q, \Sigma, I, \delta, F)$ défini par :

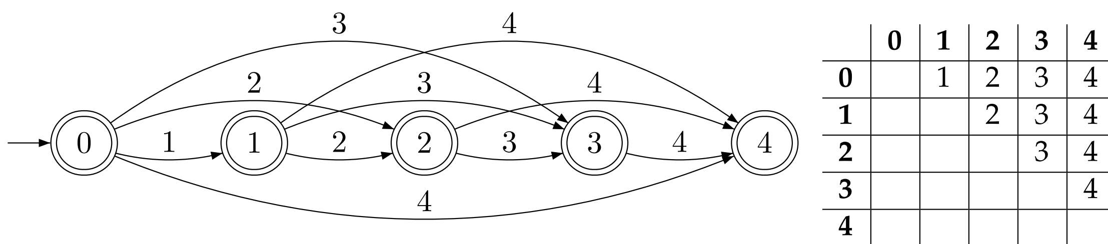
- $\Sigma = \{1, 2, \ldots, n\}$,
- $Q = \Sigma \cup \{0\}$,
- $F = Q$,
- $I = \{0\}$,
- $\delta = \{(p, q, q) \in Q \times \Sigma \times Q \mid q > p\}$.

est l'automate fini déterministe minimal reconnaissant le langage triangulaire $L(E_n)$. Par convention on note $\mathcal{A}_0$ et $\mathcal{A}_{-1}$ les automates déterministes minimaux reconnaissant $L(E_0)$ et $L(E_{-1})$ respectivement.



	0	**1**	**2**	**3**
0		1	2	3
1			2	3
2				3
3				

FIGURE 3.4 – L'automate $\mathcal{A}_3$ et sa table de transition.



	0	**1**	**2**	**3**	**4**
0		1	2	3	4
1			2	3	4
2				3	4
3					4
4					

FIGURE 3.5 – L'automate $\mathcal{A}_4$ et sa table de transition.

Remarquons que la table de transition d'un automate $\mathcal{A}_n$ est une matrice carrée *triangulaire* supérieure.

	0	**1**	**2**	**3**	…	**k**	…	**n**
0		1	2	3	…	k	…	n
1			2	3	…	k	…	n
2				3	…	k	…	n
⋮					…			⋮
k-1						k	…	n
k							…	n
⋮								⋮
n-1								n
n								

FIGURE 3.6 – Table de transition de l'automate triangulaire $\mathcal{A}_n$.

Dans ce qui suit, nous nous intéressons à la réduction du nombre de transitions de l'automate triangulaire $\mathcal{A}_n$. On donnera un algorithme qui calcule des systèmes de CFS particuliers $S(\mathcal{A}_n)$ fournissant des automates $\mathcal{C}_{S(\mathcal{A}_n)}$ ayant $n+1$ états avec un nombre de transitions plus petit.

3.2.1 Nombre d'états dans l'automate réduit de $\mathcal{A}_n$

Il est facile de voir que tout automate fini reconnaissant le langage triangulaire $L(E_n)$ possède au moins $n+1$ états (pour accepter déjà le mot $1.2.3\ldots n$). Nous avons besoin de la proposition suivante.

Proposition 2. *Il existe un automate fini, non-déterministe, sans ε-transition, ayant $n+1$ états, minimal en nombre de transitions qui reconnaît le langage triangulaire $L(E_n)$.*

Démonstration. L'idée est de montrer, qu'à partir de n'importe quel automate minimal en nombre de transitions reconnaissant le langage triangulaire $L(E_n)$, nous pouvons obtenir un automate équivalent minimal en nombre de transitions reconnaissant le langage triangulaire $L(E_n)$ et possédant $n+1$ états. Soit $\mathcal{A} = (Q, \Sigma, I, F, \delta)$ un automate fini non-déterministe sans ε-transition reconnaissant $L(E_n)$. L'automate $\mathcal{A}$ a au moins $n+1$ états. Pour un état $q \in Q$, soit $X_q = \{i \mid \exists p \in Q \text{ tel que } (p, i, q) \in \delta\}$. Définissons le *niveau* de q par : $\text{level}(q) = \max\ X_q$ si $X_q \neq \emptyset$, et $\text{level}(q) = 0$ sinon. Il est clair que le nombre de niveaux possibles est $n+1$. Nous définissons sur l'ensemble des états de $\mathcal{A}$ la relation d'équivalence $\sim$ définie par : $p \sim q \Leftrightarrow \text{level}(p) = \text{level}(q)$. On note $[q]$ la classe d'équivalence de q. Soit $\mathcal{A}/{\sim} = (Q/{\sim}, \Sigma, I/{\sim}, F/{\sim}, \delta/{\sim})$ l'automate quotient obtenu à partir de $\mathcal{A}$ en fusionnant les états équivalents :

- $Q/{\sim} = \{[q] \mid q \in Q\}$,
- $I/{\sim} = \{[q] \mid q \in I\}$,
- $F/{\sim} = \{[q] \mid q \in F\}$,
- $\delta/{\sim} = \{([p], i, [q]) \mid (p, i, q) \in \delta\}$.

Le nombre d'états de $\mathcal{A}/{\sim}$ est $n+1$ et chaque chemin dans $\mathcal{A}$ correspond à un chemin dans $\mathcal{A}/{\sim}$. D'autre part, un mot w reconnu par $\mathcal{A}/{\sim}$ est tel que pour chaque facteur $i.j$ de longueur deux dans w, nous avons $i < j$ (car l'automate triangulaire $\mathcal{A}_n$ reconnaît exactement l'ensemble de toutes les sous-séquences du mot $1.2.3\ldots n$) ce qui caractérise le langage $L(E_n)$. Ainsi, $\mathcal{A}/{\sim}$ reconnaît exactement $L(E_n)$. Maintenant si le nombre de transitions dans $\mathcal{A}/{\sim}$ est inférieur au nombre de transitions dans $\mathcal{A}$, ceci contredit la minimalité du nombre de transitions de $\mathcal{A}$, ce qui veut dire que $\mathcal{A}$ contenait $n+1$ états. Si le nombre de transitions dans $\mathcal{A}/{\sim}$ est égal au nombre de transitions dans $\mathcal{A}$ alors $\mathcal{A}/{\sim}$ est un automate minimal en nombre de transition ayant $n+1$ états. Enfin, le nombre de transitions de $\mathcal{A}/{\sim}$ ne dépasse pas le nombre de transitions de $\mathcal{A}$. □

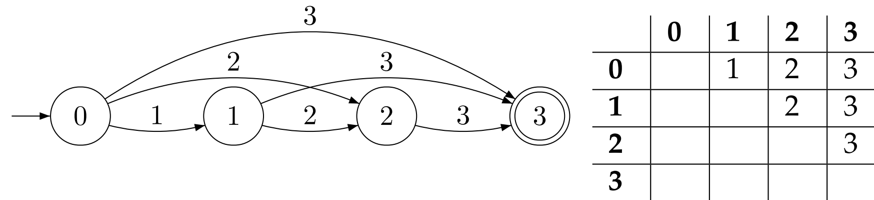
3.2.2 CFS pour les automates $\mathcal{A}_n$

Nous allons définir, pour un entier naturel n fixé, des systèmes de CFS particuliers pour les automates triangulaires $\mathcal{A}_n$ qu'on note $Z(\mathcal{A}_n)$. On note $\text{CFSZ}(n)$ l'ensemble de tous les $Z(\mathcal{A}_n)$. Chaque système $Z(\mathcal{A}_n) \in \text{CFSZ}(n)$ donne un automate fini $\mathcal{C}_{Z(\mathcal{A}_n)}$ non-déterministe sans ε-transition ayant $n+1$ états dont un ou plusieurs initiaux.

Sans perte de généralité et dans le but de faciliter les notations et l'écriture des preuves, dans tout ce qui suit nous associons l'état fictif $\#$, distinguant les états finaux, au nombre $n+1$.

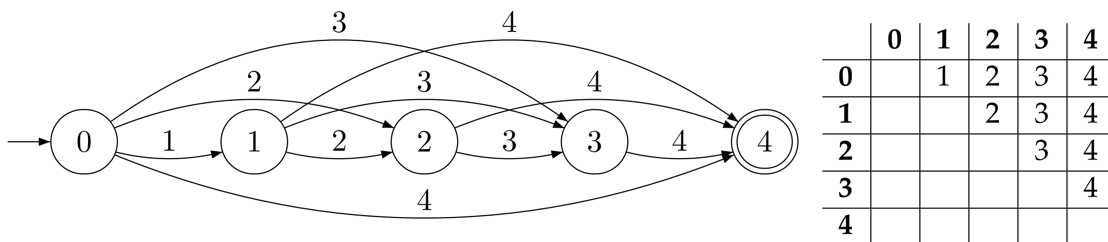
Il est facile de voir que le langage $L(E_n)$ est exactement le langage reconnu par l'automate ayant $n+2$ états dont un seul initial et un seul final $(Q, \Sigma, I, \delta, F)$ défini par :

- $\Sigma = \{1, 2, \ldots, n\}$,
- $Q = \Sigma \cup \{0, n+1\}$,
- $I = \{0\}$,
- $F = \{n+1\}$,
- $\delta = \{(p, q, q) \in Q \times \Sigma \times Q \mid q > p\}$.



	0	**1**	**2**	**3**
0		1	2	3
1			2	3
2				3
3				

FIGURE 3.7 – L'automate ayant un seul état final équivalent à $\mathcal{A}_2$ plus sa table de transition.



	0	**1**	**2**	**3**	**4**
0		1	2	3	4
1			2	3	4
2				3	4
3					4
4					

FIGURE 3.8 – L'automate ayant un seul état final équivalent à $\mathcal{A}_3$ plus sa table de transition.

Pour mieux comprendre les systèmes $Z(\mathcal{A}_n)$, nous allons expliquer comment on peut les obtenir à partir de la table de transition de l'automate ayant un seul état final équivalant à l'automate triangulaire $\mathcal{A}_n$. Cette table de transition, notée M^{n+2}, est une matrice carrée $(n+2) \times (n+2)$ triangulaire supérieure dont la première diagonale constitue l'ensemble $\{1, 2, \ldots, n, n+1\}$. La case $M^{n+2}[i, j]$ correspond à une transition de l'état i vers l'état j étiquetée par la valeur de cette case. Choisissons d'abord un élément quelconque de la diagonale, disons k. Ce choix divise la matrice en trois parties :

1. Un bloc B_k de surface rectangulaire maximale de $(n-k+2) \times k$. Il possède l'élément $M^{n+2}[k-1, k]$ comme coin inférieur gauche et $M^{n+2}[0, n+1]$ comme coin supérieur droit. Ce bloc est en fait associé à l'ensemble $\{k, k+1, \ldots, n, n+1\}$ qui fera partie de la décomposition de chaque état q inférieur ou égal $k-1$.
2. Une sous-matrice triangulaire supérieure M_a^k de taille $k \times k$.
3. Une sous-matrice triangulaire supérieure M_b^{n-k+2} de taille $(n-k+2) \times (n-k+2)$.

Donc l'idée est de choisir d'abord un élément sur la première diagonale. Ensuite, à partir de cet élément, on crée un bloc rectangulaire de surface maximale dont le coin inférieure gauche est l'élément choisi :

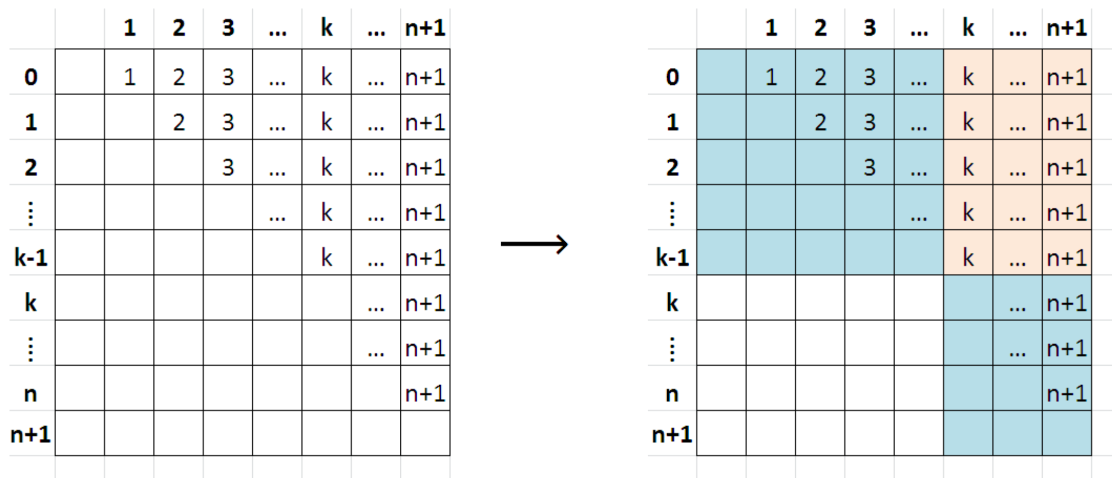


FIGURE 3.9 – Décomposition en trois éléments.

L'étape suivante consiste à réitérer ce découpage [1] sur les deux sous-matrices triangulaires en résultant, en choisissant un élément quelconque de la diagonale de chacune de ces deux sous-matrices. Ainsi, ce processus de décomposition est répété jusqu'à épuisement de tous les éléments de la diagonale de la matrice initiale (la table de transition). À la fin de ce procédé, tous les éléments de la table de transition sont regroupés en blocs.

Exemple 24. Dans cet exemple on montre deux exemples de système de partitions CFSZ associés à deux découpages en blocs de la matrice de transition de l'automate triangulaire $\mathcal{A}_3$:

$$\begin{aligned}
\mathrm{dec}(0) &= \{ \{ 1, 2, 3, 4 \} \} & 0 &: 1\ 2\ 3\ 4 \\
\mathrm{dec}(1) &= \{ \{ 2 \}, \{ 3, 4 \} \} & 1 &: 2\ 3\ 4 \\
\mathrm{dec}(2) &= \{ \{ 3, 4 \} \} & 2 &: 3\ 4 \\
\mathrm{dec}(3) &= \{ \{ 4 \} \} & 3 &: 4
\end{aligned}$$

$$\begin{aligned}
\mathrm{dec}(0) &= \{ \{ 1 \}, \{ 2, 3, 4 \} \} & 0 &: 1\ 2\ 3\ 4 \\
\mathrm{dec}(1) &= \{ \{ 2, 3, 4 \} \} & 1 &: 2\ 3\ 4 \\
\mathrm{dec}(2) &= \{ \{ 3, 4 \} \} & 2 &: 3\ 4 \\
\mathrm{dec}(3) &= \{ \{ 4 \} \} & 3 &: 4
\end{aligned}$$

L'algorithme 2 résume la construction effective d'un système de décomposition CFSZ tel

1. Découpage en trois éléments : un bloc de surface maximale et deux sous-matrices triangulaires.

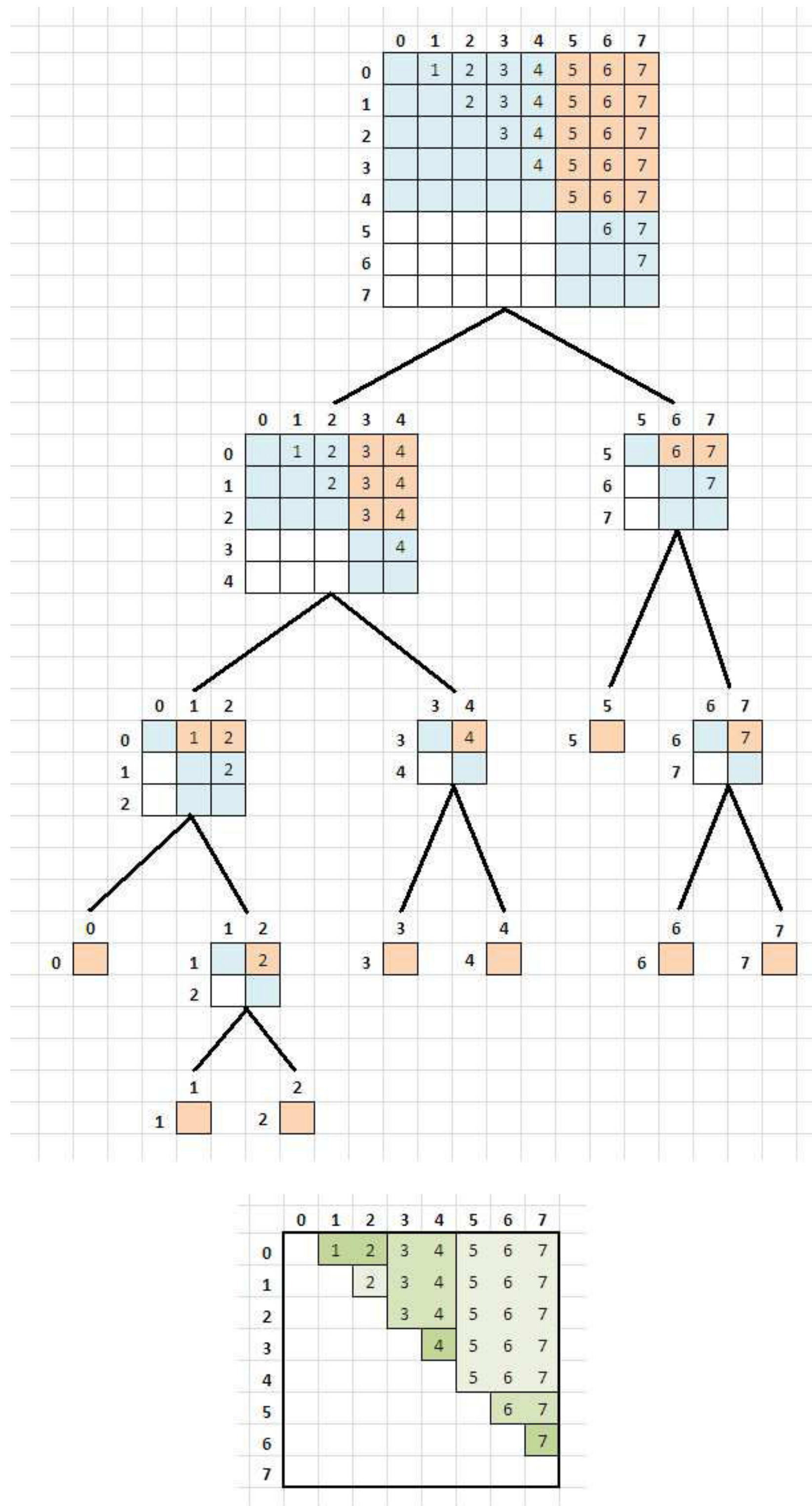


FIGURE 3.10 – Voici un exemple plus détaillé qui montre toutes les étapes d'une décomposition possible en CFSZ pour l'automate triangulaire $\mathcal{A}_6$.

que décrit ci-dessus.

Algorithme 2 : CFSZ_Itératif(n)

```
Require: n ∈ ℕ
Ensure: Z(A_n)
 1: Q ← {0, 1, 2, 3, 4, ..., n}
 2: for i = 0 to n do
 3:    follow(i) ← {j ∈ Q | j > i} ∪ {n + 1}
 4:    dec(i) ← ϕ
 5:    Q_i ← ϕ
 6: end for
 7: for i = 0 to n do
 8:    Choisir j dans Q
 9:    Q ← Q \ {j}
10:    Q_j ← follow(j)
11:    for all k ∈ Q do
12:       if (Q_j ⊆ follow(k)) then
13:          dec(k) ← dec(k) ∪ {Q_j}
14:          follow(k) ← follow(k) \ Q_j
15:       end if
16:    end for
17: end for
```

On remarque qu'il s'agit d'un algorithme non-déterministe. En effet, les différentes choix possibles des valeurs de j sur la ligne 8 de l'algorithme conduisent à des systèmes de partitions CFSZ différents (la ligne 8 correspond au choix d'un élément sur la première diagonale de la table de transitions).

Notre façon de construire un système de décomposition CFSZ, basée sur un découpage particulier de la matrice de transition, peut être décrite aussi par un algorithme récursif.

Algorithme 3 : CFSZ_Récursif(n_1, n_2)

```
Require: n_1, n_2 ∈ ℕ
Ensure: Z(A_n) pour n_1 = 0 et n_2 = n
 1: if n_1 ≤ n_2 then
 2:    Choisir un entier j entre n_1 et n_2, j ∈ {n_1, ..., n_2}
 3:    Q_j ← {j + 1, ..., n_2 + 1}
 4:    for k = n_1 to j do
 5:       dec(k) ← dec(k) ∪ {Q_j}
 6:    end for
 7:    CFSZ_Récursif(n_1, j − 1)
 8:    CFSZ_Récursif(j + 1, n_2)
 9: end if
```

L'algorithme 3 est une version récursive de l'algorithme 2. Son premier appel est effectué par CFSZ_Récursif$(0, n)$ qui correspond au découpage de la totalité de la table de transition. L'appel CFSZ_Récursif(n_1, n_2) correspond au découpage de la sous-matrice carrée dont la première diagonale est $[n_1 \cdots n_2]$, le segment diagonal qui est entre la case $M^{n+2}[n_1 - 1, n_1]$

et la case $M^{n+2}[n_2 - 1, n_2]$.

Là aussi, on remarque qu'il s'agit d'un algorithme non-déterministe. En effet, les différentes choix possibles des valeurs de j sur la ligne 2 de l'algorithme conduisent à des systèmes de partitions CFSZ différents (la ligne 2 correspond au choix d'un élément sur la première diagonale de la table de transitions).

Il est clair que notre manière de construire un système CFSZ, en particulier celle décrite par le processus de découpage récursif de l'algorithme 3, peut être représentée par un arbre binaire. La racine de cet arbre correspond à la table de transition (représentée sous forme d'une matrice carrée). Le fils gauche et le fils droit de cette racine correspondent aux deux sous-matrices résultant du découpage (choix d'un élément sur la diagonale). Ceci nous permet de donner une définition formelle de nos systèmes CFSZ basée sur les arbres binaires, et qui permet d'établir une bijection entre $\mathrm{CFSZ}(n)$, l'ensemble de tous les systèmes CFSZ possibles de l'automate triangulaire $\mathcal{A}_n$, et l'ensemble des arbres binaires complets à $n+2$ feuilles.

Un *arbre binaire* est une structure définie sur un ensemble fini de nœuds qui est soit vide, soit constitué de trois ensembles disjoints de nœuds :

1. un nœud *racine*,
2. un arbre binaire appelé le *sous-arbre gauche*,
3. un arbre binaire appelé le *sous-arbre droit*.

L'arbre binaire ne contenant aucun nœud est appelé *arbre vide*. Si le sous-arbre gauche est non vide, sa racine est appelé *fils gauche* de la racine de l'arborescence. De même, La racine du sous-arbre droit est appelé *fils droit*. Par conséquent, dans un *arbre binaire complet* chaque nœud est soit une feuille soit de degré 2. Dans ce qui suit, nous appelons un *n-arbre* un arbre binaire complet avec n feuilles. Il y a un unique n-arbre pour $n = 0$ à 2.

Soit t un n-arbre et soit π un chemin dans t. Le *poids gauche* (resp. *poids droit*) a_π (resp. b_π) est défini comme étant le nombre d'arêtes gauches (resp. droites) dans le chemin π. La *longueur* de π, notée $l_\pi = a_\pi + b_\pi$, est la longueur du chemin π. Notons $w_\pi = a_\pi b_\pi$ le *poids* de π. Le *coût* c_π de π est la somme de son poids et sa longueur. Nous avons donc $c_\pi = w_\pi + l_\pi$.

Soit ν un nœud dans t. On note π_ν le chemin depuis le nœud ν à la racine de t. On note ν_l (resp. ν_r) le fils gauche de ν (resp. le fils droit de ν). On note f_ν le père [2] de ν. Si π est un chemin à partir du nœud ν à la racine de t alors nous noterons f_π le chemin depuis le nœud f_ν à la racine de t. Nous associons également a_{π_ν}, b_{π_ν}, w_{π_ν}, l_{π_ν} et c_{π_ν} au nœud ν et on les note respectivement a_ν, b_ν, w_ν, l_ν et c_ν. L'ensemble des feuilles d'un arbre t sera noté L_t. Le poids $w(t)$ de l'arbre t est défini comme la somme des poids de ses feuilles, qui est $w(t) = \sum_{\nu \in L_t} w_\nu$.

Définition 38 (Le système CFSZ)**.** Pour le 1-arbre, le système de CFSZ correspondant consiste en $\mathrm{dec}(0) = \emptyset$. Pour le 2-arbre, le système de CFSZ correspondant consiste en $\mathrm{dec}(0) = \{\{1\}\}$ et $\mathrm{dec}(1) = \emptyset$. Soit t un $(n+2)$-arbre avec $n > 0$ et soient t_l et t_r les sous-arbres gauche et droit de t. On numérote les feuilles de t par $0, 1, \ldots, n+1$ de gauche à droite. On définit le paramètre k comme étant le numéro de la feuille la plus à gauche dans le sous-arbre droit. La translation d'une décomposition $\mathrm{dec}(i)$ par k est la décomposition $\mathrm{dec}(i+k)$ obtenue en ajoutant k à chaque élément dans les sous-ensembles de $\mathrm{dec}(i)$. Supposons que les

2. Le premier ancêtre.

feuilles de 0 à $k-1$ sont dans t_l tandis que les feuilles de k à $n+1$ sont dans t_r. Soient $C(t_l)$ et $C(t_r)$ les systèmes CFSZ correspondant aux t_l et t_r respectivement. Nous ajoutons à chaque décomposition dans $C(t_l)$ le sous-ensemble $\{k, k+1, \ldots, n, n+1\}$. Les décompositions de $C(t_r)$ devraient d'abord être translatées par k, ensuite ajoutées aux décompositions modifiées de $C(t_l)$. C'est cette union qui constitue le système CFSZ correspondant à t.

Avant de s'appuyer sur cette dernière définition pour détailler notre algorithme de réduction basé sur les arbres, nous allons d'abord prouver quelques propositions utiles pour la suite.

Proposition 3. *Le nombre de systèmes de partitions $Z(\mathcal{A}_n)$ est égal au $(n+1)^{ième}$ nombre de Catalan $\mathbb{C}_{n+1}$: $|\,\mathrm{CFSZ}(n)| = \frac{1}{n+2}\binom{2n+2}{n+1}$.*

Démonstration. L'automate $\mathcal{A}_{-1}$ possède un système CFSZ unique. Même chose pour l'automate $\mathcal{A}_0$. On a $|\,\mathrm{CFSZ}(-1)| = 1 = \mathbb{C}_0$ et $|\,\mathrm{CFSZ}(0)| = 1 = \mathbb{C}_1$. Supposons que la proposition 3 est vraie pour tout $k < n$ et prouvons que la proposition est vraie pour n. D'après la définition 38, un système CFSZ de CFSZ(n) est obtenu en reliant deux systèmes CFSZ l'un de CFSZ($k-2$) et l'autre de CFSZ($n-k$) respectivement, pour un certain k, $1 \leq k \leq n+1$. Donc :

$$\begin{aligned}|\,\mathrm{CFSZ}(n)| &= \sum_{k=1}^{n+1} |\,\mathrm{CFSZ}(k-2)| \times |\,\mathrm{CFSZ}(n-k)| \\ &= \sum_{i=0}^{n} |\,\mathrm{CFSZ}(i-1)| \times |\,\mathrm{CFSZ}(n-i-1)| \\ &= \sum_{i=0}^{n} \mathbb{C}_i\,\mathbb{C}_{n-i} \quad \text{la relation de récurrence du nombre de Catalan} \\ &= \mathbb{C}_{n+1}.\end{aligned}$$

□

Proposition 4. *Soit $Z(\mathcal{A}_n)$ un système CFSZ qui correspond à un $(n+2)$-arbre t donné. Le nombre de transitions dans l'automate réduit $\mathcal{C}_{Z(\mathcal{A}_n)}$ est égal au poids de t, soit : $T_{\mathcal{C}_{Z(\mathcal{A}_n)}} = w(t)$.*

Démonstration. D'après le lemme 5 nous avons $T_{\mathcal{C}_{Z(\mathcal{A}_n)}} = \sum_{q\in Q} a(q)b(q)$. Nous avons également :

$$\begin{aligned}w(t) &= \sum_{q=0}^{n+1} a_{\nu_q} b_{\nu_q} \quad \text{où } \nu_q \text{ est la feuille marquée par } q \text{ dans } t \\ &= \sum_{q=0}^{n} a_{\nu_q} b_{\nu_q} \quad \text{parce que } a_{\nu_{n+1}} = 0.\end{aligned}$$

Donc, pour prouver la dernière proposition, il suffit de prouver que :

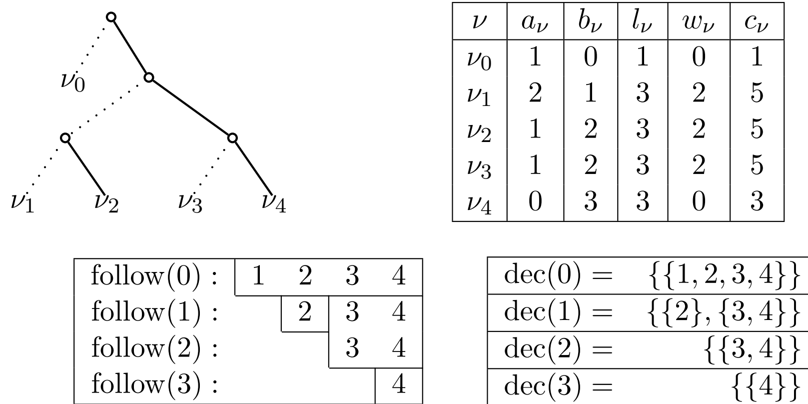
$$a(q) = a_{\nu_q} \quad \text{pour } 0 \leq q \leq n \tag{3.1}$$

$$b(q) = b_{\nu_q} \quad \text{pour } 0 \leq q \leq n \tag{3.2}$$

Par construction, d'après la définition 38, nous pouvons vérifier que les égalités (3.1) et (3.2) sont vraies pour le 1-arbre et le 2-arbre. Supposons maintenant que les égalités (3.1) et (3.2) sont vraies pour les deux sous-arbres t_l et t_r. Lorsque nous unissons t_l et t_r pour obtenir t, nous ajoutons aux décompositions qui correspondent à t_l le sous-ensemble $\{k, k+1, \ldots, n, n+1\}$, ce qui signifie que :

1. On incrémente d'un cran le nombre d'arêtes gauches dans chaque chemin de t_l et en même temps, nous ajoutons un nouveau sous-ensemble dans les décompositions correspondantes de t_l, ce qui incrémente d'une unité la taille de ces décompositions. Ainsi, l'égalité (3.1) reste vraie pour t.
2. On incrémente d'un cran le nombre d'arêtes droites dans chaque chemin de t_r et en même temps, on incrémente d'une unité le nombre d'occurrences de chaque $q \in \{k, k+1, \ldots, n, n+1\}$ dans les décompositions correspondantes de t_r. Ainsi, l'égalité (3.2) reste vraie pour t.

□



ν	a_ν	b_ν	l_ν	w_ν	c_ν
ν_0	1	0	1	0	1
ν_1	2	1	3	2	5
ν_2	1	2	3	2	5
ν_3	1	2	3	2	5
ν_4	0	3	3	0	3

follow(0) :	1	2	3	4
follow(1) :		2	3	4
follow(2) :			3	4
follow(3) :				4

dec(0) =	$\{\{1,2,3,4\}\}$
dec(1) =	$\{\{2\},\{3,4\}\}$
dec(2) =	$\{\{3,4\}\}$
dec(3) =	$\{\{4\}\}$

FIGURE 3.11 – Un 5-arbre binaire complet et un système de partition $Z(\mathcal{A}_3)$ associé (les arêtes gauches sont représentées par des lignes en pointillés et les arêtes droites avec des lignes continues).

Nous avons essayé, dans cette section, de décrire selon divers points de vue nos systèmes CFSZ qui sont des systèmes de décompositions particuliers, qui peuvent être obtenus à partir du découpage en blocs de la matrice de transition, qui peut être décrit par un algorithme récursif, itératif ou encore une définition qui met en évidence la correspondance entre nos décompositions et les arbres binaires complets.

Notre objectif, dans les sections suivantes, est de calculer toutes les décompositions minimales en CFSZ(n). Pour ce faire, une approche naïve consiste à engendrer tous les systèmes CFSZ pour l'automate $\mathcal{A}_n$ qui peut être effectué en un temps factoriel. En effet, le nombre total des systèmes CFSZ pour l'automate $\mathcal{A}_n$ est égal à $\mathbb{C}_{n+1}$, le $(n+1)^{\text{ième}}$ nombre de Catalan.

Proposition 5. *Le calcul de tous les systèmes de partitions minimaux* $Z(\mathcal{A}_n)$ *dans* CFSZ(n) *peut être effectué en temps factoriel* $O(n!)$.

Démonstration. Cela peut être effectué en appelant l'algorithme non-déterministe 2 ou 3 pour toutes les exécutions possibles. Par exemple pour l'algorithme 2 il y a $n!$ exécutions possibles qui correspondent aux différent choix successifs des valeurs de j (ligne 8). □

3.2.3 Réduction polynomiale pour les automates $\mathcal{A}_n$

En utilisant la programmation dynamique, nous pouvons accélérer la recherche en force brute du temps factoriel $O(n!)$ pour obtenir l'algorithme polynomial 4.

Algorithme 4 : CFSZ_Dynamique(n)

Require: $n \in \mathbb{N}$
Ensure: Un tableau tridimensionnel Z

```
1:  {Initialisation}
2:  for p = 0 à n do
3:     for q = 0 à n do
4:        Z[1][p][q] ← p * q
5:     end for
6:  end for
7:  {Main}
8:  for i = 1 à n do
9:     for p = 0 à n do
10:       for q = 0 à n do
11:          for k = 0 à i do
12:             T[k] ← Z[k][p + 1][q] + Z[i − k − 1][p][q + 1]
13:          end for
14:          Z[i][p][q] ← min_{k∈{0,...,i}} T[k]
15:       end for
16:    end for
17: end for
```

Théorème 15. *L'algorithme 4 calcule tous les systèmes de partitions minimaux $Z(\mathcal{A}_i)$ dans* CFSZ(i) *pour tous les $i = 1$ à n en temps polynomial $O(n^4)$ et en espace cubique $O(n^3)$.*

Démonstration. Ici nous utilisons la notation de la barre : $n = \overline{i, j}$ signifie $n = i$ à j.

Dans la suite on détaillera comment l'algorithme 4 a été obtenu à partir de l'algorithme 3 en appliquant les principes de la programmation dynamique sur les algorithmes récursifs.

Principe de base. Soit M^n une matrice triangulaire supérieure $n \times n$. Pour chaque ligne i de M^n, on associe un nombre naturel a_i qui représente le nombre de blocs de la ligne i. Pour chaque colonne j de M^n, on associe un nombre naturel b_j qui représente le nombre de blocs dans la colonne j.

L'algorithme de décomposition récursive 3 choisit à chaque étape un bloc B_k de taille maximale qui possède comme coin inférieur gauche l'élément $M^n[k+1, k]$ et $M^n[n, 1]$ comme coin supérieur droit. Ceci décompose la matrice M^n en :

1. un bloc B_k de taille $(n-k)\times k$,
2. une sous-matrice triangulaire supérieure M_a^k de taille $k\times k$,
3. une sous-matrice triangulaire supérieure $M_b^{(n-k)}$ de taille $(n-k)\times(n-k)$.

Supposons maintenant que la matrice M_a^k a une décomposition analogue et que les a'_i représentent le nombre de blocs horizontaux relatifs aux lignes de la matrice M_a^k (respectivement les b'_j représentent le nombre de blocs verticaux relatifs aux colonnes de la matrice M_a^k).

De la même manière, supposons que la matrice $M_b^{(n-k)}$ a une décomposition analogue et que les a''_i représentent le nombre de blocs horizontaux relatifs aux lignes de la matrice $M_b^{(n-k)}$ (respectivement les b''_j représentent le nombre de blocs verticaux relatifs aux colonnes de la matrice $M_b^{(n-k)}$).

Dans ce cas, les a_i et les b_j peuvent être facilement calculés comme suit :

$$a_i = \begin{cases} a'_i + 1 & i = \overline{1,k} \\ a''_{i-k} & i = \overline{k+1,n} \end{cases}$$

$$b_j = \begin{cases} b'_j & j = \overline{1,k} \\ b''_{j-k} + 1 & j = \overline{k+1,n} \end{cases}$$

Pour $p,q\in\mathbb{N}$ on note $T(M^n,p,q)$ la somme :

$$T(M^n,p,q) = \sum_{i=1}^{n}(a_i+p)(b_i+q)$$

et $T(M^n,p,q,k)$ la somme :

$$T(M^n,p,q,k) = \sum_{i=1}^{k}(a_i+p)(b_i+q) + \sum_{i=k+1}^{n}(a_i+p)(b_i+q)$$

L'objectif principal est de minimiser la somme $\sum\limits_{i=1}^{n} a_ib_i$ représentant le nombre de transitions (voir le lemme 5) :

$$\begin{aligned}
\min\left(\sum_{i=1}^{n} a_ib_i\right) &= \min T(M^n,0,0) \\
&= \min_{k=\overline{1,n}} T(M^n,0,0,k) \\
&= \min_{k=\overline{1,n}} \left(\sum_{i=1}^{k} a_ib_i + \sum_{i=k+1}^{n} a_ib_i\right) \\
&= \min_{k=\overline{1,n}} \left(\sum_{i=1}^{k} (a'_i+1)b'_i + \sum_{i=k+1}^{n} a''_{i-k}(b''_{i-k}+1)\right) \\
&= \min_{k=\overline{1,n}} \left(\sum_{i=1}^{k} (a'_i+1)b'_i + \sum_{i=1}^{n-k} a''_i(b''_i+1)\right) \\
&= \min_{k=\overline{1,n}} \left(\sum_{i=1}^{k} (a'_i+1)(b'_i+0) + \sum_{i=1}^{n-k} (a''_i+0)(b''_i+1)\right) \\
&= \min_{k=\overline{1,n}} \left(T(M_a^k,1,0) + T(M_b^{(n-k)},0,1)\right)
\end{aligned}$$

Étant donné le choix d'un bloc B_k nous avons :

$$\begin{aligned}\min\left(\sum_{i=1}^{n} a_i b_i\right)_k &= \min T(M_a^k, 1, 0) + \min T(M_b^{(n-k)}, 0, 1)\\ &= \min_{k_1=\overline{1,k}} T(M_a^k, 1, 0, k_1) + \min_{k_2=\overline{1,n-k}} T(M_b^{(n-k)}, 0, 1, k_2)\end{aligned}$$

Donc, pour minimiser $T(M^n, 0, 0)$ par rapport à un entier k fixé, il nous faut trouver indépendamment :

– la décomposition de la matrice M_a^k minimisant $T(M_a^k, 1, 0)$,
– la décomposition de la matrice $M_b^{(n-k)}$ minimisant $T(M_b^{(n-k)}, 0, 1)$.

De même :

$$\begin{aligned}\min\left(\sum_{i=1}^{n} (a_i+1) b_i\right) &= \min T(M^n, 1, 0)\\ &= \min_{k=\overline{1,n}} T(M^n, 1, 0, k)\\ &= \min_{k=\overline{1,n}} \left(\sum_{i=1}^{k} (a_i+1)b_i + \sum_{i=k+1}^{n} (a_i+1)b_i\right)\\ &= \min_{k=\overline{1,n}} \left(\sum_{i=1}^{k} (a'_i+2)b'_i + \sum_{i=k+1}^{n} (a''_{i-k}+1)(b''_{i-k}+1)\right)\\ &= \min_{k=\overline{1,n}} \left(\sum_{i=1}^{k} (a'_i+2)b'_i + \sum_{i=1}^{n-k} (a''_i+1)(b''_i+1)\right)\\ &= \min_{k=\overline{1,n}} \left(\sum_{i=1}^{k} (a'_i+2)(b'_i+0) + \sum_{i=1}^{n-k} (a''_i+1)(b''_i+1)\right)\end{aligned}$$

Étant donné le choix d'un bloc B_k, nous avons :

$$\begin{aligned}\min\left(\sum_{i=1}^{n} (a_i+1) b_i\right)_k &= \min T(M_a^k, 2, 0) + \min T(M_b^{(n-k)}, 1, 1)\\ &= \min_{k_1=\overline{1,k}} T(M_a^k, 2, 0, k_1) + \min_{k_2=\overline{1,n-k}} T(M_b^{(n-k)}, 1, 1, k_2)\end{aligned}$$

Donc, pour minimiser $T(M^n, 1, 0)$ par rapport à un entier k fixé, il nous faut trouver indépendamment :

– la décomposition de la matrice M_a^k minimisant $T(M_a^k, 2, 0)$,
– la décomposition de la matrice $M_b^{(n-k)}$ minimisant $T(M_b^{(n-k)}, 1, 1)$.

Et :

$$\begin{aligned}
\min\left(\sum_{i=1}^{n} a_i(b_i+1)\right) &= \min T(M^n,0,1)\\
&= \min_{k=\overline{1,n}} T(M^n,0,1,k)\\
&= \min_{k=\overline{1,n}}\left(\sum_{i+1}^{k} a_i(b_i+1)+\sum_{i=k+1}^{n} a_i(b_i+2)\right)\\
&= \min_{k=\overline{1,n}}\left(\sum_{i+1}^{k} (a'_i+1)(b'_i+1)+\sum_{i=k+1}^{n} a''_{i-k}(b''_{i-k}+2)\right)\\
&= \min_{k=\overline{1,n}}\left(\sum_{i+1}^{k} (a'_i+1)(b'_i+1)+\sum_{i+1}^{n-k} a''_{i}(b''_{i}+2)\right)\\
&= \min_{k=\overline{1,n}}\left(\sum_{i+1}^{k} (a'_i+1)(b'_i+1)+\sum_{i+1}^{n-k} a''_{i}(b''_{i}+2)\right)
\end{aligned}$$

Étant donné le choix d'un bloc B_k, nous avons :

$$\begin{aligned}
\min\left(\sum_{i=1}^{n} a_i(b_i+1)\right)_k &= \min T(M_a^k,1,1)+\min T(M_b^{(n-k)},0,2)\\
&= \min_{k_1=\overline{1,k}} T(M_a^k,1,1,k_1)+\min_{k_2=\overline{1,n-k}} T(M_b^{(n-k)},0,2,k_2)
\end{aligned}$$

Donc, pour minimiser $T(M^n,0,1)$ par rapport à un entier k fixé, il nous faut trouver indépendamment :

- la décomposition de la matrice M_a^k minimisant $T(M_a^k,1,1)$,
- la décomposition de la matrice $M_b^{(n-k)}$ minimisant $T(M_b^{(n-k)},0,2)$.

En général :

$$\begin{aligned}
\min\left(\sum_{i=1}^{n} (a_i+p)(b_i+q)\right) &= \min T(M^n,p,q)\\
&= \min_{k=\overline{1,n}} T(M^n,p,q,k)\\
&= \min_{k=\overline{1,n}}\left(\sum_{i=1}^{k} (a_i+p)(b_i+q)+\sum_{i=k+1}^{n} (a_i+p)(b_i+q+1)\right)\\
&= \min_{k=\overline{1,n}}\left(\sum_{i=1}^{k} (a'_i+p+1)(b'_i+q)+\sum_{i=k+1}^{n} (a''_{i-k}+p)(b''_{i-k}+q+1)\right)\\
&= \min_{k=\overline{1,n}}\left(\sum_{i=1}^{k} (a'_i+p+1)(b'_i+q)+\sum_{i=1}^{n-k} (a''_{i}+p)(b''_{i}+q+1)\right)\\
&= \min_{k=\overline{1,n}}\left(\sum_{i=1}^{k} (a'_i+p+1)(b'_i+q)+\sum_{i=1}^{n-k} (a''_{i}+p)(b''_{i}+q+1)\right)
\end{aligned}$$

Étant donné le choix d'un bloc B_k nous avons :

$$\begin{aligned}
\min\left(\sum_{i=1}^{n} (a_i+p)(b_i+q)\right)_k &= \min T(M_a^k,p+1,q)+\min T(M_b^{(n-k)},p,q+1)\\
&= \min_{k_1=\overline{1,k}} T(M_a^k,p+1,q,k_1)+\min_{k_2=\overline{1,n-k}} T(M_b^{(n-k)},p,q+1,k_2)
\end{aligned} \tag{3.3}$$

Donc, pour minimiser $T(M^n, p, q)$ par rapport à un entier k fixé, il nous faut trouver indépendamment :

- la décomposition de la matrice M_a^k minimisant $T(M_a^k, p+1, q)$,
- la décomposition de la matrice $M_b^{(n-k)}$ minimisant $T(M_b^{(n-k)}, p, q+1)$.

Enfin, l'optimisation de la somme $\sum_{i=1}^{n} a_i b_i$ par rapport à un ordre n nécessite le calcul préalable et l'optimisation de toutes les sommes de la forme :

$$\sum_{i=1}^{m} (a_i + p)(b_i + q)$$

pour tout ordre m et deux nombres entiers p, q strictement inférieurs à n. Fondamentalement, ceci est considéré comme un problème d'optimisation à trois paramètres :

$$\begin{cases} m & = \overline{1, n-1} \\ p & = \overline{0, n-1} \\ q & = \overline{0, n-1} \end{cases}$$

Pour traduire ce principe comme algorithme d'optimisation, nous avons besoin d'un tableau $Z[n][n][n]$ de taille $n \times n \times n$:

$$Z[x][y][z] = \min T(M^x, y, z)$$

$n = 1$ **:** Pour cet ordre, la matrice M^n notamment M^1 admet une décomposition unique pour laquelle seuls $a_1 = 0$ et $b_1 = 0$, cependant :

$$\begin{aligned} \min\left(\sum_{i=1}^{n} (a_i + p)(b_i + q)\right) &= \min\left(\sum_{i=1}^{1} (a_i + p)(b_i + q)\right) \\ &= \min\left((a_1 + p)(b_1 + q)\right) \\ &= \min(p \times q) \\ &= p \times q \end{aligned}$$

Cette valeur de n correspond exactement à l'étape d'initialisation de l'algorithme.

$n \geq 2$ **:** Supposons maintenant que nous avons déjà calculé toutes les sommes de la forme :

$$\sum_{i=1}^{m} (a_i + p)(b_i + q)$$

pour chaque m et tous p, q strictement inférieurs à n, qui sont stockés dans le tableau Z. On a seulement besoin d'utiliser ces valeurs et l'équation 3.3 pour compléter les valeurs de Z pour l'ordre n.

Listing 3.1 – Code en C++ de l'algorithme dynamique

```
int Z[n][n][n];
int T[n];
void Init(){
 for(int p=0;p<n;p++)
 for(int q=0;q<n;q++)
 Z[1][p][q]=p*q;
}
void main(){
 Init();
 for(int i=1;i<n;i++)
 for(int p=0;p<n;p++)
 for(int q=0;q<n;q++){
    for(int k=0;k<i;k++) T[k]=Z[k][p+1][q]+Z[i-k-1][p][q+1];
    int min=T[0];
    for(int k=0;k<i;k++) if(T[k]<min) min=T[k];
    Z[i][p][q]=min;
 }
}
```

Le programme 5.1 est la version en code C++ de l'algorithme 4. Cet algorithme est polynomial en temps d'exécution $O(n^4)$ et cubique en espace mémoire $O(n^3)$. □

Nous allons présenter, dans la section qui suit, notre deuxième algorithme, qui est basé sur les arbres. Il calcule efficacement des systèmes $Z(\mathcal{A}_n)$ minimaux.

3.2.4 Réduction quasi-linéaire pour les automates $\mathcal{A}_n$

Nous montrons ici comment un arbre binaire peut être utilisé pour calculer un système de partitions particulier qui sera utilisé pour construire un automate réduit.

La proposition 4 affirme que chaque système de partitions $Z(\mathcal{A}_n)$ correspond à un unique n-arbre t, ce qui correspond à un automate réduit avec $w(t)$ transitions. Donc, trouver un système de partitions minimal $Z(\mathcal{A}_n)$ se réduit à trouver un n-arbre ayant un poids minimal. Afin de calculer efficacement un système de décomposition minimal de CFSZ(n), nous allons utiliser les arbres binaires complets correspondants. En effet, le calcul d'un système $Z(\mathcal{A}_n)$ minimal se réduit à la recherche d'un $(n+2)$-arbre ayant un poids minimal.

Soit $\mathrm{Split}(t)$ la fonction qui renvoie l'arbre obtenu à partir de t en substituant une feuille ayant un coût minimal en t par l'unique 2-arbre.

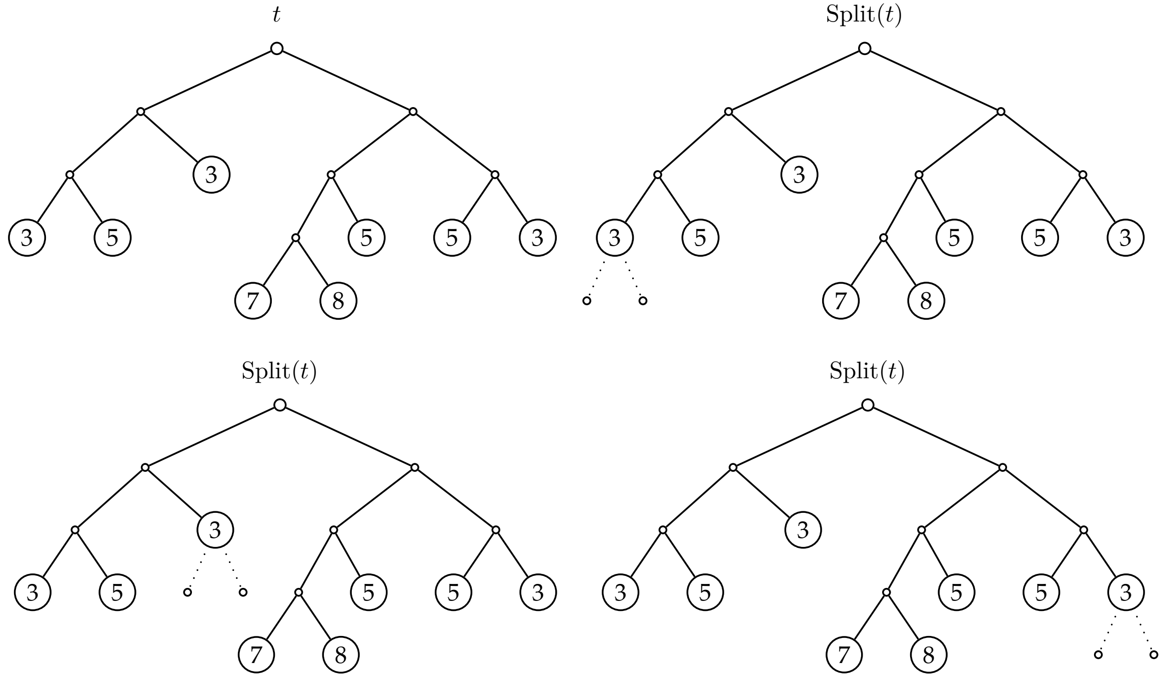


FIGURE 3.12 – Un 8-arbre t avec ses Split arbres possibles. Les valeurs dans les nœuds sont les coûts.

Appelons Z-arbre de rang n un n-arbre de poids minimal.

Proposition 6. *La suite des Z-arbres peut être calculée inductivement comme suit :*
- *le Z-arbre de rang un est le* 1*-arbre,*
- *si t est un Z-arbre (de rang i) alors* $\mathrm{Split}(t)$ *est un Z-arbre (de rang* $i+1$*).*

Démonstration. Soit t_n un Z-arbre de rang n. Pour obtenir un arbre t_{n+1} de rang $n+1$ à partir de t_n nous devons éclater une feuille μ. Le poids de t_{n+1} est :

$$\begin{aligned} w(t_{n+1}) &= \sum_{\nu \in L_{t_{n+1}}} w_\nu \\ &= \left(\sum_{\nu \in L_{t_n}} w_\nu \right) - w_\mu + w_{\mathrm{filsGauche}(\mu)} + w_{\mathrm{filsDroit}(\mu)} \\ &= w(t_n) - a_\mu b_\mu + (a_\mu + 1) b_\mu + a_\mu (b_\mu + 1) \\ &= w(t_n) + c_\mu \end{aligned}$$

Si μ est la feuille de t_n qui a un coût minimal alors, l'arbre t_{n+1} aura un poids minimal. □

Ainsi, cette construction inductive nous permet d'avoir l'arbre de poids minimal. La différence de poids entre deux arbres minimaux consécutifs est exactement le coût de la feuille éclatée. Il est clair que, pour un n donné, il peut exister plusieurs Z-arbres de rang n. Tous

les Z-arbres de rang inférieur à n peuvent être générés par l'algorithme suivant :

Algorithme 5 : min-Z-arbre(n)

Require: $n \in \mathbb{N}$
Ensure: $\mathrm{resultat}$: un ensemble de Z-arbres

```
1: resultat ← ∅
2: t ← 1-arbre
3: for i = 1 to n do
4:    t ← Split(t)
5:    ajouter t à resultat
6: end for
```

Théorème 16. *L'algorithme min-Z-arbre calcule un Z-arbre de rang i pour tout i de 1 à n en une complexité temporelle en $O(n \log_2 n)$.*

Démonstration. À chaque étape de l'algorithme, nous recherchons une feuille ayant un coût minimal, ensuite nous l'éclatons. Nous pouvons maintenir les coûts des feuilles dans une structure dynamique qui nous permet, en un temps logarithmique :

– de localiser les feuilles de coût minimal,
– d'insérer les nouvelles feuilles obtenues à partir de la fonction Split.

□

Afin d'étudier la complexité de la taille des automates réduits (le nombre de transitions), nous introduisons une sous-classe d'arbres binaires complets (appelé P-arbres), pour lesquels les Z-arbres sont uniques.

3.2.4.1 Les P-arbres

Le but de cette section est de définir une sous-classe de Z-arbres permettant d'étudier le nombre de transitions de nos automates minimaux.

On note M_t l'ensemble des feuilles ayant un coût minimal dans t, à savoir :

$$M_t = \arg\min_{\nu \in L_t} c_\nu.$$

La fonction $\mathrm{SplitAll}(t)$ renvoie l'arbre obtenu à partir de t en substituant chaque feuille dans M_t par l'unique 2-arbre. Voir la figure 3.13.

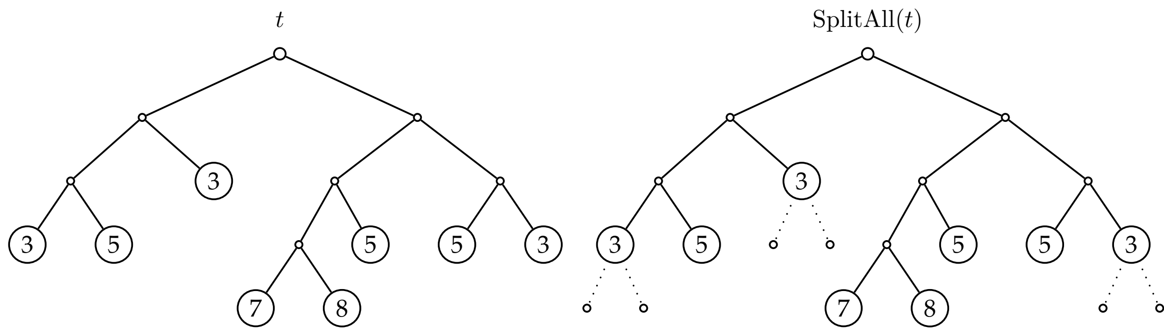


FIGURE 3.13 – Un 8-arbre t avec son SplitAll arbre. Les valeurs dans les nœuds sont les coûts.

Définition 39. La classe $(t_n)_{n>0}$ de P-arbres est définie par induction comme suit :
- t_1 est le 1-arbre,
- $t_{(n+1)} = \mathrm{SplitAll}(t_n)$.

Remarque 2. Notez que si $\nu \in M_{t_n}$ alors $c_\nu = n - 1$. Ceci peut être prouvé par induction sur n. Par conséquent, pour obtenir $t_{(n+1)}$ à partir de t_n nous éclatons les feuilles du coût $n - 1$.

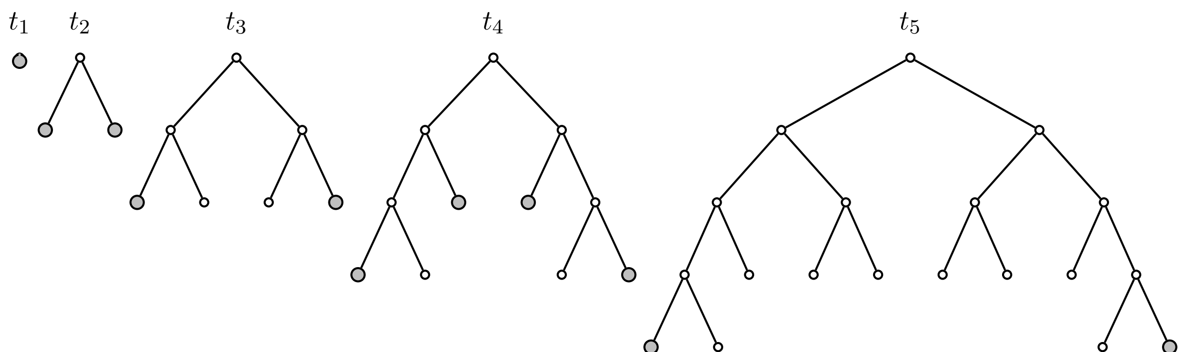


FIGURE 3.14 – Les cinq premiers P-arbres. Les feuilles avec grand cercle ont un coût minimal.

Afin d'évaluer le nombre de transitions de l'automate réduit produit par notre construction basée sur les systèmes CFSZ, nous aurons besoin d'établir une bijection entre les Z-arbres et un objet combinatoire appelé *triangle d'Ératosthène - Pascal*.

3.2.4.2 Triangle d'Ératosthène - Pascal

Le triangle d'Ératosthène - Pascal est construit à partir du triangle arithmétique comme suit : on entrelace chaque colonne k du triangle arithmétique avec $(k - 1)$ zéros. L'élément du triangle d'Ératosthène - Pascal à la $n^{\text{ième}}$ ligne et la $k^{\text{ième}}$ colonne est noté T_n^k avec $n \geq 1$ et $k \geq 1$.

	1	2	3	4	5	6	7	8	9	10	···
1	1										
2	1	1									
3	1	0	1								
4	1	2	0	1							
5	1	0	0	0	1						
6	1	3	3	0	0	1					
7	1	0	0	0	0	0	1				
8	1	4	0	4	0	0	0	1			
9	1	0	6	0	0	0	0	0	1		
10	1	5	0	0	5	0	0	0	0	1	
11	1	0	0	0	0	0	0	0	0	0	1

Soit n un nombre naturel. On note D_n l'ensemble des diviseurs de n.

Proposition 7.

$$T_n^k = \begin{cases} \binom{\frac{n}{k}+k-2}{k-1} & \textit{si } k \in D_n \\ 0 & \textit{sinon.} \end{cases}$$

Démonstration. L'élément du triangle arithmétique à la $n^{\text{ième}}$ ligne et la $j^{\text{ième}}$ colonne est $\binom{i-1}{j-1}$. Cet élément est déplacé dans le triangle d'Ératosthène - Pascal à la ligne $r = (i-j+1)j$ dans la même colonne j. Nous avons donc :

$$T^j_{(i-j+1)j} = \binom{i-1}{j-1}. \quad \text{Ainsi} \quad T^j_r = \binom{\frac{r}{j}+j-2}{j-1}.$$

□

3.2.4.3 Les P-arbres et le triangle d'Ératosthène - Pascal

Nous allons décrire dans cette section le lien entre le triangle d'Ératosthène - Pascal et l'ensemble des P-arbres. Soit S_n la somme de la $n^{\text{ième}}$ ligne du triangle d'Ératosthène - Pascal.

$$S_n = \sum_{k=1}^{n} T_n^k = \sum_{k \in D_n} \binom{\frac{n}{k}+k-2}{k-1}$$

Théorème 17. *La somme des éléments de la $n^{\text{ième}}$ ligne du triangle d'Ératosthène Pascal, S_n, est exactement $|M_{t_n}|$, le nombre de feuilles de coût minimal dans le P-arbre t_n.*

Pour démontrer ce théorème, nous introduisons une famille F_n qui est en bijection avec les lignes du triangle d'Ératosthène - Pascal ainsi qu'avec les P-arbres. Nous nous concentrons sur la question suivante : *Étant donné un entier naturel n, quels sont les chemins possibles ayant $n-1$ comme coût ?* Pour répondre à cette question, définissons F_{n-1} comme l'ensemble des chemins de coût $n-1$:

$$F_{n-1} = \{\pi \mid c_\pi = n-1\}$$

Lemme 6. *Pour tout $n \geq 1$, on a :* $|F_{n-1}| = S_n$.

Démonstration. Pour $0 \leq i \leq n-1$, soit F^i_{n-1} l'ensemble de chemins de F_{n-1} ayant exactement i arêtes gauches. On a $F_{n-1} = \bigcup_{i=0}^{n-1} F^i_{n-1}$ tel que :

$$\begin{aligned} F^i_{n-1} &= \{\pi \in F_{n-1} \mid a_\pi = i\} \\ &= \{\pi \mid (a_\pi b_\pi + a_\pi + b_\pi = n-1) \wedge (a_\pi = i)\} \\ &= \{\pi \mid (b_\pi = \frac{n}{i+1} - 1) \wedge (a_\pi = i)\} \end{aligned}$$

$$\text{Ainsi, nous obtenons } |F^i_{n-1}| = \begin{cases} \binom{\frac{n}{i+1} + i - 1}{i} & \text{if } (i+1) \in D_n \\ 0 & \text{sinon.} \end{cases}$$

Cela correspond aux différentes manières d'arranger i arêtes gauches dans un chemin de longueur $\frac{n}{i+1} + i - 1$. Soit $k = i+1$ alors :

$$|F^{k-1}_{n-1}| = \begin{cases} \binom{\frac{n}{k} + k - 2}{k-1} & \text{si } k \in D_n \\ 0 & \text{sinon.} \end{cases}$$

Ainsi, $|F^{k-1}_{n-1}| = T^k_n$. Et pour $0 \leq i < n$ nous obtenons $|F^i_{n-1}| = T^{i+1}_n$. Finalement :

$$|F_{n-1}| = \sum_{i=0}^{n-1} |F^i_{n-1}| = \sum_{i=0}^{n-1} T^{i+1}_n = \sum_{i=1}^{n} T^i_n = S_n$$

Par conséquent, nous pouvons associer la $n^{\text{ième}}$ ligne du triangle d'Ératosthène - Pascal à F_{n-1}. □

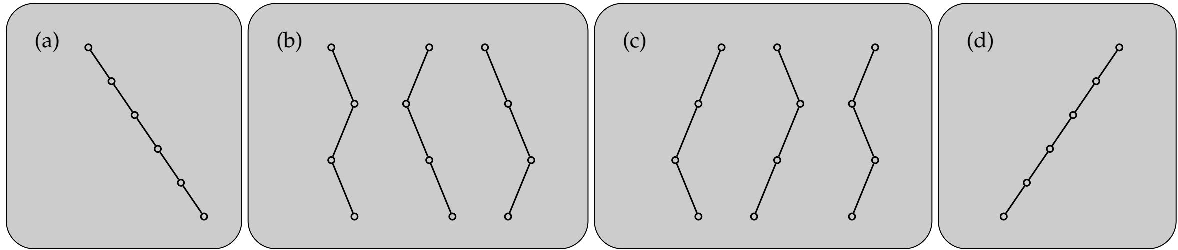


FIGURE 3.15 – Tous les chemins de coût 5. (a) Le chemin unique de coût 5 sans arête gauche, qui correspond à $\binom{5}{0} = 1$. (b) Les trois chemins de coût 5 avec une seule arête gauche, qui correspondent à $\binom{3}{1} = 3$. (c) Les trois chemins de coût 5 avec deux arêtes gauches, qui correspondent à $\binom{3}{2} = 3$. (d) Le chemin unique de coût 5 avec cinq arêtes gauches, qui correspond à $\binom{5}{5} = 1$. Il n'y a pas de chemin de coût 5 avec trois ou quatre arêtes gauches ce qui correspond aux zéros dans le 6ième ligne du triangle d'Ératosthène - Pascal.

Lemme 7. *Pour tout $n \geq 1$, on a $|F_{n-1}| = |M_{t_n}|$.*

Démonstration. Nous allons montrer que chaque ensemble F_n est associé au P-arbre t_n. Par induction sur n. Supposons que, pour tout $k \leq n$, le P-arbre t_n contient tous les chemins de coût inférieur ou égal à n, c'est-à-dire que, pour tout $k \leq n$, F_k est dans le P-arbre t_n. À partir de cette hypothèse, nous devons montrer que $t_{(n+1)} = \mathrm{SplitAll}(t_n)$ contient tous les chemins de coût inférieur ou égal à $(n+1)$. Évidemment $t_{(n+1)}$ contient tous les chemins de coût inférieur ou égal à n car l'arbre $t_{(n+1)}$ est obtenu à partir de t_n. Cependant, *est-ce que $t_{(n+1)}$ contient tous les chemins de coût égal à $(n+1)$?* Pour répondre à cette question, nous procédons par l'absurde. Soit π un chemin de coût égal à $c_\pi = (n+1)$ qui n'est pas dans $t_{(n+1)}$. Il est clair que le coût du père de π est :

$$c_{f_\pi} = \begin{cases} (a_\pi - 1)b_\pi + (a_\pi - 1) + b_\pi & \text{si } \pi \text{ est un fils gauche,} \\ a_\pi(b_\pi - 1) + a_\pi + (b_\pi - 1) & \text{si } \pi \text{ est un fils droit.} \end{cases}$$

Comme $c_{f_\pi} < c_\pi$ alors $c_{f_\pi} \leq n$. On en déduit que lorsque $c_{f_\pi} < n$, f_π le père de π a été éclaté par la fonction SplitAll dans une étape antérieure ou il sera éclaté dans l'arbre courant t_n dans le cas où $c_{f_\pi} = n$ (voir la remarque 2). Dans les deux cas, le chemin π est nécessairement dans $t_{(n+1)}$. □

À partir des deux derniers lemmes, construisons une bijection entre les P-arbres et les lignes du triangle d'Ératosthène - Pascal. Nous affirmons le corollaire suivant :

Corollaire 14. *Soit t_n un P-arbre. Alors $|M_{t_n}| = |F_{n-1}| = S_n$.*

D'après le corollaire 14, on a $S_n = \sum_{k|n} \binom{\frac{n}{k} + k - 2}{k-1}$.

Proposition 8. *Soit t_n un P-arbre. Sa taille $s(t_n) = |L_{t_n}|$ (le nombre de feuilles), et son poids $w(t_n)$ sont : $s(t_n) = 1 + \sum_{i=1}^{n-1} S_i$ et $w(t_n) = \sum_{i=2}^{n} (i-2)S_{i-1}$.*

Démonstration. D'après le corollaire 14 et la remarque 2, la taille de l'arbre t_n est la somme de la taille de l'arbre t_{n-1} et du nombre de toutes les feuilles minimales éclatées, soit $s(t_n) = s(t_{n-1}) + |M_{t_{n-1}}|$. Nous avons aussi que le poids de l'arbre t_n est la somme du poids de l'arbre t_{n-1} et des coûts de toutes les feuilles minimales éclatées, soit $w(t_n) = w(t_{n-1}) + (n-2)|M_{t_{n-1}}|$. □

D'après la proposition 4, le P-arbre t_n (qui est un Z-arbre) correspond au système de partitions $Z(\mathcal{A}_{s(t_n)})$. L'automate CFS associé à $Z(\mathcal{A}_{s(t_n)})$ possède $w(t_n)$ transitions.

3.2.5 Minimalité asymptotique

Nous avons montré dans les sections précédentes qu'un P-arbre d'ordre n est associé à un automate réduit dont le nombre d'états est donné par $s(t_n) = 1 + \sum_{1}^{n-1} S_i$ et un nombre

de transitions donné par $w(t_n) = \sum_{i=1}^{n-2} iS_{i+1}$. Nous allons montrer, dans cette section, un résultat d'évaluation asymptotique du nombre de transitions $w(t_n)$ par rapport au nombre d'états $s(t_n)$. Nous montrons notamment que le nombre de transitions des automates réduits obtenus par notre algorithme est asymptotiquement équivalent à $s(t_n)(\log_2 s(t_n))^2$. Ce qui signifie que le nombre de transitions est asymptotiquement minimal dans la mesure où nous atteignons la borne inférieure de Schnitger [Sch06]. Le théorème suivant, qui affirme ce résultat, a été obtenu grâce à une collaboration avec EL HOUCEIN EL ABDALAOUI de l'université de Rouen, France.

Théorème 18. $w(t_n) \sim s(t_n)(\log_2 s(t_n))^2$.

Avant de détailler la démonstration du théorème 18, prolongeons les $s(t_n)$ et $w(t_n)$ sur la droite des réels. Par conséquent, pour tout $x \geq 2$, nous avons :

$$s(t_x) = 1 + \sum_{1}^{\lfloor x \rfloor - 1} S_i \text{ et } w(t_x) = \sum_{i=1}^{\lfloor x \rfloor - 2} iS_{i+1}.$$

Nous rappelons que la fonction arithmétique classique $\pi(x)$ désigne le nombre de nombres premiers inférieurs à x. Nous aurons également besoin de l'identité classique suivant (dûe à Abel).

Proposition 9 (Par exemple [Apo76]). *Pour toute fonction arithmétique $a(n)$ soit :*

$$A(x) = \sum_{n \leq x} a(n)$$

tel que $A(x) = 0$ si $x < 1$. Supposons qu'une fonction f a une dérivée continue sur l'intervalle $[y, x]$, avec $0 < y < x$. Nous avons alors :

$$\sum_{y < n \leq x} a(n)f(n) = A(x)f(x) - A(y)f(y) - \int_{y}^{x} A(u)f'(u)du. \tag{3.4}$$

On aura besoin aussi du lemme suivant.

Lemme 8 (Par exemple [FS09]). $\binom{2n}{n} \sim \frac{4^n}{\sqrt{\pi}\sqrt{n}}$.

On déduit facilement de l'identité d'Abel (proposition 3.4) le lemme suivant.

Lemme 9. *Pour tout entier $n \geq 1$, nous avons :*

$$w(t_n) = (n-2)\widetilde{s}(t_n) - 3 - \int_{2}^{n-2} \widetilde{s}(t_u)du,$$

avec $\widetilde{s}(t_u) = s(t_u) - 1$.

Nous avons besoin d'estimer $w(t_n)$ par rapport à $s(t_n)$. Nous aurons besoin pour cela, de la forme faible du théorème des nombres premiers (WPNT en abrégé) due à Tchebychev.

Proposition 10 (Par exemple [Apo76]). *Pour tout entier $n \geq 2$ nous avons :*

$$\frac{1}{6}\frac{n}{\ln(n)} < \pi(n) < 6\frac{n}{\ln(n)}. \tag{3.5}$$

On déduit de WPNT la proposition indispensable suivante.

Proposition 11. *Pour tout $u \geq 4$, nous avons :*

$$\widetilde{s}(t_u) \geq \frac{1}{3}\frac{u-2}{\ln(u-2)}. \tag{3.6}$$

Démonstration. On peut montrer que pour tout nombre premier $p \geq 2$, nous avons $S_p = 2$. Par conséquent :

$$\widetilde{s}(t_u) = \sum_{i \leq u} S_i \geq 2\pi(u-1).$$

Par conséquent, et d'après WPNT nous obtenons :

$$\widetilde{s}(t_u) \geq \frac{1}{3}\frac{[u]-1}{\ln([u]-1)}.$$

Mais la fonction $x \mapsto \frac{x}{\ln(x)}$ est une fonction croissante sur l'intervalle $[3, +\infty[$. Il en découle que, pour tout $u \geq 3$:

$$\widetilde{s}(t_u) \geq \frac{1}{3}\frac{u-2}{\ln(u-2)},$$

ce qui nous mène à la preuve de la proposition. □

Nous sommes maintenant en mesure de formuler notre estimation de $w(t_n)$ par rapport à $s(t_n)$ dans la proposition suivante.

Proposition 12. $\limsup \left(\dfrac{w(t_n)}{(n-2)\widetilde{s}(t_n)}\right) \leq 1.$

Démonstration. Par application du lemme 9, pour tout $n \geq 3$, nous avons :

$$\frac{w(t_n)}{(n-2)\widetilde{s}(t_n)} = 1 - \frac{3}{(n-2)\widetilde{s}(t_n)} - \frac{1}{(n-2)\widetilde{s}(t_n)}\int_2^{n-2} \widetilde{s}(t_u)du.$$

Par conséquent, pour tout $n \geq 5$, nous avons :

$$\frac{w(t_n)}{(n-2)\widetilde{s}(t_n)} \leq 1 - \frac{3}{(n-2)\widetilde{s}(t_n)}$$

D'après la proposition 11 on en déduit que pour tout $n \geq 5$ nous avons :

$$\frac{3}{(n-2)\widetilde{s}(t_n)} \leq \frac{1}{\ln(n-2)}$$

Par conséquent, quand n tend vers ∞ nous obtenons :

$$\frac{3}{(n-2)\widetilde{s}(t_n)} \xrightarrow[n\to\infty]{} 0.$$

Ce qui implique que :

$$\limsup \left(\frac{w(t_n)}{(n-2)\widetilde{s}(t_n)}\right) \leq 1,$$

ce qui achève la démonstration de la proposition. □

Nous aurons besoin dans la suite d'encadrer la valeur de $s(t_x)$. Plus précisément, nous affirmons le résultat suivant.

Proposition 13. *Pour un nombre $x > 0$ suffisamment grand nous avons :*

$$x^{\frac{3}{4}}\frac{4^{\sqrt{x}-1}}{\sqrt{\pi}} \leq s(t_x) \leq x^{\frac{3}{4}}\ln(x)\frac{4^{\sqrt{x}-1}}{\sqrt{\pi}}.$$

Démonstration. Pour $x \geq 2$,

$$\begin{aligned} s(t_x) &= \sum_{n\leq x}\sum_{d|n}\binom{\frac{n}{d}+d-2}{d-1} \\ &= \sum_{dq\leq x}\binom{q+d-2}{d-1} \\ &= \sum_{d=1}^{\lfloor x\rfloor}\sum_{q=1}^{\lfloor \frac{x}{d}\rfloor}\binom{q+d-2}{d-1} \end{aligned}$$

De cela, nous déduisons que :

$$\begin{aligned} s(t_x) &\leq \sum_{d=1}^{\lfloor x\rfloor}\lfloor\frac{x}{d}\rfloor\binom{2(\lfloor x\rfloor-1)}{\lfloor x\rfloor-1} \\ &\leq x\ln(x)\binom{2(\lfloor x\rfloor-1)}{\lfloor x\rfloor-1}, \end{aligned} \tag{3.7}$$

et :

$$s(t_x) \geq \lfloor x\rfloor\binom{2(\lfloor x\rfloor-1)}{\lfloor x\rfloor-1}, \tag{3.8}$$

En utilisant la relation $\lfloor x\rfloor = x + O(1)$ combinée avec (3.7) et (3.8), nous obtenons

$$x\binom{2(\lfloor x\rfloor-1)}{\lfloor x\rfloor-1} \leq s(t_x) \leq x\ln(x)\binom{2(\lfloor x\rfloor-1)}{\lfloor x\rfloor-1}.$$

D'après le lemme 8 :

$$x^{\frac{3}{4}}\frac{4^{\sqrt{x}-1}}{\sqrt{\pi}} \leq s(t_x) \leq x^{\frac{3}{4}}\ln(x)\frac{4^{\sqrt{x}-1}}{\sqrt{\pi}},$$

ce qui prouve la proposition. □

En utilisant la proposition 13, nous pouvons étendre la proposition 12 comme suit.

Proposition 14. *Les suites $w(t_n)$ et $(n-2)s(t_n)$ sont équivalentes. Autrement dit :*

$$\frac{w(t_n)}{(n-2)s(t_n)} \xrightarrow[n\to\infty]{} 1.$$

Démonstration. D'après le lemme 9 :

$$\frac{w(t_n)}{(n-2)s(t_n)} = 1 - \frac{3}{(n-2)s(t_n)} - \frac{1}{(n-2)s(t_n)} \int_2^{n-2} s(t_x)dx.$$

Soit $\varepsilon > 0$. Alors, d'après la proposition 13, pour un nombre x suffisamment grand nous avons :

$$x^{\frac{3}{4}} \frac{4^{\sqrt{x}-1}}{\sqrt{\pi}} \leq s(t_x) \leq x^{\frac{3}{4}} \ln(x) \frac{4^{\sqrt{x}-1}}{\sqrt{\pi}}. \tag{3.9}$$

Mais :

$$\begin{aligned}
\int_2^{n-2} 4^{\sqrt{x}}dx &\overset{u=\sqrt{x}}{=} \int_{\sqrt{2}}^{\sqrt{n-2}} 2\, u\, 4^u du \\
&= \Big[\frac{2u}{\ln(4)}4^u\Big]_{\sqrt{2}}^{\sqrt{n-2}} - \Big[\frac{2}{(\ln 4)^2(}4^u\Big]_{\sqrt{2}}^{\sqrt{n-2}} \\
&= \frac{2\sqrt{n-2}\, .4^{\sqrt{n-2}}}{\ln(4)} - \frac{2\sqrt{2}\, .4^{\sqrt{2}}}{\ln(4)} - \frac{2\, .4^{\sqrt{n-2}}}{(\ln 4)^2} + \frac{2\, .4^{\sqrt{2}}}{(\ln 4)^2}.
\end{aligned}$$

Comme $\frac{1}{(n-2)s(t_n)}$ s'annule à l'infini et $\widetilde{s}(t_n)$ est équivalent à $s(t_n)$, on peut supposer que (3.9) est vraie à partir de 2 et la proposition 13 est valide pour $\widetilde{s}(t_n)$. Par conséquent :

$$\begin{aligned}
\frac{1}{(n-2)\widetilde{s}(t_n)} \int_2^{n-2} \widetilde{s}(t_x)dx &\leq \frac{1}{(n-2)(n-2)^{\frac{3}{4}}\, 4^{\sqrt{n}}} \int_2^{n-2} x^{\frac{3}{4}} \ln(x)\, 4^{\sqrt{x}}dx \\
&\leq \frac{(n-2)^{\frac{3}{4}} \ln(n-2)}{(n-2)(n-2)^{\frac{3}{4}}\, 4^{\sqrt{n}}} \int_2^{n-2} 4^{\sqrt{x}}dx \\
&\leq \frac{\ln(n-2)}{(n-2)4^{\sqrt{n}}} \int_2^{n-2} 4^{\sqrt{x}}dx.
\end{aligned} \tag{3.10}$$

D'après (3.10) combiné avec (3.10), il en résulte que :

$$\frac{1}{(n-2)\widetilde{s}(t_n)} \int_2^{n-2} \widetilde{s}(t_x)dx \leq$$
$$\frac{2\sqrt{n-2}\, .4^{\sqrt{n-2}}}{\ln(4)} \times \frac{\ln(n-2)}{(n-2)4^{\sqrt{n}}} + \frac{2\, .4^{\sqrt{2}}}{(\ln 4)^2} \times \frac{\ln(n-2)}{(n-2)4^{\sqrt{n}}} \xrightarrow[n\to\infty]{} 0.$$

Nous concluons que :

$$\frac{w(t_n)}{(n-2)\widetilde{s}(t_n)} \xrightarrow[n\to\infty]{} 1.$$

ce qui prouve la proposition. □

Maintenant, nous sommes en mesure de donner la preuve du théorème 18.

Preuve du théorème 18. D'après la proposition 14, il suffit de montrer que :

$$s(t_n)(\ln s(t_n))^2 \sim (n-2)s(t_n).$$

Pour cela, observons que nous avons :

$$\frac{s(t_n)(\ln s(t_n))^2}{(n-2)s(t_n)} = \frac{(\ln s(t_n))^2}{n-2}$$

Cependant :

$$\ln(s(t_n)) \sim \ln(4)\sqrt{n}.$$

D'où :

$$(\ln s(t_n))^2 \sim (\ln 4)^2\, n.$$

Par conséquent :

$$\frac{(\ln s(t_n))^2}{(n-2)} \xrightarrow[n\to\infty]{} (\ln 4)^2.$$

On en déduit que :

$$\frac{w(t_n)}{s(t_n)(\ln s(t_n))^2} \xrightarrow[n\to\infty]{} \frac{1}{(\ln 4)^2} < 1.$$

Ce qui achève la démonstration du théorème. □

3.2.6 Résultats expérimentaux

On montre expérimentalement que pour les petites valeurs de n les automates minimaux en nombre de transitions coïncident avec ceux obtenus par notre construction. Avec notre méthode, l'automate CFS associé au système de partitions $Z(\mathcal{A}_n)$ peut contenir plusieurs états initiaux. Toutefois, afin de comparer le nombre d'automates minimaux et leur nombre de transitions avec ceux obtenus par Cox [Cox07] (voir le tableau 3.1), nous devons limiter notre étude aux systèmes de partitions CFS conduisant à des automates ayant un état initial unique.

Il est facile de voir qu'un système CFSZ conduit à un automate ayant un unique état initial si, et seulement si, l'arbre correspondant possède le 1-arbre comme sous-arbre gauche. En effet, le calcul d'un système $Z(\mathcal{A}_n)$ minimal ayant un unique état initial se réduit à trouver un $(n+2)$-arbre de poids minimal possédant le 1-arbre comme sous-arbre gauche.

Le n-arbre de poids minimal ayant le 1-arbre comme sous-arbre gauche définira le $\overline{Z}$-arbre de rang n.

Soit $\overline{\mathrm{Split}}(t)$ la fonction qui renvoie l'arbre obtenu à partir de t en remplaçant une feuille ayant un coût minimal dans le sous-arbre droit du t par l'unique 2-arbre.

La suite des $\overline{Z}$-arbres peut être calculée de manière inductive comme suit :
- le $\overline{Z}$-arbre de rang deux est le 2-arbre,
- si t est un $\overline{Z}$-arbre (de rang $i > 1$) alors $\overline{\mathrm{Split}}(t)$ est un $\overline{Z}$-arbre (de rang $i+1$).

D'une manière similaire à celle décrite dans la section 3.2.4.1, nous pouvons définir l'ensemble des $\overline{P}$-arbres et vérifier que, pour un $\overline{P}$-arbre t_n, on a $s(t_n) = 1 + \sum_{i=1}^{n-1} \frac{S_{i+2}}{2}$ et $w(t_n) = \sum_{i=1}^{n} \frac{iS_{i+1}}{2}$.

Remarque 3. Nos expérimentations montrent que, pour les petites valeurs de n, tous les automates finis minimaux en nombre de transitions, non-déterministes, sans ε-transition, ayant $n+1$ états dont un seul initial, reconnaissant le langage $L(E_n)$ sont exactement ceux produit par notre construction (voir tableau 3.1).

Remarque 4. En utilisant les fonctions génératrices, nous avons calculé le nombre de transitions de l'automate associé au système de partitions minimal $Z(\mathcal{A}_n)$ pour un nombre d'états dépassant le milliard (voir figure 3.16). L'étude asymptotique du nombre de transitions produit par notre construction (détaillée dans la section suivante) montre que ce nombre est asymptotiquement minimal.

Ces deux dernières remarques nous mènent à conjecturer que tous les automates finis minimaux en nombre de transitions, non-déterministes, sans ε-transition, ayant $n+1$ états dont un seul initial reconnaissant le langage $L(E_n)$ peuvent être obtenus complètement depuis les arbres binaires complets d'un poids minimal.

Conjecture 1. La suite d'automates $(\mathcal{A}_{n_i})_{i\geq 1}$ pour laquelle l'automate équivalent minimisé non-déterministe, sans ε-transition, ayant $n_i + 1$ états dont un seul initial, est unique, est donnée par :

$$n_i = 1 + \sum_{j=1}^{i-1} \frac{S_{j+2}}{2}.$$

Dans ce cas, le nombre de transitions dans l'automate équivalent minimisé unique est égal à $\sum_{j=1}^{i} \frac{jS_{j+1}}{2}$.

n	1	2	3	4	5	6	7	8	9	10	11	12	13	14
i	1	3	6	9	13	18	23	≤ 28	≤ 34	≤ 41	?	?	?	?
ii	1	3	6	9	13	18	23	28	33	39	46	53	60	67
iii	1	1	2	1	1	4	6	≥ 1	≥ 1	≥ 1	?	?	?	?
iv	1	1	2	1	1	4	6	4	1	1	5	10	10	5

TABLE 3.1 – Tableau de comparaison. (i) Le nombre minimal de transitions estimé par Cox [Cox07] (ii) Le nombre de transitions dans nos automates réduits (iii) Le nombre d'automates minimaux estimé par Cox [Cox07] (iv) Le nombre d'automates réduit estimé par notre approche.

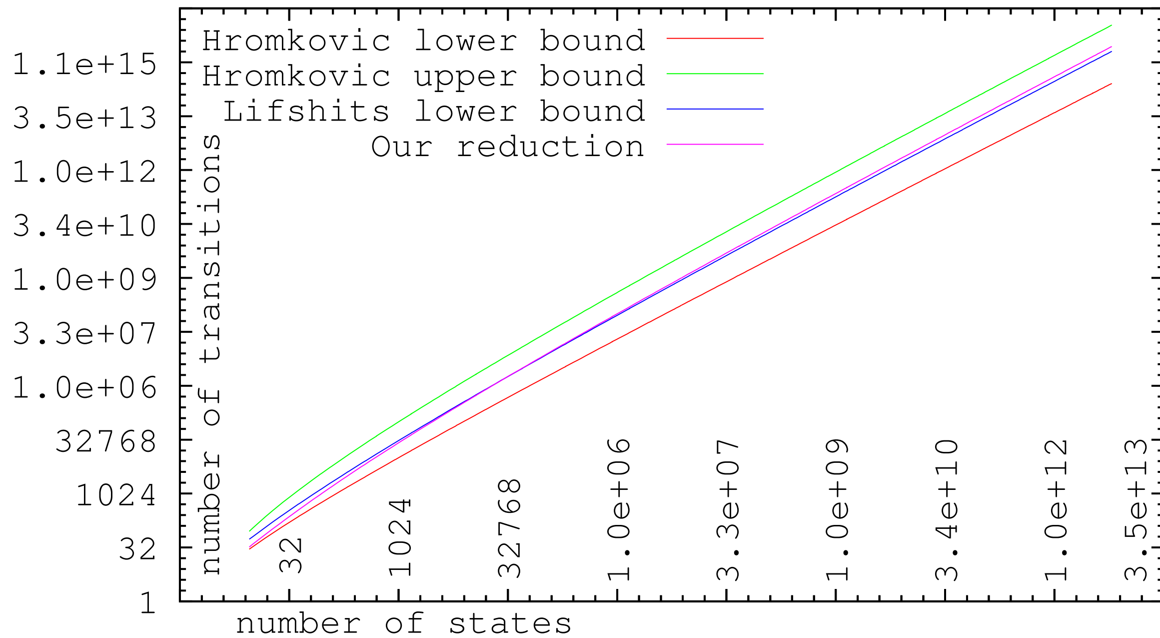


FIGURE 3.16 – Résultats expérimentaux.

Nous avons montré que les P-arbres peuvent être utilisés pour réduire efficacement le nombre de transitions d'un automate triangulaire. Dans la section suivante on discute un résultat indépendant qui concerne une sous-classe de l'ensemble des P-arbres.

3.3 Les arbres premiers

Nous montrons dans cette section une application particulière de la classe des P-arbres qui permet d'engendrer la suite des nombres premiers en se basant sur ce qu'on a appelé les *arbres premiers*. Un P-arbre est dit premier si et seulement s'il ne possède que deux feuilles de coût minimal.

Définition 40. Un arbre premier t est un P-arbre ayant $|M_t| = 2$.

Le théorème suivant établit une bijection entre les arbres premiers et les nombres premiers.

Théorème 19. *Pour tout $n \geq 1$, n est premier si, et seulement si, t_n, le P-arbre d'ordre n, est un arbre premier.*

Démonstration. Comme déjà défini dans la section 3.2.4.3, S_n est la somme de la $n^{\text{ième}}$ ligne du triangle d'Ératosthène - Pascal. Par définition de ce triangle, un entier naturel n est premier

si, et seulement si, $S_n = 2$:

$$\overbrace{1, \underbrace{0, 0, ..., 0}_{\text{zéros}}, 1}^{n \text{ éléments}}$$

Or, d'après le théorème 17, établissant une bijection entre les P-arbres et les lignes du triangle d'Ératosthène - Pascal, et d'après le corollaire 14, on conclut qu'un P-arbre est premier seulement s'il possède exactement deux feuilles de coût minimal. □

L'algorithme 6 exploite le théorème 19 pour produire la suite des nombres premiers. Si nous disposons de l'arbre premier t_p associé au nombre premier p, le calcul du prochain nombre premier peut être réalisé en utilisant l'algorithme suivant.

Algorithme 6 : $\text{Successeur}(t_p)$

Require: p un nombre premier
Ensure: $t_{\text{Successeur}(p)}$
1: **repeat**
2: $\quad t_p \leftarrow \text{SplitAll}(t_p)$
3: **until** $|M_{t_p}| = 2$

D'un point de vue de performance, l'algorithme tel qu'il est donné n'est pas rapide (temps d'exécution en $O(n)$) et il nécessite beaucoup d'espace mémoire (de l'ordre de $O(n)$), alors que des algorithmes en $O(\log(n))$ sont bien connus. Cependant l'essentiel de cet algorithme est de montrer une correspondance assez remarquable entre les nombres premiers et les arbres binaires tout en engendrant la suite des premiers seulement en éclatant des feuilles de coût minimal et en vérifiant à chaque étape si l'arbre produit est premier ou pas. Notons que le coût d'une feuille se calcule simplement par la formule $ab+a+b$, où a et b sont respectivement le nombre d'arêtes gauches et droites dans le chemin reliant la feuille en question à la racine de l'arbre.

3.4 Conclusion

Nous avons montré dans ce chapitre comment les arbres binaires peuvent être utilisés pour concevoir un algorithme rapide pour calculer un automate avec un nombre réduit de transitions [3] reconnaissant le langage $L(E_n)$. Nous avons bien vérifié que notre algorithme donne le nombre minimal de transitions pour $n = 1$ à 7 (voir le tableau 3.1) et nous avons montré que notre réduction est asymptotiquement une minimisation. C'est pourquoi nous conjecturons que l'algorithme 5 produit l'automate minimal en nombre de transitions, dans la mesure où le nombre de transitions est équivalent à $n(\log_2(n))^2$, qui correspond, à la fois à la borne supérieure et à la borne inférieure. Montrer que l'algorithme de réduction, défini dans ce chapitre, est une minimisation pour les automates triangulaires reste ouvert (conjecture 1).

3. Asymptotiquement minimal.

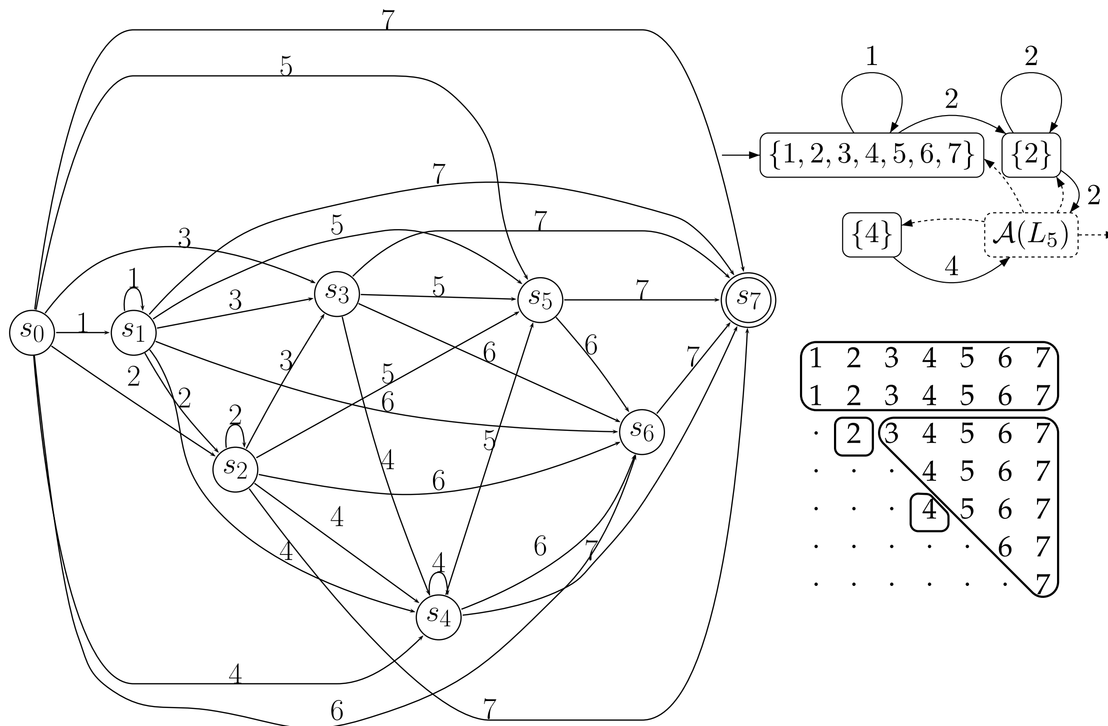


FIGURE 3.17 – L'automate réduit (à droite) est obtenu à partir d'une décomposition de la table de transition (en bas à droite) de l'automate homogène (à gauche). L'automate $\mathcal{A}(L_5)$ est la partie de l'automate réduit qui représente le triangle (dans la décomposition de la table de transition).

La minimisation du nombre de transitions des automates reconnaissant le langage $L(E_n)$ pourrait être utilisée pour une heuristique de réduction du nombre de transitions d'un automate fini non-déterministe homogène quelconque. L'idée serait de décomposer la table de transition d'un automate donné en blocs de forme rectangulaire et triangulaire. Après cela, nous utilisons l'algorithme 5 pour calculer l'automate minimal en nombre de transitions de chaque fragment triangulaire. Puis, en appliquant le concept de *Common Follow Sets* sur la décomposition en bloc en résultant, nous pouvons construire un automate réduit avec moins de transitions que l'automate de départ. Cette idée est illustrée sur un exemple sur la figure 3.17. Une suite possible de cette étude consiste à généraliser le concepts du CFS aux automates non homogènes, ou encore à trouver d'autres applications pour les P-arbres et les Z-arbres (voir section 3.3).

Deuxième partie

Partie pratique

CHAPITRE 4

AUTOMATES À TROUS POUR LES DICTIONNAIRES DE MOTS

DANS ce chapitre on propose une structure de données particulière, appelée *automate à trous binaire* permettant d'engendrer la liste des mots voisins en temps et en espace expérimentalement linéaires. Théoriquement, et dans des cas extrêmes (pire des cas), cette structure peut être lourde en recherche ou grande en volume (voir théorèmes 20 et 21).

Le contexte de ce chapitre est moins théorique. Il est plutôt orienté vers une étude beaucoup plus pratique avec des tests et des résultats expérimentaux effectués sur des dictionnaires engendrés d'une façon pseudo-aléatoire selon plusieurs critères (nombre de mots, taille des mots, nombre de symboles de l'alphabet) et des dictionnaires de mots naturels de plusieurs langues.

4.1 Notations et définitions

Définition 41. Soit Σ un alphabet. On appelle *dictionnaire* un ensemble fini de mot finis sur Σ.

Définition 42. Soient Σ un alphabet, $d : \Sigma \times \Sigma \rightarrow \mathbb{N}$ une distance sur les couples d'éléments de Σ , k un entier et w un mot sur Σ. On appelle *voisin* de w d'*ordre* k tout mot sur Σ qui est distant au plus k de w : $V_k(w) = \{v \in \Sigma^* \mid d(v, w) \leq k\}$.

La distance de Hamming permet de quantifier la différence entre deux mots. Elle est utilisée en informatique, en bio-informatique, en traitement du signal et en télécommunication. Pour deux mots de même longueur, la distance de Hamming retourne le nombre de positions qui correspondent à deux lettres différents sur les deux mots.

Définition 43. Soit Σ un alphabet. Soient $a = a_1a_2\cdots a_n$ et $b = b_1b_2\cdots b_n$ deux mots sur Σ de même longueur. On appelle *distance de Hamming* la distance entre a et b définie comme suit : $d(a,b) = |\{i : a_i \neq b_i\}|$.

Étant donnés un mot u et un dictionnaire D sur Σ, on veut rechercher le mot u dans le dictionnaire D d'une manière approchée. En d'autres termes, on veut retrouver les mots voisins de u dans D selon une distance donnée.

Définition 44. Soit $u = a_1a_2\ldots a_n$ un mot de Σ^* et soit $p \in \mathbb{N}$ un entier naturel. On définit pour $1 \leq p \leq n$ l'ensemble des mots

$$Sub_p(u) = \{a_1a_2\ldots a_{p-1}aa_{p+1}\ldots a_n \mid a \in \Sigma\}$$

obtenus par le remplacement de a_p, la $p^{\text{ième}}$ lettre du mot u, par chaque lettre de l'alphabet Σ. Dans ce cas on définit l'ensemble $Sub(u)$ par

$$Sub(u) = \bigcup_{1\leq p\leq n} Sub_p(u)$$

Exemple 25. Pour un alphabet $\Sigma = \{a, b, c, i, n, o, p, t, u, z\}$ et pour le mot *but* on a :
- $Sub_1(but) = \{aut, but, cut, iut, nut, out, put, tut, uut, zut\}$
- $Sub_2(but) = \{bat, bbt, bct, bit, bnt, bot, bpt, btt, but, bzt\}$
- $Sub_3(but) = \{bua, bub, buc, bui, bun, buo, bup, but, buu, buz\}$
- $Sub(but) = \{aut, bat, bbt, bct, bit, bnt, bot, bpt, btt, bua, bub, buc, bui, bun, buo, bup, but, buu, buz, bzt, cut, iut, nut, out, put, tut, uut, zut\}$

Définition 45. Soit $u \in \Sigma^*$ et $k \in \mathbb{N}$. On définit récursivement *le voisinage de Hamming d'ordre* k du mot u comme suit :
- $H_0(u) = \{u\}$
- $H_1(u) = Sub(u)$
- $H_k(u) = \bigcup_{v\in H_{k-1}(u)} Sub(v)$ pour $k > 1$

Le voisinage de Hamming d'ordre k d'un dictionnaire D est :

$$H_k(D) = \bigcup_{u\in D} H_k(u)$$

Exemple 26. Soit $D = \{in, on, put, but\}$. On a $H_0(D) = D$ et :
$H_1(D) = \{ia, an, aut, bat, bbt, bct, bit, bn, bnt, bot, bpt, btt, bua, bub, buc, bui, bun, buo, bup, but, buu, buz, bzt, cn, cut, ib, ic, ii, in, io, ip, it, iu, iut, iz, nn, nut, oa, ob, oc, oi, on, oo, op, ot, ou, out, oz, pat, pbt, pct, pit, pn, pnt, pot, ppt, ptt, pua, pub, puc, pui, pun, puo, pup, put, puu, puz, pzt, tn, tut, un, uut, zn, zut\}$.

4.2 Solution naïve du problème de la recherche approchée

Pour un entier $k \in \mathbb{N}$, un alphabet fini Σ, un mot $w \in \Sigma^*$, un langage fini $D \subset \Sigma^*$ appelé dictionnaire, et une distance[1] d, la recherche approchée d'ordre k de w dans D consiste à

1. En général on s'intéresse à la distance de Hamming ou à la distance d'édition (distance de Levenstein).

calculer tous les mots v de D tels que la distance entre chaque v et w est au plus k : $\{v \in D \mid d(v,w) \leq k\}$.

Il y a deux méthodes naïves :

1. La première consiste à construire la liste des voisins d'ordre k de w par rapport à Σ, ensuite à rechercher les éléments de cette liste dans le dictionnaire D. Cette méthode est linéaire en espace mais exponentielle en temps de recherche, car le nombre de voisins d'ordre k de w est de l'ordre de $O((|\Sigma| \times |w|)^k)$ pour la distance de Hamming par exemple.
2. La deuxième méthode consiste à pré-construire un deuxième dictionnaire D' contenant les mots du dictionnaire D et les voisins d'ordre k de chaque mot de D. Le dictionnaire D' doit et peut être organisé d'une façon permettant de retrouver facilement les voisins d'un mot w par rapport au dictionnaire D. Cette méthode est linéaire en temps de recherche mais la taille du dictionnaire D' est exponentielle : $O\left(\sum_{v \in D} (|\Sigma| \times |v|)^k\right)$.

Maintenant, si on fixe un entier $k \in \mathbb{N}$, et la mesure de Hamming comme distance, notre recherche approchée consiste à retourner les mots de D qui font partie du voisinage de Hamming de u d'ordre k : $H_k(u) \cap D$.

4.3 Automate à trous

Soit Σ un alphabet fini. On note $\bullet$ la lettre *trou*, et $\#$ la lettre *séparateur*, ces deux lettres n'appartenant pas à l'alphabet : $\bullet \notin \Sigma$ et $\# \notin \Sigma$.

Définition 46. On appelle *mot à trous* tout mot de $(\Sigma \cup \{\bullet\})^* \# \Sigma^*$.

Définition 47. Soit $u = a_1 a_2 \dots a_n$ un mot à trous de $(\Sigma \cup \{\bullet\})^*$ et soit $p \in \mathbb{N}$ un entier naturel. On définit pour $1 \leq p \leq n$ le mot à trous :

$$\dot{S}ub_p(u) = a_1 a_2 \dots a_{p-1} \bullet a_{p+1} \dots a_n$$

obtenu par le remplacement de a_p, la $p^{\text{ième}}$ lettre du mot u, par la lettre $\bullet$. Dans ce cas on définit l'ensemble $\dot{S}ub(u)$ par :

$$\dot{S}ub(u) = \{\dot{S}ub_p(u) \mid 1 \leq p \leq n\}$$

Exemple 27. Pour le mot *but* on a :
- $\dot{S}ub_2(but) = b \bullet t$
- $\dot{S}ub_3(but) = bu\bullet$
- $\dot{S}ub(but) = \{\bullet ut, b \bullet t, bu\bullet\}$

Définition 48. Soit $u \in (\Sigma \cup \{\bullet\})^*$ et $k \in \mathbb{N}$. On définit récursivement *le voisinage de Hamming à trous d'ordre k* du mot u comme suit :
- $\dot{H}_0(u) = \{u\}$
- $\dot{H}_1(u) = \dot{S}ub(u)$
- $\dot{H}_k(u) = \bigcup_{v \in \dot{H}_{k-1}(u)} \dot{S}ub(v)$ pour $k > 1$

Ainsi, l'ensemble $\dot{H}_k(u)$ est une version compacte de l'ensemble des voisins du mot u. Au lieu de remplacer k lettres du mot u par toutes les lettres de l'alphabet, il suffit de les remplacer par la lettre spécifique trou $\bullet$.

Dans un but purement technique nous avons rajouté une queue pour chaque mot à trous dans $\dot{H}_k(u)$. Cette queue est en fait le mot lui-même u précédé du symbole $\#$. Ceci nous permet de retrouver le mot qui était à l'origine de chaque mot à trous. En prenant en considération cette queue, le voisinage de Hamming à trous d'ordre k d'un dictionnaire D est défini par :

$$\dot{H}_k(D) = \bigcup_{u \in D} \dot{H}_k(u).\{\#u\}$$

Exemple 28. Soit $D = \{in, on, put, but\}$. On a :
- $\dot{H}_0(D) = \{in\#in, on\#on, put\#put, but\#but\}$
- $\dot{H}_1(D) = \{in\#in, on\#on, put\#put, but\#but, \bullet n\#in, \bullet n\#on, \bullet ut\#put, \bullet\, ut\#but, i\bullet\#in, o\bullet\#on, p\bullet t\#put, b\bullet t\#but, pu\bullet\#put, bu\bullet\#but\}$
- $\dot{H}_2(D) = \{in\#in, on\#on, put\#put, but\#but, \bullet n\#in, \bullet n\#on, \bullet ut\#put, \bullet\, ut\#but, i\bullet\#in, o\bullet\#on, p\bullet t\#put, b\bullet t\#but, pu\bullet\#put, bu\bullet\#but, \bullet\bullet\#in, \bullet\bullet\#on, \bullet\bullet t\#put, \bullet\bullet t\#but, p\bullet\bullet\#put, b\bullet\bullet\#but, \bullet u\bullet\#put, \bullet u\bullet\#but\}$

Définition 49. Soient D un dictionnaire et k un entier naturel. On appelle *automate à trous* d'ordre k pour le dictionnaire D l'automate minimal déterministe reconnaissant $\dot{H}_k(D)$.

Exemple 29. L'automate illustré dans la figure 4.1 est l'automate à trous d'ordre 1 pour le dictionnaire $D = \{in, on, put, but\}$.

On remarque que l'automate à trous est acyclique (voir page 15).

Théorème 20. *Soit $w \in \Sigma^*$. L'automate à trous d'ordre k pour un dictionnaire D permet de retrouver tous les voisins v de w dans D en un temps de $O(|w|^k)$.*

Démonstration. La recherche des voisins d'ordre k du mot w en utilisant l'automate à trous se résume à un ensemble de lectures particulières non déterministes du mot w dans l'automate à trous. Une lecture consiste à lire w_i la prochaine lettre du mot w ensuite :

1. avancer dans l'automate en lisant la transition étiqueté par w_i et continuer une recherche d'ordre k du reste du mot, ou
2. avancer dans l'automate en lisant la transition étiqueté par le trou $\bullet$ et continuer une recherche d'ordre $(k-1)$ du reste du mot.

Pendant une lecture, quand on lit k trous, la lecture devient déterministe, ce qui signifie qu'on ne va plus suivre les transitions étiquetées par le trou. Ainsi, une lecture s'arrête quand on ne peut plus avancer dans l'automate ou quand on consomme la totalité du mot. Dans ce cas, si la prochaine transition sur l'automate est le séparateur $\#$ alors tous les mots qui peuvent être lus après cette transition sont des voisins du mot w. Finalement, la recherche correspond à la lecture des mots à trous voisins de w en nombre maximum de $2|w|^k$. □

Théorème 21. *Soit L la longueur du plus long mot du dictionnaire D. Soit $w \in \Sigma^*$. La taille de l'automate à trous d'ordre k pour un dictionnaire D est de l'ordre de $O\left(\sum_{v \in D} (|v|)^k\right) \approx O(|D| \times L^k)$.*

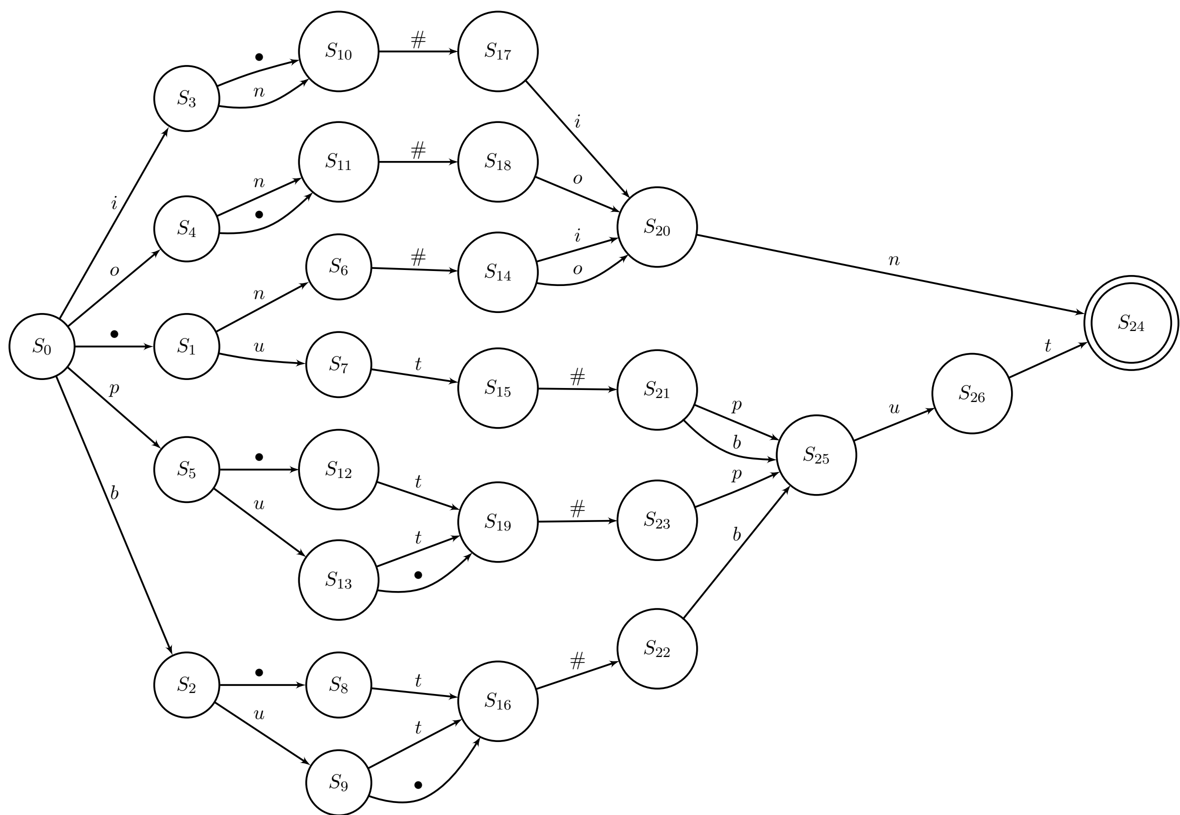


FIGURE 4.1 – Automate à trous.

Démonstration. L'automate à trous possède le pouvoir de reconnaître le voisinage de Hamming à trous du dictionnaire D. Ce voisinage contient les voisins à trous de chaque mot du dictionnaire D. Ainsi le nombre de mots dans ce voisinage est au maximum égal à $\sum_{v \in D} (2|v|)^k$. □

Les complexités en temps et en espace énoncées dans les deux derniers théorèmes laissent penser que l'automate à trous est exponentiel en taille et en temps de recherche. Avec SAÏD ABDEDDAÏM, de l'université de Rouen, France, nous avions la conviction que l'automate à trous est *pratiquement* beaucoup plus petit et permet une recherche beaucoup plus rapide.

La section suivante montre les résultats expérimentaux obtenus en utilisant l'automate à trous.

4.4 Résultats expérimentaux

Comme expliqué en introduction, les tests ont été effectués pour des dictionnaires engendrés d'une façon pseudo-aléatoire selon plusieurs critères, comme le nombre de mots, la taille des mots ou encore le nombre de symboles de l'alphabet, et des dictionnaires de mots naturels de plusieurs langues.

	Critère	**min**	**max**
$\|D\|$	**Nombre de mots**	10	1000
L	**Taille des mots**	1	15
$\|\Sigma\|$	**Nombre de symboles de l'alphabet**	3	30

TABLE 4.1 – Plus de 1000 dictionnaires pseudo-aléatoires ont été engendrés selon ces critères.

Langue	**Taille moyenne des mots**	**Nombre de mots**
Afrikaans	9,50	130000
Allemand	11,00	160000
Anglais	9,25	219000
Espagnol	7,75	86000
Français	9,50	336000
Italien	7,50	60000
Polonais	8,00	109000
Turc	7,00	25000

TABLE 4.2 – Dictionnaires de mots naturels de plusieurs langues.

Observation 1. *Le nombre de mots à trous du voisinage $\dot{H}_k(D)$ est pratiquement (expérimentalement), comme théoriquement, de l'ordre de $O(L^k)$.*

Le graphe de la figure 4.2 montre le rapport $\frac{|\dot{H}_k(D)|}{|D|}$ pour plusieurs petites valeurs de k. On remarque bien évidemment que le rapport entre le nombre des voisins à trous d'un dictionnaire D et le nombre de mots du dictionnaire est exponentiel en taille moyenne des mots, ainsi $\frac{|\dot{H}_k(D)|}{|D|}$ est de l'ordre de L^k. Par conséquent, $|\dot{H}_k(D)|$ est bel et bien de l'ordre de $O(L^k)$.

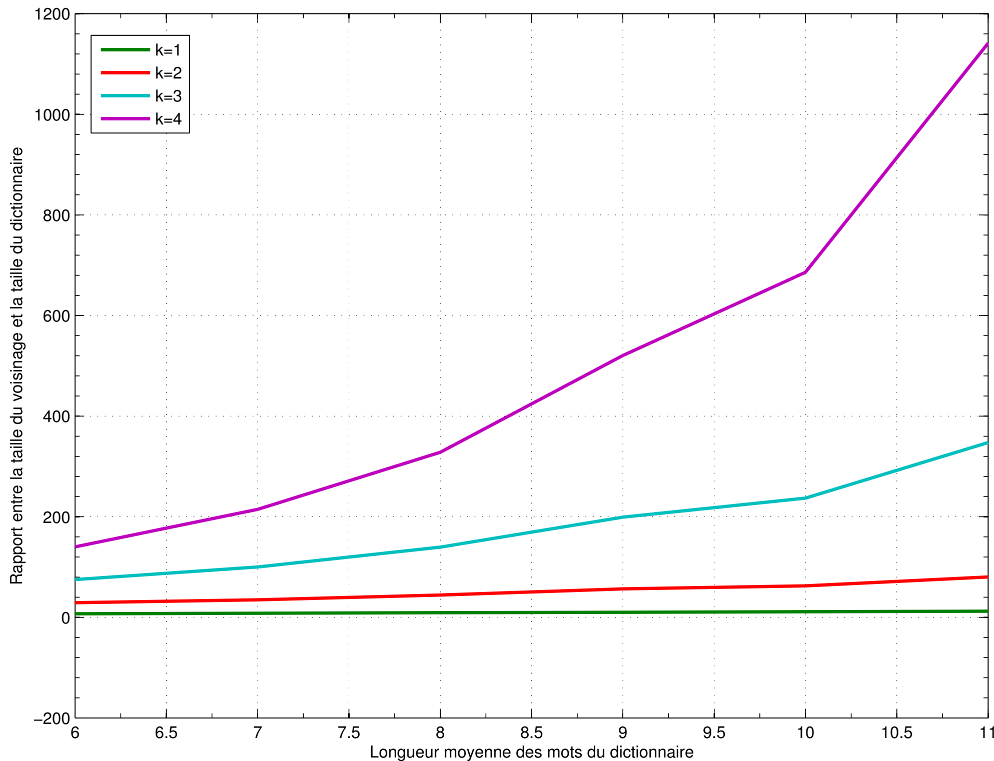


FIGURE 4.2 – Taille du voisinage par rapport à la taille du dictionnaire.

Observation 2. *Pour les petites valeurs de k, la taille (en nombre d'états ou en nombre de transitions) de l'automate à trous d'ordre k du dictionnaire D est pratiquement (expérimentalement) proche de la taille de l'automate minimal reconnaissant D.*

Les deux courbes de la figure 4.3 illustrent la taille moyenne de l'automate à trous en nombre d'états et en nombre de transitions pour plusieurs valeurs de k pour la plupart des dictionnaires testés. On note que certains dictionnaires n'épousent pas forcement ces courbes, mais globalement, et en moyenne, la majorité des dictionnaires suivent cette allure. Mis à part pour $k = 1$, la taille de l'automate à trous n'est pas exponentielle mais plutôt du même ordre que l'automate minimal reconnaissant le dictionnaire lui-même. Dans le but de compacter davantage l'automate à trous, on a introduit *l'automate à trous binaire*. Ce dernier reconnaît, exactement, une représentation binaire particulière des mots reconnus par l'automate à trous défini au départ. Cette représentation binaire consiste à encoder les lettres

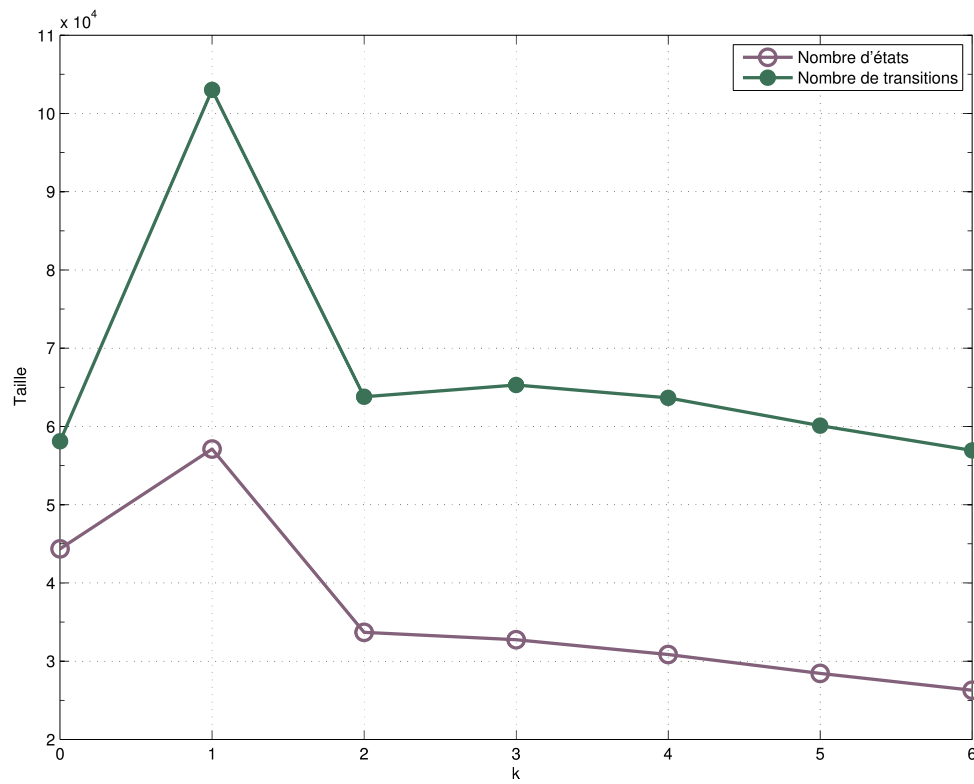


FIGURE 4.3 – Nombre d'états et nombre de transitions de l'automate à trous.

du voisinage $\dot{H}_k(u)$ par un codage de Huffman ; ce dernier s'appuie sur la fréquence d'apparition de chaque lettre. Ainsi on a proposé deux versions pour le calcul de la fréquence de chaque lettre. Une première version calcule la fréquence des lettres directement à partir de l'ensemble du voisinage $\dot{H}_k(u)$. La deuxième version calcule ces fréquences à partir des étiquettes des transitions de l'automate à trous. Le bleu et le rouge servent à différencier entre les deux versions du calcul des fréquences de Huffman. On remarque qu'il faut bien coder, préalablement, le mot en entrée w par le même code de Huffman utilisé pour calculer l'automate binaire.

Observation 3. *Pour les petites valeurs de k, l'automate à trous binaire compacte l'automate à trous avec un facteur de 3. Ce facteur croît quand k croît, ou encore quand le nombre de mots de D croît.*

L'histogramme de la figure 4.4 montre le rapport entre la taille (en bits) de l'automate à trous binaire et de l'automate à trous pour plusieurs valeurs de k. On remarque que pour les petites valeurs de k, l'automate à trous binaire est cinq fois plus petit que l'automate à trous. Il est clair que plus k est grand, plus l'automate à trous binaire est compact. Le graphe

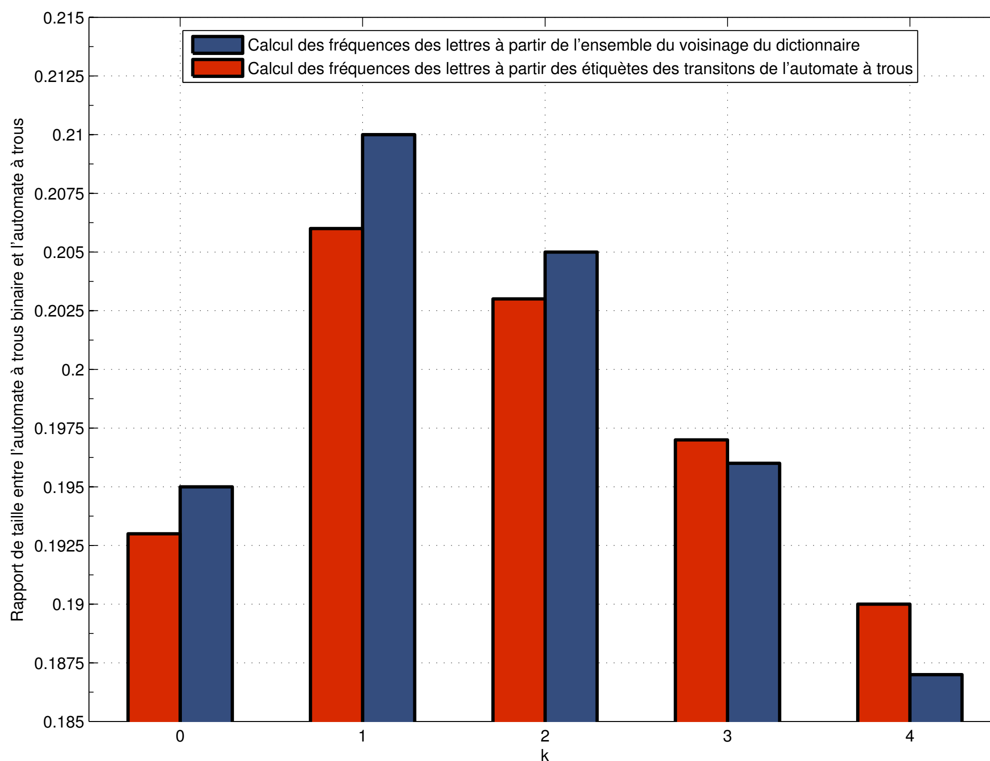


FIGURE 4.4 – Rapport de taille entre l'automate à trous binaire et l'automate à trous selon k.

de la figure 4.5 montre le rapport entre la taille (en bits) de l'automate à trous binaire et de l'automate à trous pour plusieurs valeurs de $|D|$. On remarque clairement que plus $|D|$ est grand, plus l'automate à trous binaire est compacte. L'histogramme de la figure 4.6 montre

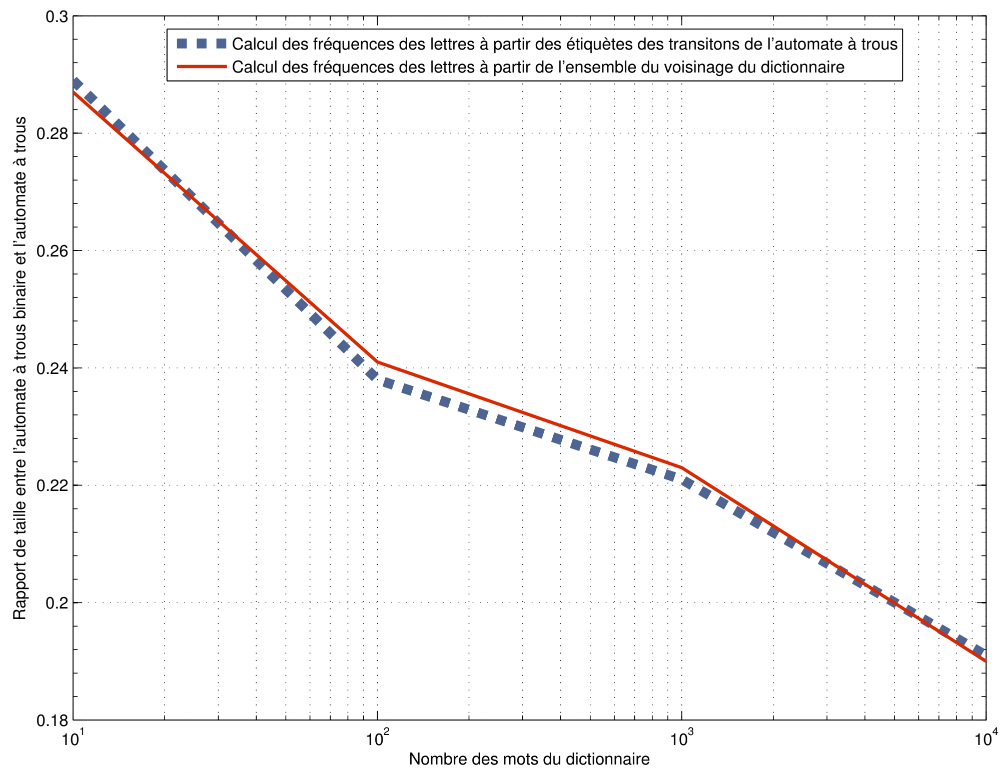


FIGURE 4.5 – Rapport de taille entre l'automate à trous binaire et l'automate à trous selon $|D|$.

le rapport entre la taille (en bits) de l'automate à trous binaire et de l'automate à trous pour plusieurs dictionnaires de langues.

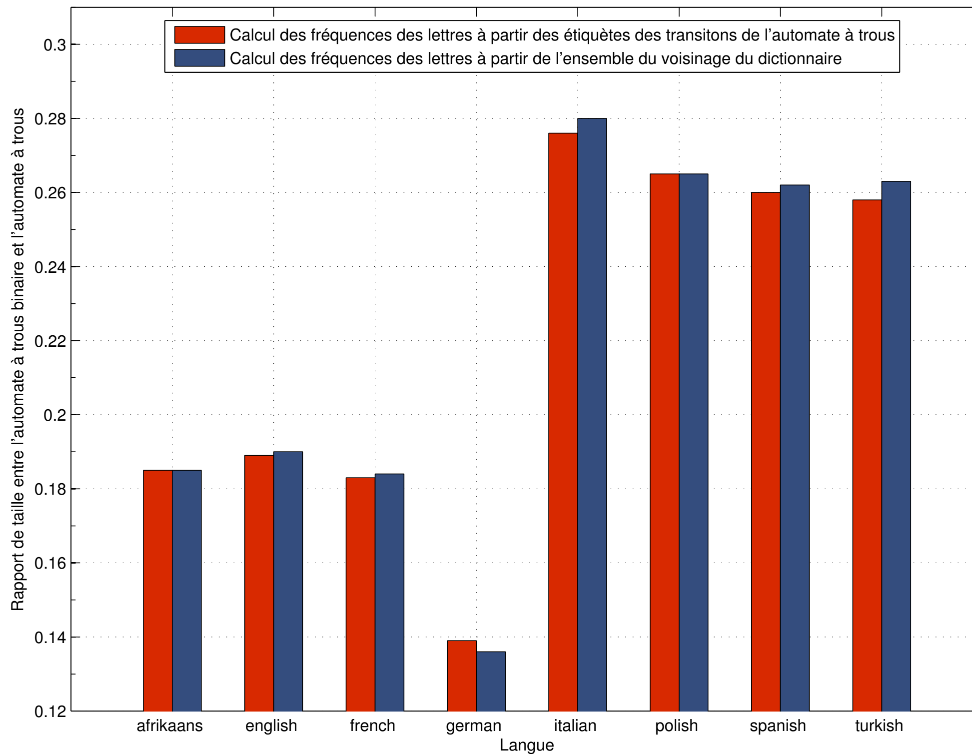


FIGURE 4.6 – Rapport de taille entre l'automate à trous binaire et l'automate à trous pour plusieurs dictionnaires de langues.

Observation 4. *Les tailles respectives des deux versions de l'automate à trous binaire sont légèrement différentes.*

Observation 5. *Pour les petites valeurs de k, pratiquement, la taille en bits de l'automate à trous binaire du dictionnaire D représente moins de 10% de la taille du voisinage $\dot{H}_k(D)$.*

Observation 6. *Pour les petites valeurs de k :*

1. *Pratiquement, la taille de l'automate à trous binaire est linéaire de l'ordre de $|D| \times k \times L$ et non pas $|D| \times L^k$.*
2. *Le temps de recherche expérimental nécessaire, pour retrouver les voisins d'ordre k d'un mot w dans un dictionnaire D en utilisant l'automate à trous, est proportionnel à k, le nombre d'erreurs, et à $|w|$ la longueur du mot, pratiquement, ce temps de recherche est de l'ordre de $|w| \times k$.*

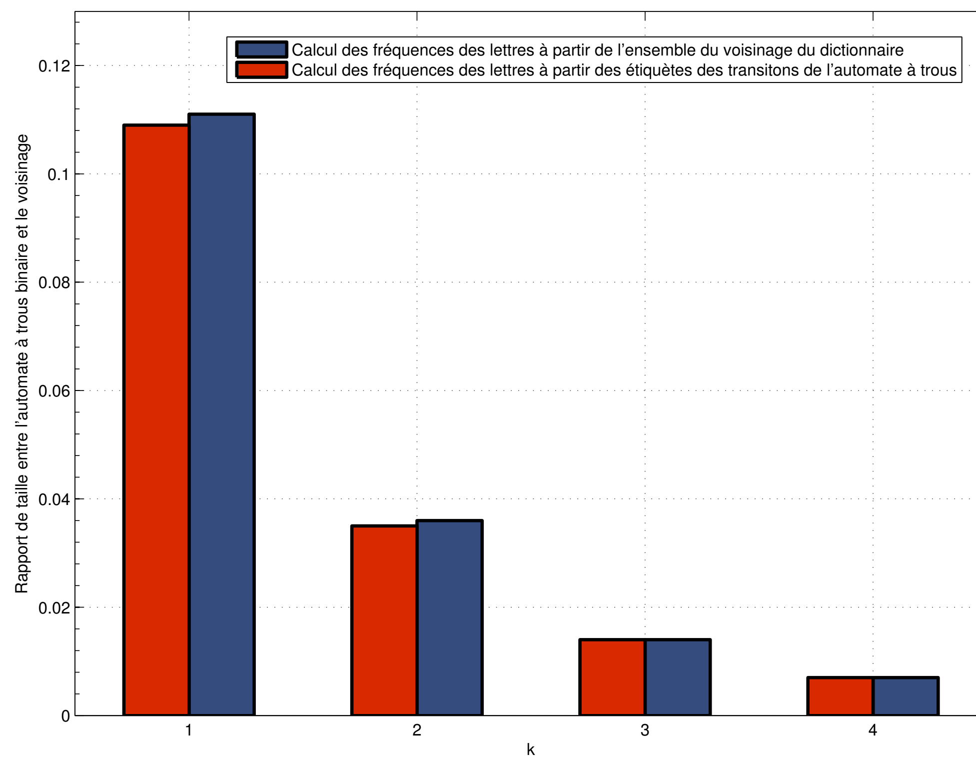


FIGURE 4.7 – Rapport de taille entre l'automate à trous binaire et le voisinage $\dot{H}_k(D)$ pour plusieurs petites valeurs de k.

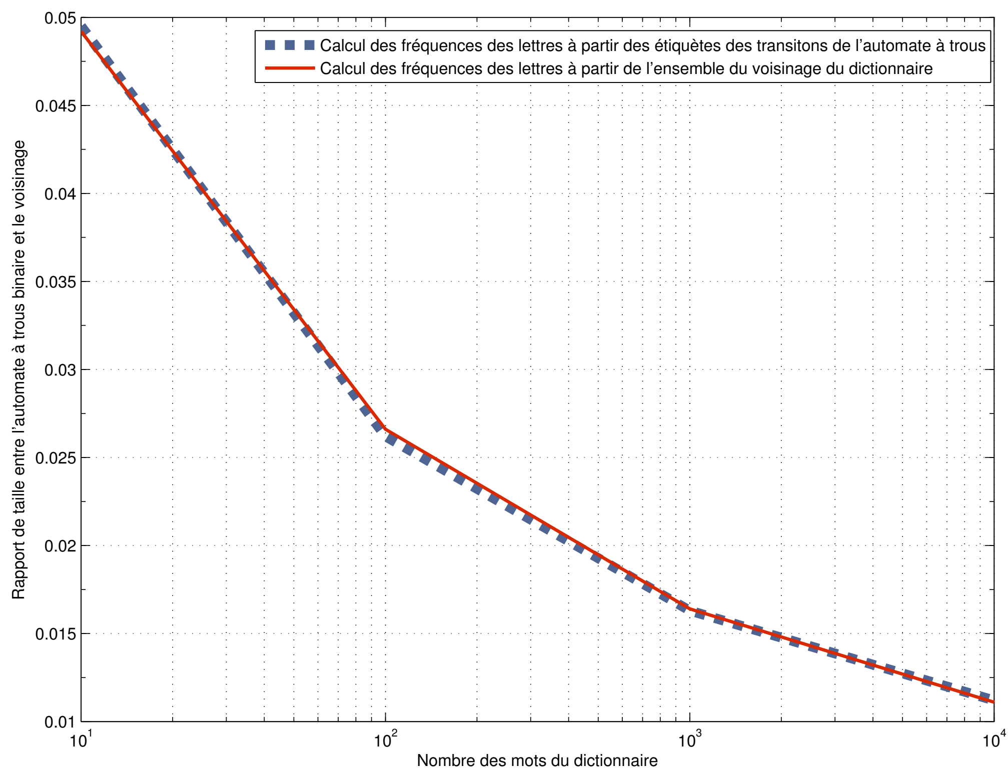


FIGURE 4.8 – Rapport de taille entre l'automate à trous binaire et le voisinage $\dot{H}_k(D)$ pour plusieurs taille du dictionnaire D.

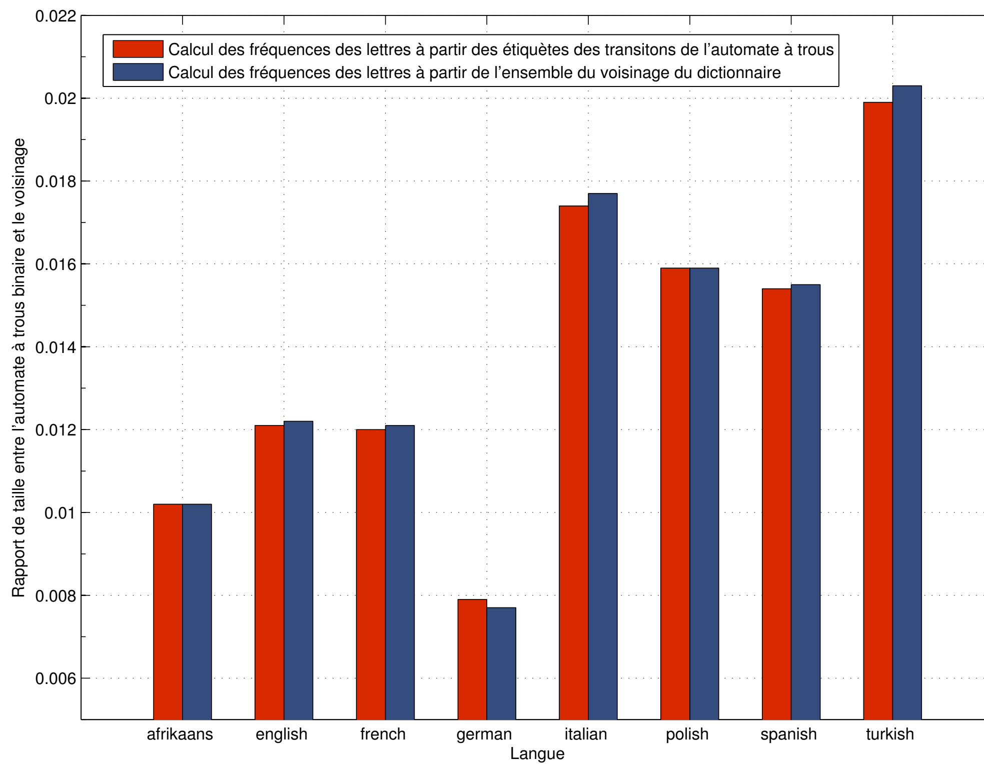


FIGURE 4.9 – Rapport de taille entre l'automate à trous binaire et le voisinage $\dot{H}_k(D)$ pour plusieurs dictionnaires de langues.

Les courbes illustrées dans les figures 4.10, 4.11, 4.12, 4.13, 4.14, 4.15 et 4.16 montrent la dépendance linéaire directe entre la taille de l'automate à trous binaire et les différents paramètres $|D|$, L et k, ainsi que le temps de recherche.

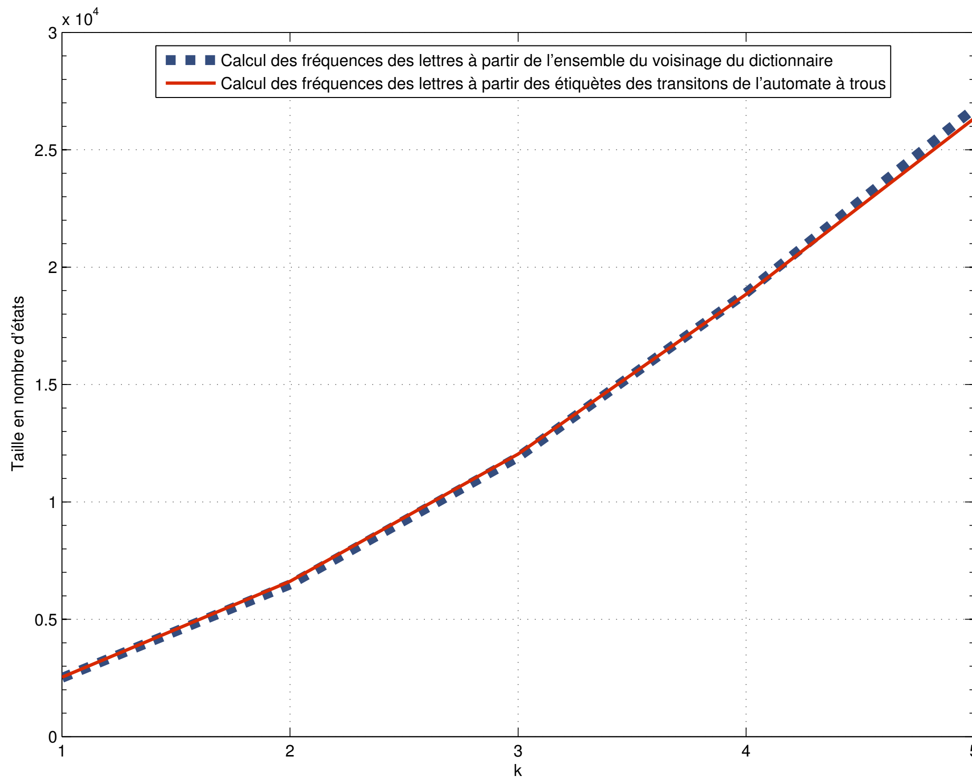


FIGURE 4.10 – Nombre d'états de l'automate à trous binaire pour plusieurs valeurs de k.

4.5 Conclusion

Les expérimentations que nous avons effectuées sur des dictionnaires pseudo-aléatoires de mots ou encore des dictionnaires issus de différentes langues montrent que la taille de l'automate à trous (la structure utilisée pour faire la recherche approchée au sein des dictionnaires) est linéaire par rapport au nombre d'erreurs k (k est un facteur multiplicatif). Contrairement à ce qu'on attend de la complexité théorique exponentielle en k. Nos expérimentations montrent aussi que le temps nécessaire à la recherche d'un motif est lui aussi linéaires. Le tableau 4.3 récapitule les résultats expérimentaux obtenus en les comparant avec les complexités théoriques.

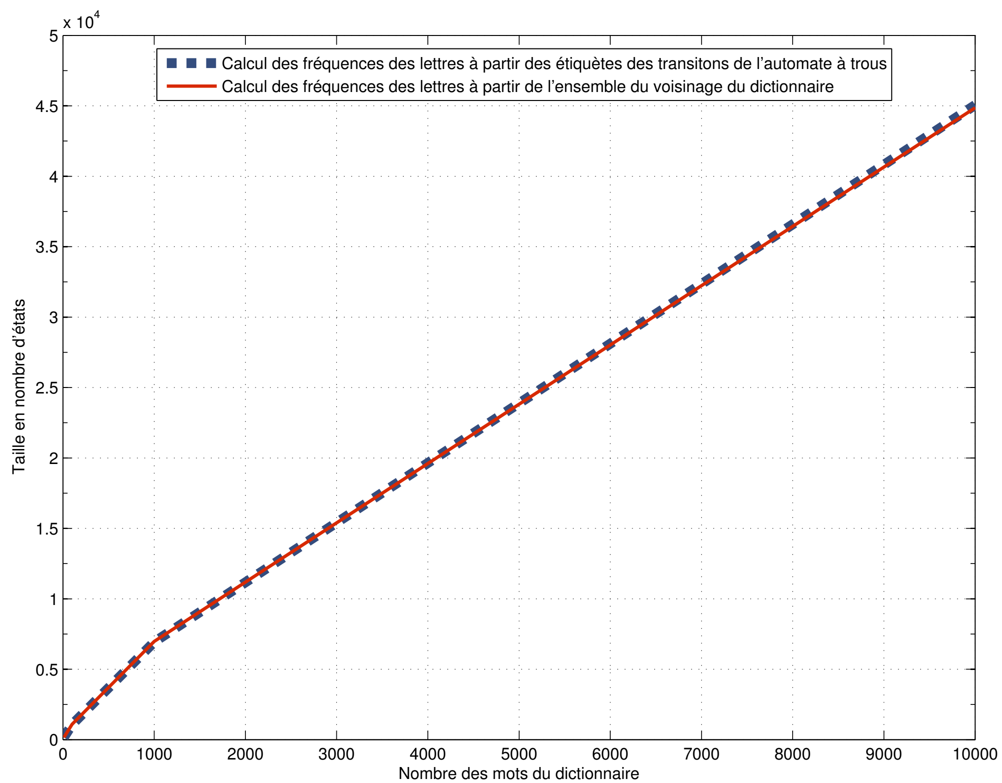


FIGURE 4.11 – Nombre d'états de l'automate à trous binaire pour plusieurs valeurs de $|D|$.

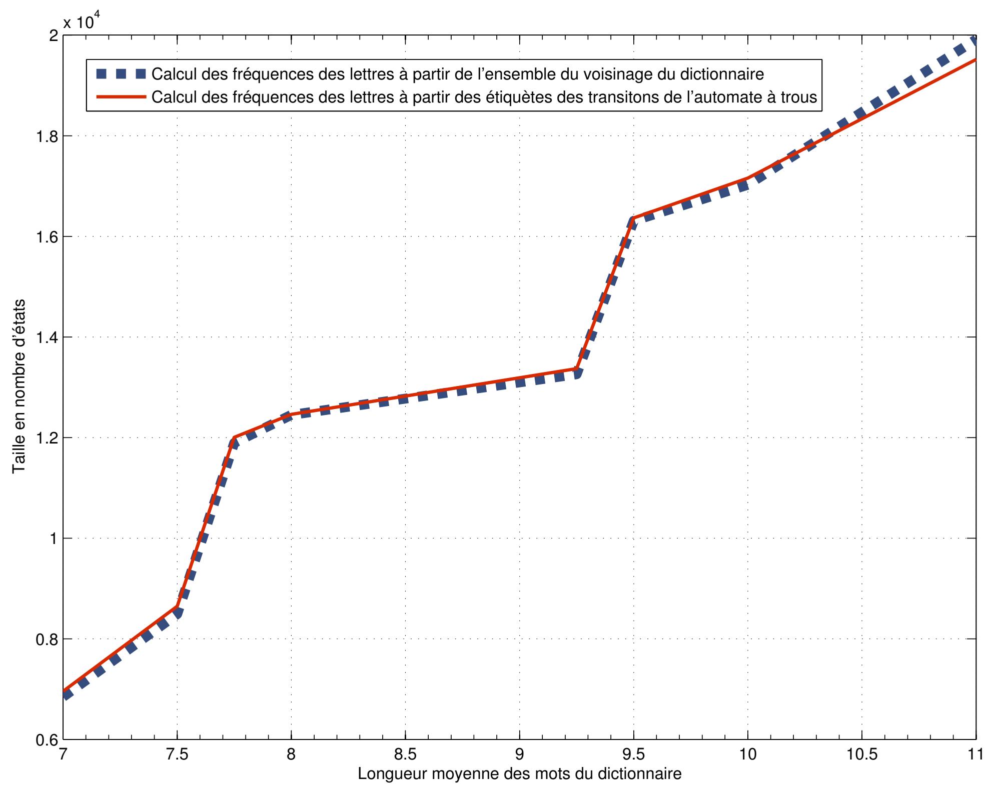


FIGURE 4.12 – Nombre d'états de l'automate à trous binaire selon L la longueur moyenne des mots du D.

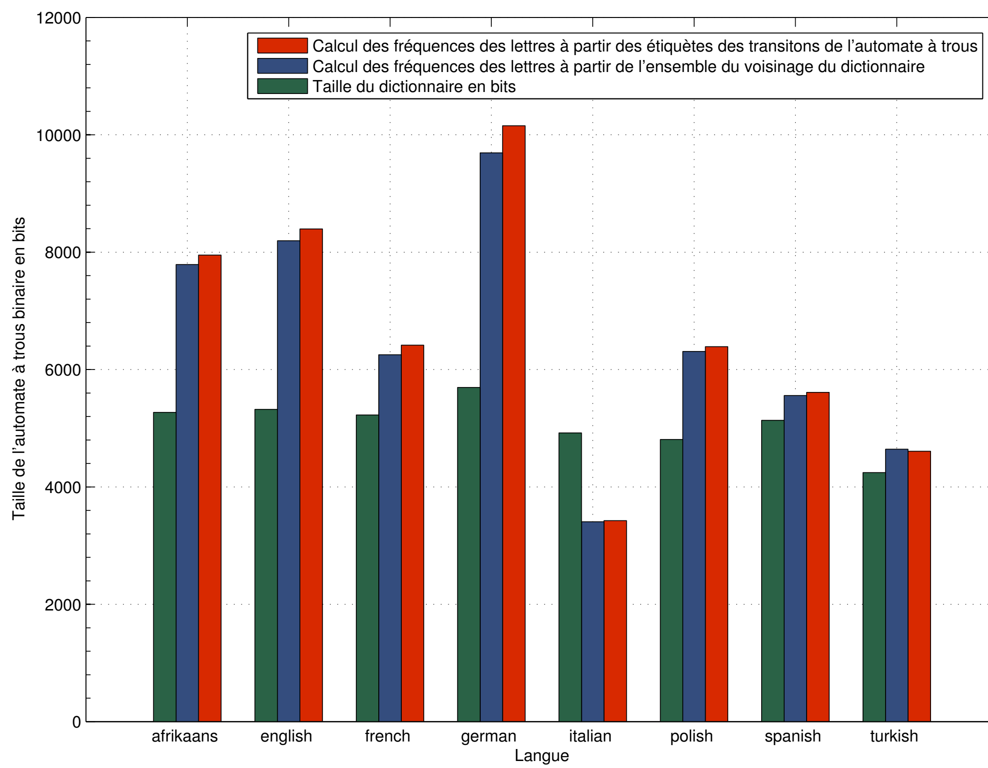


FIGURE 4.13 – La taille de l'automate à trous binaire pour $k = 1$ est proche de $1.29|D|$.

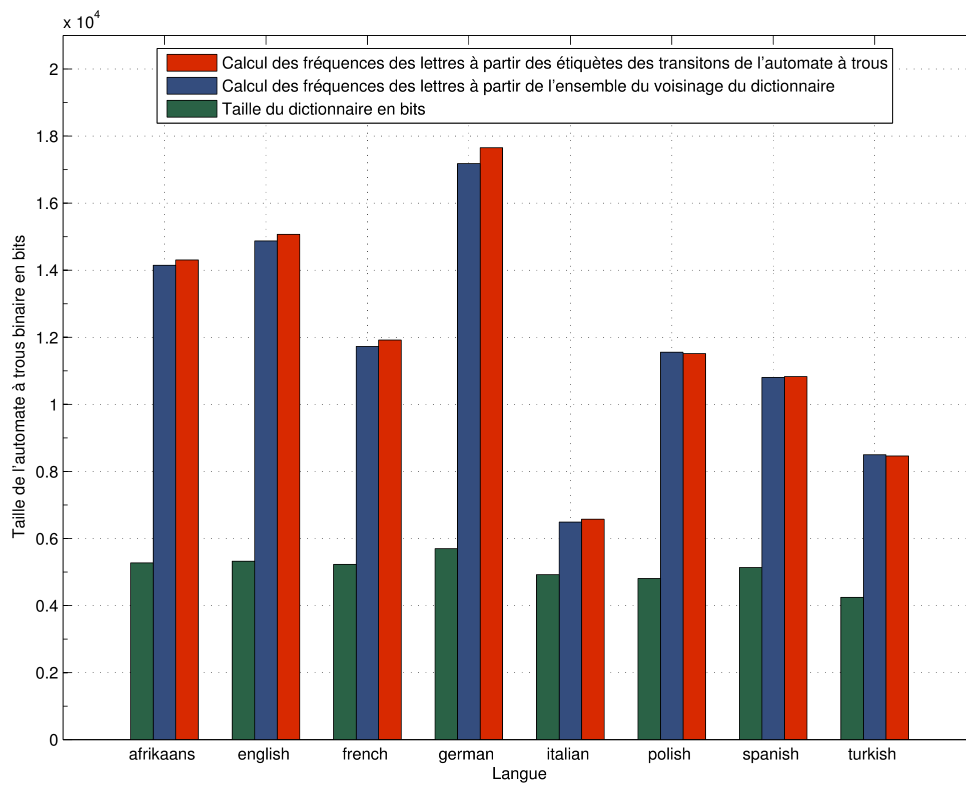


FIGURE 4.14 – La taille de l'automate à trous binaire pour $k = 2$ est proche de $2.34|D|$.

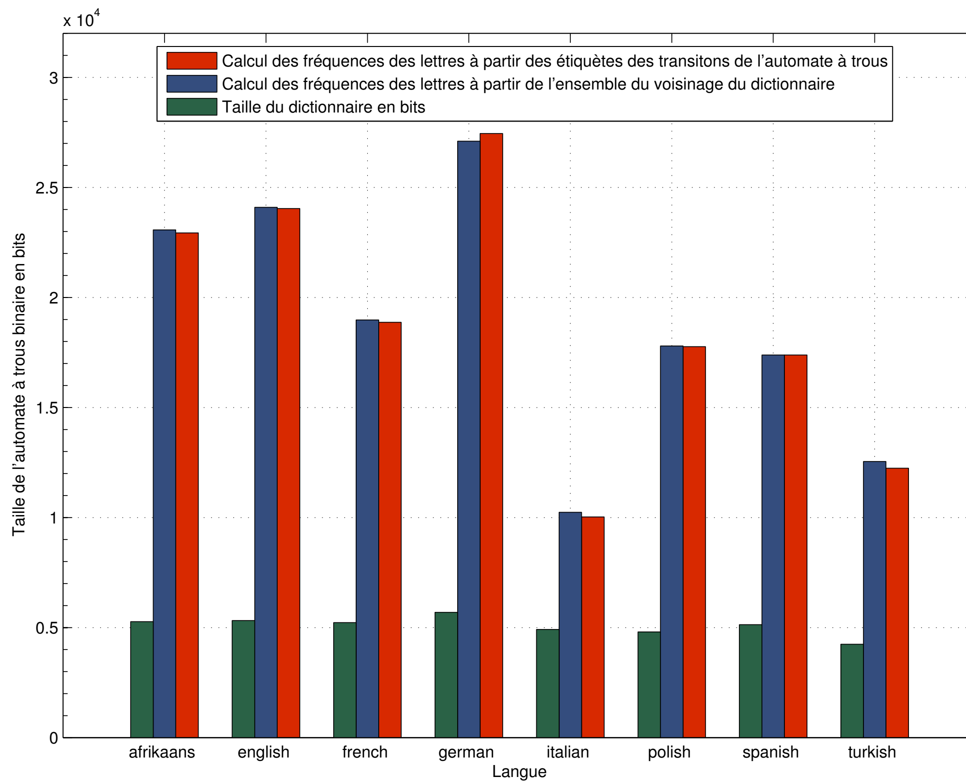


FIGURE 4.15 – La taille de l'automate à trous binaire pour $k = 3$ est proche de $3.66|D|$.

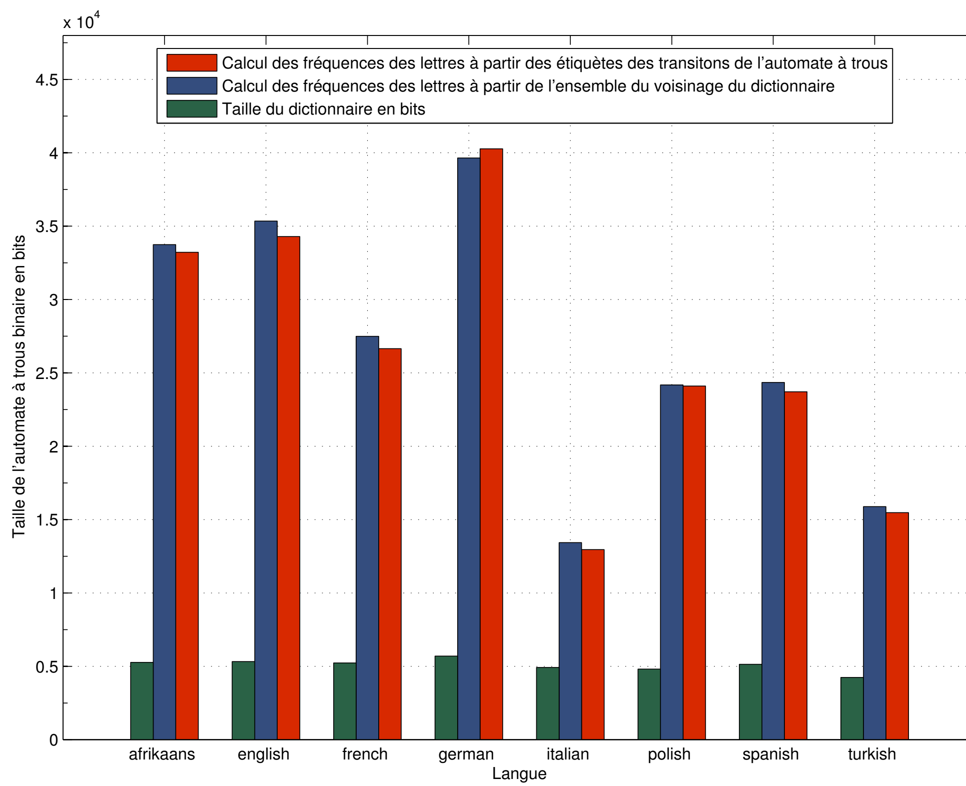


FIGURE 4.16 – La taille de l'automate à trous binaire pour $k = 4$ est proche de $5.10|D|$.

	Espace mémoire	**Temps de recherche**
Complexité théorique	$O(\|D\| \times L^k)$	$O(\|w\|^k)$
Estimation expérimentale	$c_1^{st} \times k \times \|D\| \times L$	$c_2^{st} \times k \times \|w\|$

TABLE 4.3 – Complexités théoriques et estimations pratiques expérimentales.

Dans cette étude, nous n'avons pas considéré le problème de l'optimisation de la construction de l'automate à trous. La construction naïve de cet automate consiste en :

1. engendrer les voisins à trous de chaque mot du dictionnaire,
2. construire l'automate non déterministe reconnaissant les mots à trous,
3. déterminiser et minimiser l'automate résultant.

La question de trouver un algorithme de construction plus efficace, permettant d'éviter la génération effective des mots à trous et/ou la phase de détermination, reste à traiter.

CHAPITRE 5

OUTIL GRAPHIQUE *WOMB* DE MANIPULATION DES AUTOMATES FINIS

DANS ce qui suit, nous allons décrire *womb*, un outil graphique de manipulation des automates finis, développé pour faciliter les tâches de manipulation et de dessin des automates pour la rédaction de ce manuscrit de thèse qui a été initialement développé dans un but plutôt pédagogique durant mes enseignements. Cet outil a été développé en C++ et est déjà fonctionnel sur les systèmes Windows.

5.1 Mode d'emploi

L'outil *womb* propose deux types de manipulation. Une permettant de dessiner un automate fini et l'autre permettant d'appliquer certaines transformations basiques sur l'automate dessiné.

Pour dessiner un automate, il suffit de choisir le bouton "état", et ensuite de cliquer sur l'espace de travail pour placer des états dessus. Ensuite on peut choisir le bouton "transition" pour lier les différents états tout en spécifiant les étiquettes. À tout moment on a la possibilité de choisir (sélectionner avec la souris) un sous-ensemble d'états (états colorés en rose) pour :

1. les déplacer sur le plan de travail,
2. les supprimer,
3. les rendre initiaux,
4. les rendre finaux ou
5. les rendre transitoires (ni initiaux ni finaux).

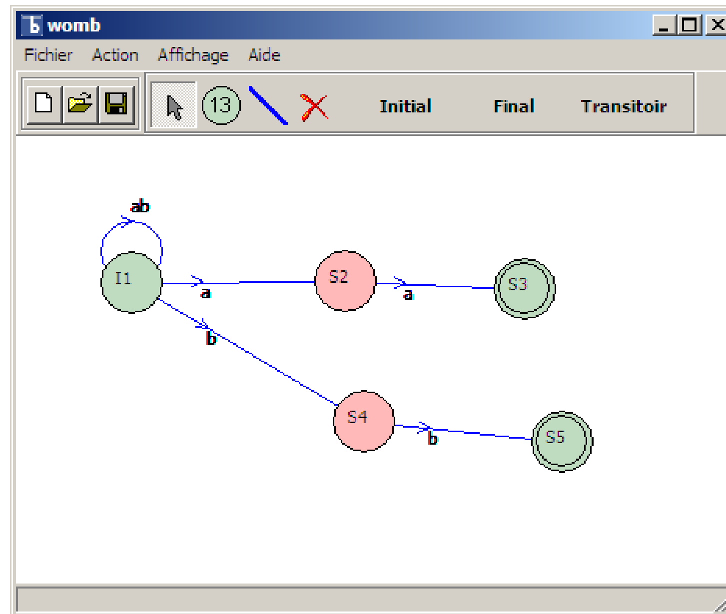


FIGURE 5.1 – Menu principal.

5.1.1 Menu Fichier

C'est le menu qui permet d'enregistrer l'automate dessiné ou d'ouvrir d'autres automates. L'outil *womb* sauvegarde l'automate dessiné dans un fichier binaire codant l'ensemble des états avec leur type (initial, final ou transitoire) et la table de transitions.

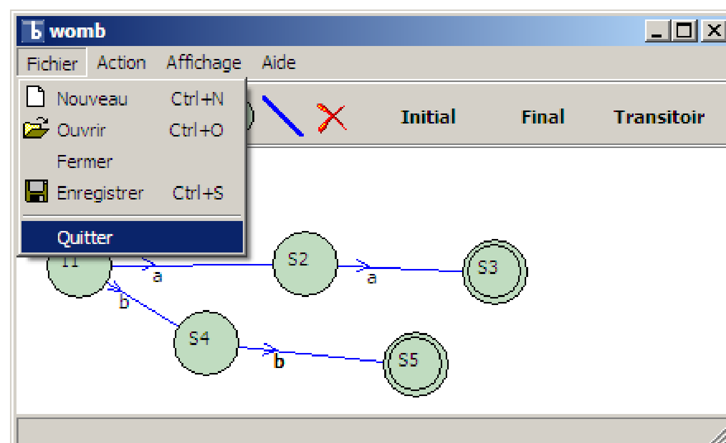


FIGURE 5.2 – Menu Fichier

5.1.2 Menu Action

C'est le menu regroupant les différents transformations basiques qui peuvent être effectuées par *womb* :

1. déterminisation,
2. élimination des transitions spontanées,
3. miroir,
4. compléter,
5. complément,
6. minimisation selon la méthode de Moore ou de Brzozowski.

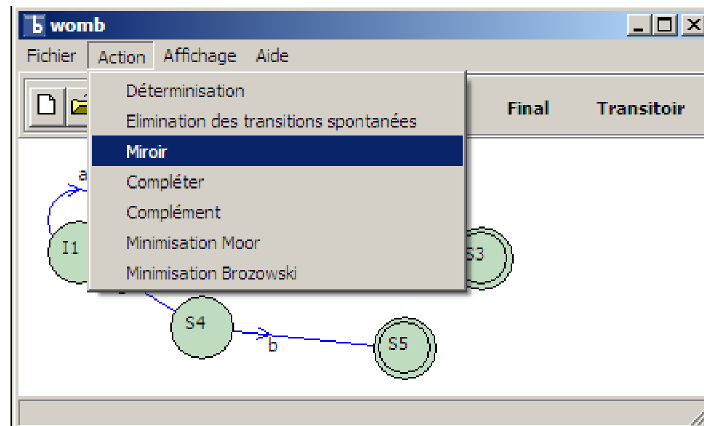


FIGURE 5.3 – Menu Fichier

5.1.3 Menu Affichage

L'utilisateur peut positionner où il veut les différents états de son automate sur le plan du travail (la zone d'affichage). De plus *womb* donne la possibilité de réorganiser les états selon deux méthodes : circulaire ou spirale (voir figure 5.6). Le sous-menu `ToGraphviz` de *womb* permet d'engendrer le format *.dot*, qui est un langage de description de graphe dans un format texte. Ce format peut être facilement utilisé pour engendrer du code LaTeX.

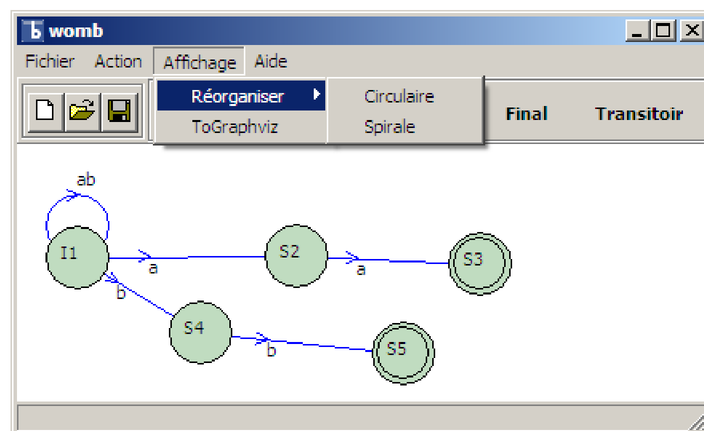


FIGURE 5.4 – Menu Affichage

5.2 Implémentation

Un automate fini sur un *alphabet* fini est composé d'un ensemble *fini* d'états, d'un ensemble d'états *initiaux*, d'un ensemble d'états *terminaux* (finaux) et d'un ensemble de *transitions* : des flèches étiquetées entre les différents états. Dans *womb*, un automate est représenté par un graphe, un tableau d'états qui contient les informations concernant un état donné, plus une matrice d'adjacence pour décrire les transitions de l'automate (les flèches du graphe).

Listing 5.1 – Types et déclarations (code en langage C++)

```
1  const MaxND=10;
2  const MaxD=1024;
3  const Sigma=28;
4  char MotVide='\$';
5  int Rn=20;
6  struct TNoeud
7  {
8      float X;
9      float Y;
10     int Initial;
11     int Final;
12     int Id;
13     int Poid;
14 };
15 int cptNoeud;
16 TNoeud N[MaxND];
17 short M_Adj[MaxND][MaxND][Sigma];
```

Comme la déterminisation d'un automate fini peut provoquer une explosion du nombre d'états, nous avons définit deux constantes `MaxND` et `MaxD` bornant le nombre d'états possibles dans un automate non déterministe et dans un automate déterministe respectivement. La constante `Sigma`, fixée par exemple à 28, précise la taille de l'alphabet sur lequel les automates seront définis. Le caractère `MotVide` est une variable permettant de fixer un symbole particulier pour le mot vide ε, le caractère $ par exemple. L'entier `Rn` fixe (en pixel) la longueur du rayon du cercle représentant un état.

La structure de données `TNoeud` permet de définir un état de l'automate :

1. Un état est représenté sur l'écran par un cercle dont les coordonnés du centre sont données par les deux variables X et Y.
2. La variable `Initial` permet de savoir si l'état est initial ou non. Sur le graphe de l'automate, un état initial est différencié par la lettre `I`.
3. La variable `Final` permet de savoir si l'état est terminal ou non. Sur le graphe de l'automate, un état terminal est discerné par un double cercle.
4. L'entier `Id` est un identifiant permettant de distinguer chaque état de l'automate.
5. La variable `Poid`, non utilisée pour cette version de l'outil, définit le poids de l'état pour une éventuelle extension aux automates à multiplicité.

Voici l'automate $\mathcal{A}$, un exemple d'automate sur lequel on va exécuter les transformations implémentées.

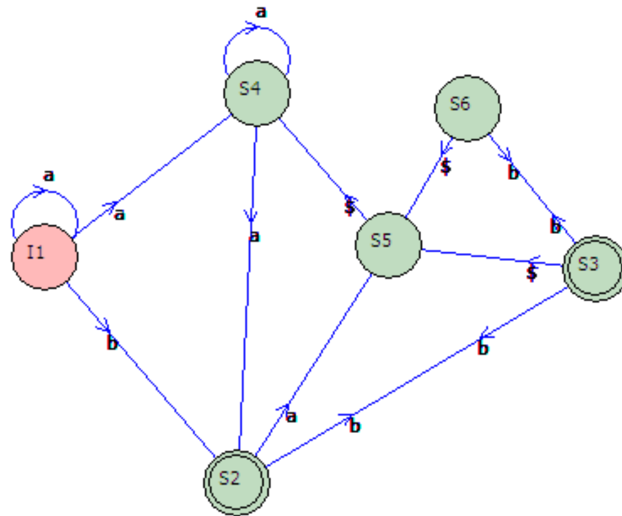


FIGURE 5.5 – Exemple d'un automate fini.

Dans `womb`, on manipule globalement un automate défini par :

1. `cptNoeud` : une variable qui compte le nombre d'états de l'automate,
2. `TNoeud N[MaxND]` : un tableau qui comporte l'ensemble des états de l'automate,
3. `M_Adj[MaxND][MaxND][Sigma]` : une table qui code les transitions de l'automate. Si, par exemple, la case `M_Adj[11][13][7]` vaut "un" cela veut dire que l'automate comporte une transition de l'état S_{11} vers l'état S_{13} étiquetée par la lettre a_7.

La fonction `Initial_state_verif()` permet de vérifier si l'automate dessiné par l'utilisateur comporte bien au moins un état initial.

Listing 5.2 – Vérification de l'existence d'un état initial

```
1 bool Initial_state_verif()
2 {
3     for (int j=0;j<cptNoeud;j++)
4         if(N[j].Initial==1) return true;
5     return false;
6 }
```

5.2.1 Affichage

Dans une application Win32, afin de faire des dessins, on peut passer par un objet de type `Canvas` défini dans le module `gdi32.dll`. Le fichier `gdi32.dll` est un fichier de bibliothèque de liaison dynamique valide enregistré pour Microsoft et utilisable pour C++. Cette bibliothèque contient des fonctions pour le GDI de Windows (*Graphics Device Interface*, interface graphique de Windows). Ceci est associé à des applications de dessin et de la gestion des polices, ce qui permet à Windows la création d'objets graphiques simples à deux dimensions.

L'affichage d'un automate sous *womb* est fait sur l'écran dans un objet de type `Canvas`. Le dessin dans une instance de cet objet est basé essentiellement sur les quatre méthodes suivantes :

1. `OutText` qui permet de dessiner du texte,
2. `Ellipse` qui permet de dessiner des cercles,
3. `lineTo` et `movTo` qui permettent de dessiner des lignes.

Le dessin de l'automate consiste à afficher les transitions de l'automate en utilisant la fonction `AfficheArete()`. Ensuite le dessin des états de l'automate.

Listing 5.3 – Dessin des transitions

```
1  void AfficheArete()
2  {
3      float a1; float b1;
4      for (int i=0;i<cptNoeud;i++)
5          for (int j=0;j<cptNoeud;j++)
6          {
7              //S est l'étiquette qui sera affichée au-dessus
8              //de la flèche représentant la transition.
9              String S="";
10             for (int k=0;k<Sigma;k++) if(M_Adj[i][j][k]==1)S=S+char(k);
11             if(S!="")
12             {
13                 if(i!=j)
14                 {
15                     float x1=N[i].X;    float y1=N[i].Y;
16                     float x2=N[j].X;    float y2=N[j].Y;
17                     float R=sqrt((x2-x1)*(x2-x1)+(y2-y1)*(y2-y1));
18                     float R1=min(10*r,R/3);
19                     a1=x1+(x2-x1)*R1/R; b1=y1+(y2-y1)*R1/R;
20                     float R2=R1-r;
21                     float a2=x1+(x2-x1)*R2/R;
22                     float b2=y1+(y2-y1)*R2/R;
23                     float tx1=a2+b1/2-b2/2; float ty1=b2-a1/2+a2/2;
24                     float tx2=a2-b1/2+b2/2; float ty2=b2+a1/2-a2/2;
25                     Canvas->MoveTo(a1,b1);  Canvas->LineTo(tx1,ty1);
26                     Canvas->MoveTo(a1,b1);  Canvas->LineTo(tx2,ty2);
27                     Canvas->MoveTo(N[i].X,N[i].Y);  Canvas->LineTo(N[j].X,N[j].Y);
28                     Canvas->TextOut(a1,b1,S.c_str());
29                 }
30                 else
31                 {
32                     a1=N[i].X;
33                     b1=N[i].Y-2*Rn;
34                     Canvas->Ellipse(N[i].X-Rn,N[i].Y-2*Rn,N[i].X+Rn,N[i].Y);
35                     Canvas->MoveTo(a1,b1);
36                     Canvas->LineTo(N[i].X-r,N[i].Y-2*Rn-r/3);
37                     Canvas->MoveTo(a1,b1);
38                     Canvas->LineTo(N[i].X-r,N[i].Y-2*Rn+2*r/3);
39                     Canvas->TextOut(a1,N[i].Y-2*Rn-2*r,S.c_str());
40                 }
41             }
42         }
43 }
```

La fonction `AfficheArete()` dessine une flèche droite pour une transition entre deux états distincts et dessine une flèche bouclée pour une transition sur le même état. Elle prépare une chaîne de caractères comportant l'étiquette de la transition à dessiner. Ensuite elle fait quelques calculs géométriques pour déterminer :

1. la position sur laquelle l'étiquette sera dessinée,
2. les coordonnées des points début et fin de la ligne représentant la transition,
3. l'inclinaison de la tête de la flèche représentant la transition.

La fonction `AfficheGraphe()` affiche la totalité de l'automate en dessinant ses transitions, et ensuite ses différents états tout en vérifiant le type de chaque état (initial, final, transitoire).

Listing 5.4 – Dessin de l'automate

```
1  void AfficheGraphe()
2  {
3      AfficheArete();
4      for (int i=0;i<cptNoeud;i++)
5      {
6
7          Canvas->Ellipse(N[i].X-Rn,N[i].Y-Rn,N[i].X+Rn,N[i].Y+Rn);
8
9          if(N[i].Final==1)
10             Canvas->Ellipse(N[i].X-0.80*Rn,N[i].Y-0.80*Rn,
11             N[i].X+0.80*Rn,N[i].Y+0.80*Rn);
12
13         String S;
14         if(N[i].Initial==1)
15             S="I";
16         else
17             S="S";
18         S=S+(i+1);
19         Canvas->TextOut(N[i].X-Rn/2,N[i].Y-Rn/2,S.c_str());
20     }
21 }
```

L'utilisateur peut positionner différents états de son automate sur le plan du travail (la zone d'affichage). De plus *womb* donne la possibilité de réorganiser les états selon deux méthodes différentes. La première, relativement simple, consiste à disposer uniformément les états tout autour d'un grand cercle ; cela formera un polygone régulier dont les sommets sont les états de l'automate.

Listing 5.5 – Réorganisation des états selon la méthode circulaire

```
1  void Reorganise1Graphe(){
2      if(cptNoeud>0){
3          float Gr=Height/3, tita=0, pas=cptNoeud;
4          pas=6.28/pas;
5          for (int i=0;i<cptNoeud;i++){
6              N[i].X=Gr*cos(tita)+Width/2;
7              N[i].Y=Gr*sin(tita)+Height/2;
8              tita=tita+pas;
9          }
10     }
11 }
```

L'autre méthode d'affichage est un peu différente, elle dispose les états tout au long d'une spirale. Cette méthode de disposition est plus adaptée si le nombre d'états est relativement grand.

Listing 5.6 – Réorganisation des états selon la méthode spirale

```
void Reorganise2Graphe()
{
    if(cptNoeud>0)
    {
        float Gr=Rn;
        float tita=0;
        float pas=20;
        pas=6.28/pas;
        for (int i=0;i<cptNoeud;i++)
        {
            N[i].X=Gr*cos(tita)+Width/2;
            N[i].Y=Gr*sin(tita)+Height/2;
            tita=tita+pas;
            Gr=Height/4+3*(i/20)*Rn;
        }
    }
}
```

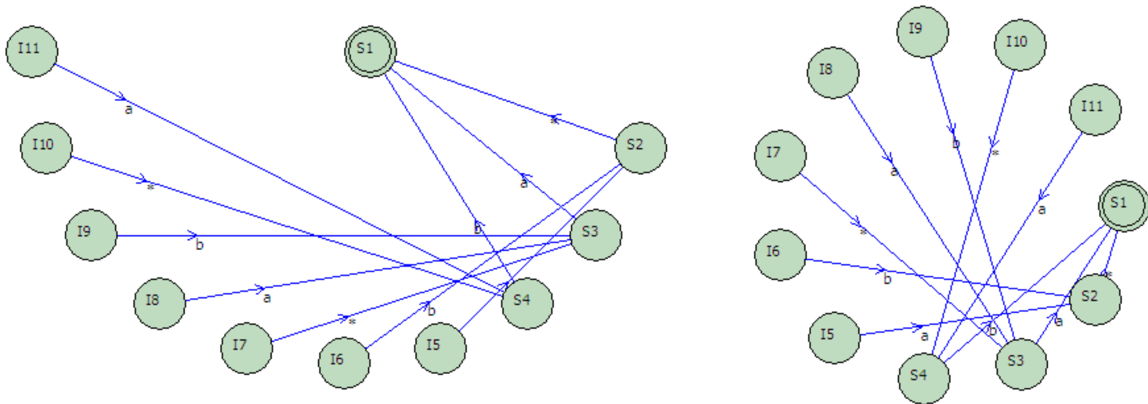


FIGURE 5.6 – Affichage spiral et circulaire.

5.2.2 Élimination des transitions spontanées

Les *automates asynchrones* sont une extension des automates finis où on autorise également le mot vide comme étiquette de flèche. L'avantage de cette convention est une plus grande souplesse dans la construction des automates. Elle a en revanche l'inconvénient d'accroître le non déterminisme des automates. Dans *womb* le mot vide est un caractère préalablement prédéfini par la variable `MotVide` et librement choisi par l'utilisateur, par exemple le caractère $, &, # ou autre.

L'algorithme d'élimination des ε-transitions est fondé essentiellement sur le calcul de la fermeture transitive de chaque état sur ε. La ε-fermeture transitive d'un état S_i est l'ensemble

de tous les états qu'on peut atteindre en lisant le mot ε^*. L'élimination des ε-transitions implique l'ajout de quelques transitions : si S_j appartient à la ε-fermeture transitive de S_i et si on a une transition de S_j vers S_k en lisant une lettre a_j différente du mot vide alors on doit ajouter une transition de S_i vers S_k par la lecture de a_j (voir section 1.2.1.3.2).

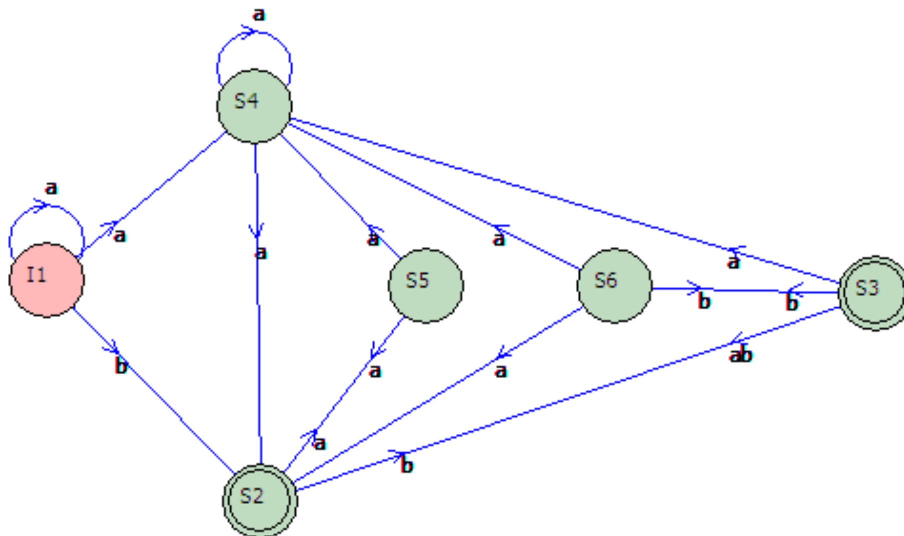


FIGURE 5.7 – Le synchrone de $\mathcal{A}$.

La fonction `Eliminer_transitions_spontanes()` implémente le calcul de la ε - fermeture transitive pour produire un automate reconnaissant le même langage mais sans les ε-transitions.

Listing 5.7 – Élimination des ε-transitions

```
void  Eliminer_transitions_spontanes()
{
    short T1[MaxND];
    short T2[MaxND];
    int boul=0;
    for (int i=0;i<cptNoeud;i++)
    {
        for (int j=0;j<cptNoeud;j++)
        {
            T1[j]=M_Adj[i][j][MotVide];
            T2[j]=M_Adj[i][j][MotVide];
        }
        do
        {
            for (int i0=0;i0<cptNoeud;i0++)
            {
                if (T1[i0]==1)
                {
                    for (int j0=0;j0<cptNoeud;j0++)
                        T2[j0]=T2[j0]|M_Adj[i0][j0][MotVide];
                }
            }
            boul=0;
            for (int k=0;k<cptNoeud;k++) boul=boul+T1[k]^T2[k];
            for (int k=0;k<cptNoeud;k++) T1[k]=T2[k];
        }while(boul!=0);
        for (int i0=0;i0<cptNoeud;i0++)
        {
            if (T1[i0]==1)
            {
                if (N[i].Initial==1)N[i0].Initial=1;
                if (N[i0].Final==1)N[i].Final=1;
                for (int j0=0;j0<cptNoeud;j0++)
                    for (int k=0;k<Sigma;k++)
                        M_Adj[i][j0][k]=M_Adj[i][j0][k]|M_Adj[i0][j0][k];
            }
        }
    }
    for (int i=0;i<cptNoeud;i++)
        for (int j=0;j<cptNoeud;j++)
            M_Adj[i][j][MotVide]=0;
}
```

5.2.3 Complétion

Plusieurs opération effectuées sur les automates finis nécessitent une version complétée de l'automate. Un automate est *complet* si, pour chaque état $p \in Q$ et pour chaque lettre a de l'alphabet, il existe au moins une transition étiquetée par a sortante de q. Si un automate n'est pas complet, on peut le compléter sans changer le langage reconnu en ajoutant un état puits s, et en ajoutant les transitions $\delta(p, a) = s$ pour tout couple (p, a) tel que $\delta(p, a)$ n'est pas défini. La fonction `Completer()` permet de compléter un automate fini (voir section 1.2.1.3.1).

Listing 5.8 – Complétion d'un automate

```
void  Completer()
{
    int s[Sigma];
    int t[Sigma];
    for(int i=0;i<Sigma;i++)t[i]=0;
    for (int i=0;i<cptNoeud;i++)
        for(int j=0;j<cptNoeud;j++)
            for (int k=0;k<Sigma;k++)
                if (M_Adj[i][j][k]==1) t[k]=1;
    bool complet=true;
    for (int i=0;i<cptNoeud;i++)
    {
        bool etatComplet=true;
        for (int k=0;k<Sigma;k++)
        {
            if(t[k]==1)
            {
                etatComplet=false;
                for(int j=0;j<cptNoeud;j++)
                    if (M_Adj[i][j][k]==1) etatComplet=true;
            }
        }
        if(!etatComplet)complet=false;
    }

    if(!complet)
    {
        //ajouter l'état puits
        N[cptNoeud].X=Canvas->Width/2;
        N[cptNoeud].Y=Canvas->Height/2;
        N[cptNoeud].Id=0;
        N[cptNoeud].Initial=0;
        N[cptNoeud].Final=0;
        for (int i=0;i<MaxND;i++)
            for (int k=0;k<Sigma;k++)
            {
                M_Adj[cptNoeud][i][k]=0;
                M_Adj[i][cptNoeud][k]=0;
            }
        cptId++;
        cptNoeud++;

        for (int i=0;i<cptNoeud;i++)
        {
            for(int j=0;j<Sigma;j++)s[j]=0;
            for (int j=0;j<cptNoeud;j++)
            {
                for (int k=0;k<Sigma;k++)
                    if (M_Adj[i][j][k]==1) s[k]=1;
            }
            for (int k=0;k<Sigma;k++)
                if (s[k]==0 && t[k]==1)M_Adj[i][cptNoeud-1][k]=1;
        }
    }
}
```

5.2.4 Déterminisation

Un *automate* est dit *déterministe* s'il possède un seul état initial et pour chaque état, il existe au plus une transition sortante de même étiquette (voir section 1.2.1.3.3). En pratique, pour déterminiser un automate, on ne construit pas en entier l'automate ensembliste qui est décrit dans la preuve de la déterminisation, mais on fait plutôt une recherche des descendants de l'état regroupant les états initiaux de l'automate, en maintenant la liste des états déjà construits de l'automate résultant plus l'ensemble des transitions qui reste à explorer et qui sont composées d'un état déjà construit et une lettre de l'alphabet. À chaque étape, on choisit un couple (état, lettre) et on construit l'état successeur. Si ce dernier n'existe pas dans la liste des états déjà construits on doit l'ajouter sinon on passe à la transition suivante. Il est clair qu'on a besoin ici d'une structure de données pratique pour représenter les ensembles car chaque état résultant est un sous-ensemble d'états de l'automate de départ.

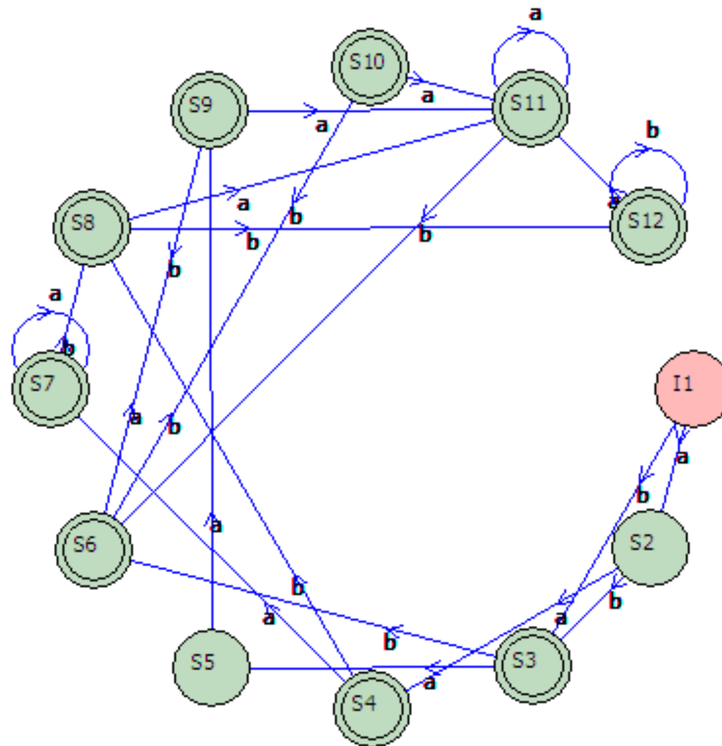


FIGURE 5.8 – Le déterminisé de $\mathcal{A}$.

Dans l'implémentation de *womb*, les ensembles sont représentés par des vecteurs de bits pour pouvoir produire rapidement l'union de deux ensembles et ceci grâce à une simple opération logique, le *ou* : *OR*. Cette représentation binaire permet facilement de savoir si deux ensembles sont égaux ou pas et cela grâce à l'opération logique du *ou exclusif* : *XOR*. La fonction `Determiniser()` implémente l'algorithme de la déterminisation ensembliste.

Listing 5.9 – Déterminisation d'un automate

```
void  Determiniser(){
    Eliminer_transitions_spontanes();
    unsigned __int64 Det[MaxD][Sigma+3];
    if(Initial_state_verif()){
        int final=0;
        for (int i=0;i<MaxD;i++)for (int k=0;k<Sigma+3;k++)Det[i][k]=-1;
        unsigned __int64 cptCourant=0, cptTotal, T1[MaxND], T2[MaxND], y=0;
        //Calcul de l'état initial
        for (int m=0;m<cptNoeud;m++)T1[m]=0;
        for (int j=0;j<cptNoeud;j++)if(N[j].Initial==1)T1[j]=1;
        y=T1[0];
        for (int i0=1;i0<cptNoeud;i0++){
            unsigned __int64 un=1;
            unsigned __int64 decalage=i0;
            y=y+T1[i0]*(unsigned __int64)(un<<decalage);
        }
        if(y!=0){
            Det[cptTotal][Sigma]=y;
            final=0;
            for (int i0=0;i0<cptNoeud;i0++)
                if((N[i0].Final==1)&&T1[i0]==1)final=1;
            Det[cptTotal][Sigma+1]=final;
            cptTotal++;
            //Début de la Déterminisation
            for (int i=0;i<cptTotal;i++){
                unsigned __int64 x=Det[i][Sigma];
                for (int i0=0;i0<cptNoeud;i0++){T1[i0]=x%2; x=x/2;}
                for (int k=0;k<Sigma;k++){
                    for (int m=0;m<cptNoeud;m++)T2[m]=0;
                    for (int j=0;j<cptNoeud;j++)
                        if(T1[j]==1)
                            for (int m=0;m<cptNoeud;m++)
                                T2[m]=T2[m]|M_Adj[j][m][k];
                    final=0;
                    for (int i0=0;i0<cptNoeud;i0++)
                        if((N[i0].Final==1)&&T2[i0]==1)final=1;
                    unsigned __int64 y=0;
                    y=T2[0];
                    for (int i0=1;i0<cptNoeud;i0++){
                        unsigned __int64 un=1;
                        unsigned __int64 decalage=i0;
                        y=y+T2[i0]*(unsigned __int64)(un<<decalage);
                    }
                    if(y!=0){
                        //Rechercher y dans Det
                        int boul=-1;
                        for (int i0=0;i0<cptTotal;i0++){
                            if(Det[i0][Sigma]==y){
                                boul=i0;
                                break;
                            }
                        }
                        if(boul==-1){
                            Det[i][k]=cptTotal;
                            Det[cptTotal][Sigma]=y;
                            Det[cptTotal][Sigma+1]=final;
                            cptTotal++;
                        }
                        else Det[i][k]=boul;
                    }
                }
            }
            cptNoeud=cptTotal;
        }
    }
}
```

5.2.5 Complémentation

Sur le même alphabet, l'opération de la complémentation d'un automate déterministe consiste à produire un automate ne reconnaissant que les mots non reconnus par l'automate initial. Pour ce faire il faut d'abord déterminiser et compléter l'automate initial et ensuite il suffit de rendre les états finaux en non-finaux et les états non-finaux en finaux. La fonction `Complement()` implémente ce procédé.

Listing 5.10 – Complémentation d'un automate

```
1  void  Complement()
2  {
3       Determiniser();
4       Completer();
5       for (int i=0;i<cptNoeud;i++)
6       {
7            if(N[i].Final!=1)N[i].Final=1;
8            else N[i].Final=0;
9       }
10 }
```

5.2.6 Automate transposé

Pour un automate fini $\mathcal{A}$, on note $\mathrm{tr}(\mathcal{A})$ l'automate transposé, qui est défini comme l'automate fini obtenu en échangeant états initiaux et états finaux et en inversant le sens des transitions. Le langage reconnu par l'automate $\mathrm{tr}(\mathcal{A})$ est le miroir du langage reconnu par l'automate $\mathcal{A}$. La fonction `Miroir()` permet de calculer l'automate transposé sous *womb*.

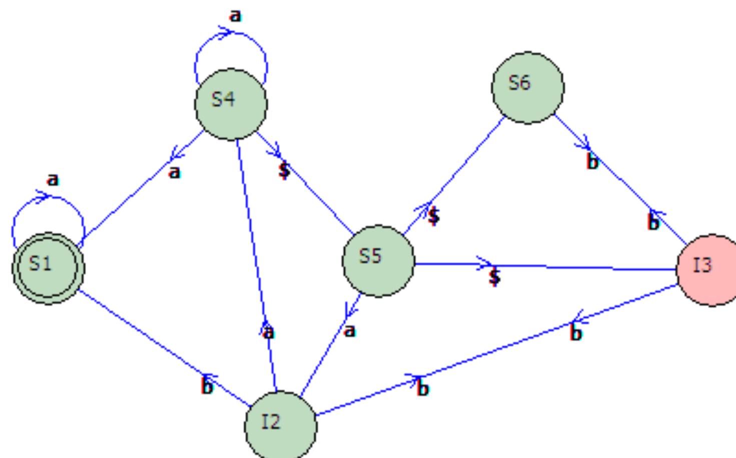


FIGURE 5.9 – Le transposé de $\mathcal{A}$.

Listing 5.11 – Automate transposé

```
void  Miroir()//transposé
{
    for (int i=0;i<cptNoeud;i++)
    {
        if((N[i].Initial==1)&&(N[i].Final!=1))
        {
            N[i].Initial=0;
            N[i].Final=1;
        }
        else
            if((N[i].Initial!=1)&&(N[i].Final==1))
            {
                N[i].Initial=1;
                N[i].Final=0;
            }
    }
    for (int i=0;i<cptNoeud;i++)
        for (int j=0;j<cptNoeud;j++)
            for (int k=0;k<Sigma;k++)
                M_Adj_Tmp[i][j][k]=M_Adj[j][i][k];

    for (int i=0;i<cptNoeud;i++)
        for (int j=0;j<cptNoeud;j++)
            for (int k=0;k<Sigma;k++)
                M_Adj[i][j][k]=M_Adj_Tmp[i][j][k];
}
```

5.2.7 Minimisation

L'ensemble des automates déterministes reconnaissant un langage donné admet un plus petit élément par le nombre d'états ; ce plus petit automate, appelé *automate déterministe minimal*, est unique à un isomophisme près (voir section 1.2.1.3.6). Dans *womb* le calcul de l'automate minimal peut se faire soit par la méthode de Moore [Moo56], soit par la méthode de la double détermination de Brzozowski [Brz64]. La fonction `MinimisationMoore()` permet de minimiser l'automate créer par l'utilisateur en se basant sur l'algorithme de Moore qui consiste principalement à fusionner les états équivalents. La deuxième méthode de minimisation est celle de la double déterminisation introduite par Brzozowski. Soit $\mathcal{B}$ un automate déterministe, complet et accessible. On peut montrer que $\mathrm{det}(\mathrm{tr}(\mathcal{B}))$ est minimal, par conséquent l'automate $\mathcal{C} = \mathrm{det}(\mathrm{tr}(\mathrm{det}(\mathrm{tr}(\mathcal{A}))))$ est l'automate minimal de $L(\mathcal{A})$.

Listing 5.12 – Minimisation de Brzozowski

```
void MinimisationBrzozowski(){
   Miroir();
   Determiniser();
   Miroir();
   Determiniser();
```

Listing 5.13 – Minimisation de Moore

```
void  MinimisationMoore()
{
    short MinimiserMoore[MaxND][MaxND];
    unsigned __int64 Det[MaxD][Sigma+3];
    for (int i=0;i<cptNoeud;i++)
        for (int j=0;j<cptNoeud;j++)
        {
            if((N[i].Final==1&&N[j].Final!=1)||(N[i].Final!=1&&N[j].Final==1))
                MinimiserMoore[i][j]=1;
            else MinimiserMoore[i][j]=0;
        }
    bool mark=false;
    do
    {
        mark=false;
        for (int i=0;i<cptNoeud;i++)
            for (int j=0;j<cptNoeud;j++)
                if(MinimiserMoore[i][j]==0)
                    for (int k=0;k<Sigma;k++)
                        for (int p=0;p<cptNoeud;p++)
                            for (int q=0;q<cptNoeud;q++)
                                if(MinimiserMoore[p][q]==1 && M_Adj[i][p][k]==1
                                && M_Adj[j][q][k]==1)
                                {
                                    MinimiserMoore[i][j]=1;
                                    mark=true;
                                }
    }while(mark);
    int cptMinimiserMoore=cptNoeud;
    int R[MaxND];
    for (int i=0;i<cptNoeud;i++)R[i]=-1;
    int cptNoeudMin=0;
    for (int i=0;i<cptNoeud;i++)
        if(R[i]==-1)
        {
            R[i]=cptNoeudMin;
            for (int j=0;j<cptNoeud;j++)
                if(MinimiserMoore[i][j]!=1)R[j]=cptNoeudMin;
            cptNoeudMin++;
        }
    for (int i=0;i<cptNoeudMin;i++)
    {
        unsigned __int64 y=0;
        for (int j=0;j<cptNoeud;j++)
        {
            unsigned __int64 un=1;
            unsigned __int64 decalage=j;
            if(R[j]==i)
            {
                y=y+(unsigned __int64)(un<<decalage);
                if(N[j].Final==1)Det[i][Sigma+1]=1;
                if(N[j].Initial==1)Det[i][Sigma+2]=1;
            }
        }
        Det[i][Sigma]=y;
    }
    for (int i=0;i<cptNoeud;i++)
        for (int j=0;j<cptNoeud;j++)
            for (int k=0;k<Sigma;k++)
                if(M_Adj[i][j][k]==1)Det[R[i]][k]=R[j];

}
```

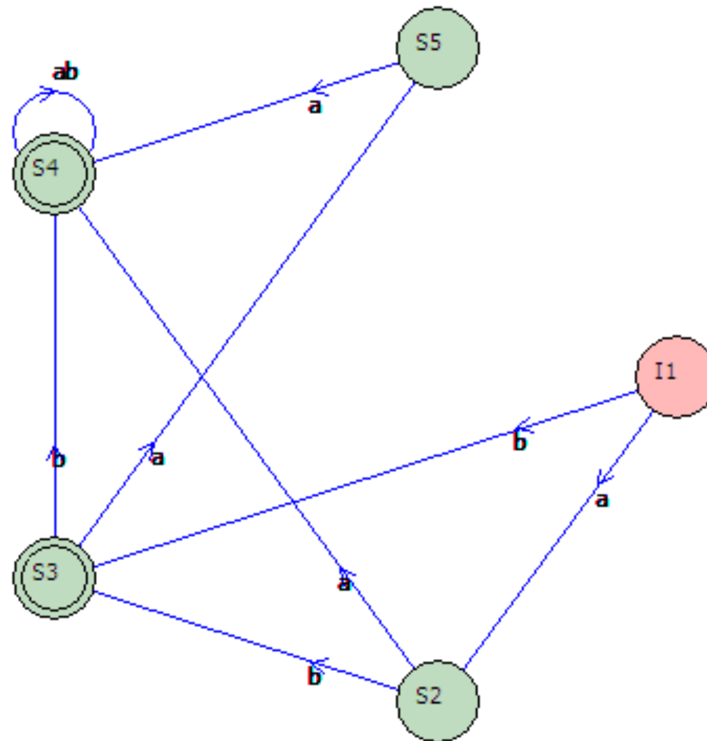


FIGURE 5.10 – Le minimal de $\mathcal{A}$.

5.3 Conclusion

L'outil *womb* est toujours en évolution. De nouvelles futures versions vont inclure probablement des implémentations plus efficaces et optimisées des opérations basiques sur les automates finis et d'autres opérations plus complexes vont être ajoutées. Elles permettront éventuellement la manipulation des automates finis à multiplicité et des transducteurs. Elles comprendront certainement d'autres formats d'import/export et d'autres modes d'affichage.

CONCLUSION

DANS ce manuscrit on a détaillé un ensemble de contributions théoriques et quelques études expérimentales portant essentiellement sur la théorie des automates et la logique mathématique.

Les premières résultats montrent l'automaticité et la décidabilité de la théorie de certaines structures portant sur un alphabet infini dénombrable, en développant des codages de plus en plus raffinés, qui conservent la décidabilité et l'automaticité de ces structures tout en ajoutant des prédicats plus complexes et des relations plus expressives, comme la relation de préfixe, le prédicat $\mathfrak{clone}$, qui est vrai lorsqu'un mot se termine par deux lettres identiques, et le prédicat $\mathfrak{diff}$, qui est lui vrai quand toutes les lettres d'un mot sont (deux à deux) distinctes. D'autre résultats d'indécidabilité, et des résultats de décidabilité pour les théories monadiques du second ordre, ont été également présentés :

Structure	**Théorie** FO	**Théorie** MSO
$\mathfrak{S}_1 = (\Sigma^*; \prec, \mathfrak{clone}, \mathfrak{less}, \{\mathfrak{mod}_{p,q} \mid p, q \in \mathbb{N}\}, \sim, \oplus)$	Décidable	?
$\mathfrak{S}_2 = (\Sigma^*; \prec, \mathfrak{clone}, \{\mathfrak{diff}_i \mid i > 2\})$	Décidable	Décidable
$\mathfrak{S}_3 = (\Sigma^*; \prec, \mathfrak{clone}, \{\mathfrak{diff}_i \mid i > 2\}, \sim)$	Décidable	?
$\mathfrak{S}_4 = (\Sigma^*; \prec, \mathfrak{clone}, \mathfrak{diff})$	Décidable	Décidable
$\mathfrak{S}_5 = (\Sigma^*; \prec, \mathfrak{clone}, \mathfrak{diff}, \sim)$	Décidable	?
$\mathfrak{S}_6 = (\Sigma^*; \prec, \{R^{\oplus}_{f_i} \mid i > 0\}, \sim)$	Décidable	?
$\mathfrak{S}_7 = (\Sigma^*; \prec, <, \mathfrak{dec}_\star)$	Décidable	Indécidable
$\mathfrak{S}_8 = (\Sigma^*; \prec, <, \mathfrak{dec}_\star, \sim)$	Indécidable	Indécidable
$\mathfrak{S}_9 = (\Sigma^*; \prec, \mathfrak{clone}, \mathfrak{lastNew}, \mathfrak{firstZero}, \sim, \ominus)$	Indécidable	Indécidable

TABLE 5.1 – Principaux résultats

D'autres questions restent pour l'instant sans réponse comme celui de l'automaticité de la structure $(\Sigma^*; \prec, \mathfrak{clone}, \sim)$ ou encore la décidabilité de la théorie logique du premier ordre de la structure $(\Sigma^*; \prec, \mathfrak{clone}, \{\mathfrak{diff}_i \mid i > 2\}, \mathfrak{diff})$ et la théorie monadique du seconde ordre des structures :
- $\mathfrak{S}_1 = (\Sigma^*; \prec, \mathfrak{clone}, \mathfrak{less}, \{\mathfrak{mod}_{p,q} \mid p, q \in \mathbb{N}\}, \sim, \oplus)$,
- $\mathfrak{S}_3 = (\Sigma^*; \prec, \mathfrak{clone}, \{\mathfrak{diff}_i \mid i > 2\}, \sim)$,
- $\mathfrak{S}_5 = (\Sigma^*; \prec, \mathfrak{clone}, \mathfrak{diff}, \sim)$,
- $\mathfrak{S}_6 = (\Sigma^*; \prec, \{R^{\oplus}_{f_i} \mid i > 0\}, \sim)$.

Dans un deuxième temps nous avons montré comment les arbres binaires peuvent être utilisés pour concevoir un algorithme rapide pour calculer un automate avec un nombre réduit de transitions, asymptotiquement minimal, reconnaissant le langage triangulaire $L(E_n)$ dénoté par l'expression rationnelle $E_n = (1 + \varepsilon) \cdot (2 + \varepsilon) \cdot (3 + \varepsilon) \cdots (n + \varepsilon)$. Nous avons bien vérifié que notre algorithme donne le nombre minimal de transitions pour les petites valeur de n et nous avons montré que notre réduction est asymptotiquement une minimisation dans la mesure où le nombre de transitions est équivalent à $n(\log_2(n))^2$, qui correspond, à la fois à la borne supérieure et à la borne inférieure. Cependant, montrer que notre algorithme de réduction est une minimisation pour les automates triangulaires reste ouvert (conjecture 1). Une suite possible de cette étude consiste à généraliser le concepts du CFS au automates non homogènes, de proposer des heuristiques de réduction pour les automates homogènes non triangulaires, ou encore de trouver d'autres applications pour les P-arbres et les Z-arbres (nous avons, par exemple, isolé une sous-classe de la classe des P-arbres qui est en remarquable bijection avec l'ensemble des nombres premiers (voir section 3.3)).

En plus des problèmes théoriques traités dans la première partie, nous nous somme intéressé par le problème pratique de la recherche approximative d'un motif dans un dictionnaire de mots. Nous avons proposé une étude expérimentale d'une structure de données particulière, appelée *automate à trous binaire* permettant d'engendrer, à partir d'un dictionnaire de mots, la liste des mots voisins du mot recherché. Théoriquement et dans des cas extrêmes (pire des cas), cette structure peut être lourde en recherche ou grande en volume. Notre étude pratique effectuée sur des dictionnaires de langues naturelles et des dictionnaires pseudo-aléatoires montre que l'automate à trous binaire permet une recherche en un temps et un espace expérimentalement linéaires.

Enfin nous avons décrit *womb*, un outil graphique de manipulation des automates finis, développé en C++ et est déjà fonctionnel sur les systèmes Windows, qui a été initialement développé dans un but pédagogique pour l'enseignement, ensuite utilisé pour faciliter les tâches de manipulation et de dessin des automates pour la rédaction de ce manuscrit de thèse.

BIBLIOGRAPHIE

[ALI02] *Automata, Logics, and Infinite Games : A Guide to Current Research [outcome of a Dagstuhl seminar, February 2001]*, volume 2500 of *Lecture Notes in Computer Science*. Springer, 2002.

[Apo76] T.M. Apostol. *Introduction to Analytic Number Theory*. Springer International Student Edition. Springer, 1976.

[BB01] Dietmar Berwanger and Achim Blumensath. The monadic theory of tree-like structures. In *Automata, Logics, and Infinite Games* [ALI02], pages 285–302.

[BBC92] D. Beauquier, J. Berstel, and P. Chrétienne. *éléments d'algorithmique*. Masson, 1992.

[BCC+03] Marco Brambilla, Stefano Ceri, Sara Comai, Piero Fraternali, and Ioana Manolescu. Specification and Design of Workflow-Driven Hypertexts. *J. Web Eng.*, 1(2) :163–182, 2003.

[BDM+11] Mikolaj Bojanczyk, Claire David, Anca Muscholl, Thomas Schwentick, and Luc Segoufin. Two-variable logic on data words. *ACM Trans. Comput. Log.*, 12(4) :27, 2011.

[Bès08] Alexis Bès. An application of the Feferman-Vaught theorem to automata and logics for words over an infinite alphabet. *Logical Methods in Computer Science*, 4(1 :8) :1–23, 2008.

[BG00] Achim Blumensath and Erich Grädel. Automatic structures. In *LICS12* [LIC00], pages 51–62.

[BHJS07] Ahmed Bouajjani, Peter Habermehl, Yan Jurski, and Mihaela Sighireanu. Rewriting systems with data. In *FCT* [FCT07], pages 1–22.

[BHM03] Ahmed Bouajjani, Peter Habermehl, and Richard Mayr. Automatic verification of recursive procedures with one integer parameter. *Theor. Comput. Sci.*, 295 :85–106, 2003.

[BLSS03] Michael Benedikt, Leonid Libkin, Thomas Schwentick, and Luc Segoufin. Definable Relations and First-order Query Languages over Strings. *J. ACM*, 50(5) :694–751, 2003.

[BM77] Robert S. Boyer and J. Strother Moore. A fast string searching algorithm. *Commun. ACM*, 20(10) :762–772, 1977.

[BM07] Aaron R. Bradley and Zohar Manna. *The Calculus of Computation : Decision Procedures with Applications to Verification*. Springer-Verlag New York, Inc., 2007.

[BMS+06] Mikolaj Bojanczyk, Anca Muscholl, Thomas Schwentick, Luc Segoufin, and Claire David. Two-variable logic on words with data. In *LICS* [LIC06], pages 7–16.

[BMSS09] Mikolaj Bojanczyk, Anca Muscholl, Thomas Schwentick, and Luc Segoufin. Two-variable logic on data trees and XML reasoning. *J. ACM*, 56(3), 2009.

[Brz64] Janusz A. Brzozowski. Derivatives of Regular Expressions. *J. ACM*, 11(4) :481–494, 1964.

[Büc60] J. R. Büchi. Weak second-order arithmetic and finite automata. *Z. Math. Logik und grundl. Math.*, 6 :66–92, 1960.

[CFB+02] Stefano Ceri, Piero Fraternali, Aldo Bongio, Marco Brambilla, Sara Comai, and Maristella Matera. *Designing Data-Intensive Web Applications*. Morgan Kaufmann Publishers Inc., 2002.

[CG09a] Christian Choffrut and Serge Grigorieff. The "equal last letter" predicate for words on infinite alphabets and classes of multitape automata. *Theor. Comput. Sci.*, 410(30-32) :2870–2884, 2009.

[CG09b] Christian Choffrut and Serge Grigorieff. Finite n-tape automata over possibly infinite alphabets : Extending a theorem of Eilenberg et al. *Theor. Comput. Sci.*, 410(1) :16–34, 2009.

[CGL04] Richard Cole, Lee-Ad Gottlieb, and Moshe Lewenstein. Dictionary matching and indexing with errors and don't cares. In *STOC* [STO04], pages 91–100.

[CHL01] M. Crochemore, C. Hancart, and T. Lecroq. *Algorithmique du texte*. Vuibert informatique, 2001.

[CL10] Edgar Chávez and Stefano Lonardi, editors. *String Processing and Information Retrieval - 17th International Symposium, SPIRE 2010, Los Cabos, Mexico, October 11-13, 2010. Proceedings*, volume 6393 of *Lecture Notes in Computer Science*. Springer, 2010.

[CLS+10] Ho-Leung Chan, Tak Wah Lam, Wing-Kin Sung, Siu-Lung Tam, and Swee-Seong Wong. Compressed Indexes for Approximate String Matching. *Algorithmica*, 58(2) :263–281, 2010.

[Cox07] Russ Cox. Minimal number of edges in ε-free non-deterministic finite automata (NFA) for regular expression $(1 + \varepsilon) \cdot (2 + \varepsilon) \cdot (3 + \varepsilon) \cdots (n + \varepsilon)$. *The On-Line Encyclopedia of Integer Sequences. Sequence A129403*, 2007. http://oeis.org/A129403.

[COZ09] Jean-Marc Champarnaud, Faissal Ouardi, and Djelloul Ziadi. An efficient computation of the equation K-automaton of a regular K-expression. *Fundamenta Mathematica*, 90(1-2) :1–16, 2009.

[DLN07] Stéphane Demri, Ranko Lazić, and David Nowak. On the freeze quantifier in constraint LTL : Decidability and complexity. *Information and Computation*, 205(1) :2–24, 2007.

[DLS08] Stéphane Demri, Ranko Lazic, and Arnaud Sangnier. Model checking freeze ltl over one-counter automata. In *FoSSaCS* [FoS08], pages 490–504.

[EES69] S Eilenberg, C.C Elgot, and J.C Shepherdson. Sets recognized by n-tape automata. *Journal of Algebra*, 13(4) :447 – 464, 1969.

[Ehr61] Andrzej Ehrenfeucht. An Application of Games to the Completeness Problem for Formalized Theories. *Fundamenta Mathematicae*, 49 :129–141, 1961.

[Eil74] S. Eilenberg. *Automata, languages, and machines*. Pure and Applied Mathematics. Elsevier Science, 1974.

[End72] Herbert B. Enderton. *A Mathematical Introduction to Logic*. Academic Press, 1972.

[Far97] M. Farach. Optimal suffix tree construction with large alphabets. In *Proceedings of the 38th Annual Symposium on Foundations of Computer Science*, page 137. IEEE Computer Society, 1997.

[FCT07] *Fundamentals of Computation Theory, 16th International Symposium, FCT 2007, Budapest, Hungary, August 27-30, 2007, Proceedings*, volume 4639 of *Lecture Notes in Computer Science*. Springer, 2007.

[FoS08] *Foundations of Software Science and Computational Structures, 11th International Conference, FOSSACS 2008, Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2008, Budapest, Hungary, March 29 - April 6, 2008. Proceedings*, volume 4962 of *Lecture Notes in Computer Science*. Springer, 2008.

[FR75] Jeanne Ferrante and Charles Rackoff. A Decision Procedure for the First Order Theory of Real Addition with Order. *Society for Industrial and Applied Mathematics, J. Comput.*, 4(1) :69–76, 1975.

[Fra54] R. Fraïssé. Sur Quelques Classifications des Systèmes de Relations. *Publ. Sci. Univ. Alger. Sér. A*, 1 :35–182, 1954.

[FS09] Philippe Flajolet and Robert Sedgewick. *Analytic Combinatorics*. Cambridge University Press, 2009.

[FV59] S. Feferman and R. Vaught. The first order properties of products of algebraic systems. *Fundamenta Mathematicae*, 47(1) :57–103, 1959.

[Gef03] Viliam Geffert. Translation of binary regular expressions into nondeterministic ε-free automata with transitions. *J. Comput. Syst. Sci.*, 66(3) :451–472, 2003.

[GKS10] Orna Grumberg, Orna Kupferman, and Sarai Sheinvald. Variable automata over infinite alphabets. In *LATA* [LAT10], pages 561–572.

[Glu61] VM Glushkov. The abstract theory of automata. *Russian Mathematical Surveys*, 16(5) :1–53, 1961.

[HM00] Christian Hagenah and Anca Muscholl. Computing ε-free NFA from Regular Expressions in $O(n \log^2(n))$ Time. *RAIRO - Theoretical Informatics and Applications*, 34(4) :257–277, 2000.

[HMU06] John E. Hopcroft, Rajeev Motwani, and Jeffrey D. Ullman. *Introduction to Automata Theory, Languages, and Computation (3rd Edition)*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2006.

[Hod83] B.R. Hodgson. Décidabilité par automate fini. *Ann. Sci. Math. Quebec*, 7 :39–57, 1983.

[Hop71a] John Hopcroft. An $n \log n$ algorithm for minimizing states in a finite automaton. In Z. Kohavi and A. Paz, editors, *Theory of Machines and Computations*, pages 189–196. Academic Press, New York, 1971.

[Hop71b] John Hopcroft. An $n \log n$ algorithm for minimizing states in a finite automaton. In Z. Kohavi and A. Paz, editors, *Theory of Machines and Computations*, pages 189–196. Academic Press, New York, 1971.

[HSW01] Juraj Hromkovič, Sebastian Seibert, and Thomas Wilke. Translating regular expressions into small ε-free nondeterministic finite automata. *J. Comput. Syst. Sci.*, 62(4) :565–588, 2001.

[ICD09] *Database Theory - ICDT 2009, 12th International Conference, St. Petersburg, Russia, March 23-25, 2009, Proceedings*, volume 361 of *ACM International Conference Proceeding Series*. ACM, 2009.

[KF94] Michael Kaminski and Nissim Francez. Finite-Memory Automata. *Theor. Comput. Sci.*, 134(2) :329–363, 1994.

[KJHMP77] Donald E. Knuth, Jr. James H. Morris, and Vaughan R. Pratt. Fast Pattern Matching in Strings. *Society for Industrial and Applied Mathematics, Journal on Computing*, 6(2) :323–350, 1977.

[KL03] Dietrich Kuske and Markus Lohrey. Decidable theories of cayley-graphs. In *STACS* [STA03], pages 463–474.

[KL06] Dietrich Kuske and Markus Lohrey. Monadic chain logic over iterations and applications to pushdown systems. In *LICS* [LIC06], pages 91–100.

[Kle56] S. C. Kleene. Representation of events in nerve nets and finite automata. In Claude Shannon and John McCarthy, editors, *Automata Studies*, pages 3–41. Princeton University Press, Princeton, NJ, 1956.

[KLS10] Daniel Karch, Dennis Luxen, and Peter Sanders. Improved fast similarity search in dictionaries. In Chávez and Lonardi [CL10], pages 173–178.

[KN94] Bakhadyr Khoussainov and Anil Nerode. Automatic presentations of structures. In *LCC* [LCC95], pages 367–392.

[KR87] Richard M. Karp and Michael O. Rabin. Efficient randomized pattern-matching algorithms. *IBM J. Res. Dev.*, 31(2) :249–260, 1987.

[LAT10] *Language and Automata Theory and Applications, 4th International Conference, LATA 2010, Trier, Germany, May 24-28, 2010. Proceedings*, volume 6031 of *Lecture Notes in Computer Science*. Springer, 2010.

[LCC95] *Logical and Computational Complexity. Selected Papers. Logic and Computational Complexity, International Workshop LCC '94, Indianapolis, Indiana, USA, 13-16 October 1994*, volume 960 of *Lecture Notes in Computer Science*. Springer, 1995.

[Lec00] T. Lecroq. *Quelques aspects de l'algorithmique du texte*. Habilitation à Diriger des Recherches en Informatique de l'Université de Rouen, 2000.

[LIC00] *15th Annual IEEE Symposium on Logic in Computer Science, Santa Barbara, California, USA, June 26-29, 2000*. IEEE Computer Society, 2000.

[LIC06] *21th IEEE Symposium on Logic in Computer Science (LICS 2006), 12-15 August 2006, Seattle, WA, USA, Proceedings*. IEEE Computer Society, 2006.

[Lif03] Yury Lifshits. A lower bound on the size of ε-free NFA corresponding toa regular expression. *Inf. Process. Lett.*, 85(6) :293–299, 2003.

[LS08] Sylvain Lombardy and Jacques Sakarovitch. The universal automaton. In *Logic and Automata*, pages 457–504, 2008.

[LW93] Rüdiger Loos and Volker Weispfenning. Applying Linear Quantifier Elimination. *Comput. J.*, 36(5) :450–462, 1993.

[McC76] Edward M. McCreight. A Space-Economical Suffix Tree Construction Algorithm. *J. ACM*, 23(2) :262–272, 1976.

[MFC84] *Mathematical Foundations of Computer Science 1984, Praha, Czechoslovakia, September 3-7, 1984, Proceedings*, volume 176 of *Lecture Notes in Computer Science*. Springer, 1984.

[MFC01] *Mathematical Foundations of Computer Science 2001, 26th International Symposium, MFCS 2001 Marianske Lazne, Czech Republic, August 27-31, 2001, Proceedings*, volume 2136 of *Lecture Notes in Computer Science*. Springer, 2001.

[Moo56] Edward F. Moore. Gedanken-experiments on sequential machines. In Princeton University Press, editor, *Automata studies*, volume 34, pages 129–153. Princeton University Press, Princeton, N. J., 1956.

[Mos52] Andrzej Mostowski. On Direct Products of Theories. *J. Symb. Log.*, 17(1) :1–31, 1952.

[Myh57] J. Myhill. Finite automata and the representation of events. Technical Report WADD TR-57-624, Wright Patterson Air Force Base (Wright Air Development Division), Ohio, 1957.

[Ner58] A. Nerode. Linear automata transformation. In *Proceedings of the American Mathematical Society*, volume 9, pages 541–544, 1958.

[NSV01] Frank Neven, Thomas Schwentick, and Victor Vianu. Towards regular languages over infinite alphabets. In *MFCS* [MFC01], pages 560–572.

[OZ08] Faissal Ouardi and Djelloul Ziadi. Efficient weighted expressions conversion. *Theoretical Informatics and Applications*, 42(2) :285–307, 2008.

[PJ91] Mojzesz Presburger and Dale Jabcquette. On the completeness of a certain system of arithmetic of whole numbers in which addition occurs as the only operation. *History and Philosophy of Logic*, 12(2) :225–233, 1991.

[Pre30] Mojzesz Presburger. Über die Vollständigkeit eines gewissen Systems der Arithmetik ganzer Zahlen, in welchem die Addition als einzige Operation hervortritt. *Spra-wozdanie z I Kongresu matematyków krajów sl owiań skich, Warsawa 1929*, 395 :92–101, 1930.

[Rab69] M.O. Rabin. Decidability of second-order theories and automata on infinite trees. *Transactions of the American Mathematical Society*, 141 :1–35, 1969.

[Rab77] M. O. Rabin. Decidable theories. In J. Barwise, editor, *Handbook of Mathematical Logic*, pages 595–629. North-Holland, Amsterdam, 1977.

[Rev92] Dominique Revuz. Minimisation of Acyclic Deterministic Automata in Linear Time. *Theor. Comput. Sci.*, 92(1) :181–189, 1992.

[Rob58] R.M. Robinson. Restricted set-theoretical definitions in arithmetic. *Proc. Am. Math. Soc.*, 9 :238–242, 1958.

[RS59] M. O. Rabin and D. Scott. Finite automata and their decision problems. *IBM J. Res. Dev.*, 3(2) :114–125, April 1959.

[Rub08] Sasha Rubin. Automata presenting structures : A survey of the finite string case. *Bull. Symbolic Logic*, 14(2) :169–209, 2008.

[Sak03] J. Sakarovitch. *éléments de théorie des automates*. Les Classiques de l'informatique. Vuibert informatique, 2003.

[Sch06] Georg Schnitger. Regular expressions and nfas without *epsilon*-transitions. In *STACS* [sta06], pages 432–443.

[Sem84] Alexei L. Semenov. Decidability of monadic theories. In *MFCS* [MFC84], pages 162–175.

[SF94] Yael Shemesh and Nissim Francez. Finite-State Unification Automata and Relational Languages. *Inf. Comput.*, 114(2) :192–213, 1994.

[She75] Saharon Shelah. The monadic theory of order. *Annals of Mathematics*, 102 :379–419, 1975.

[STA92] *STACS 92, 9th Annual Symposium on Theoretical Aspects of Computer Science, Cachan, France, February 13-15, 1992, Proceedings*, volume 577 of *Lecture Notes in Computer Science*. Springer, 1992.

[STA03] *STACS 2003, 20th Annual Symposium on Theoretical Aspects of Computer Science, Berlin, Germany, February 27 - March 1, 2003, Proceedings*, volume 2607 of *Lecture Notes in Computer Science*. Springer, 2003.

[sta06] *STACS 2006, 23rd Annual Symposium on Theoretical Aspects of Computer Science, Marseille, France, February 23-25, 2006, Proceedings*, volume 3884 of *Lecture Notes in Computer Science*. Springer, 2006.

[STO04] *Proceedings of the 36th Annual ACM Symposium on Theory of Computing, Chicago, IL, USA, June 13-16, 2004*. ACM, 2004.

[Stu75] J. Stupp. The lattice-model is recursive in the original model. Technical report, Institute of Mathematics, The Hebrew University, Jerusalem, 1975.

[Tan10] Tony Tan. On pebble automata for data languages with decidable emptiness problem. *J. Comput. Syst. Sci.*, 76(8) :778–791, 2010.

[Tho97] Wolfgang Thomas. Handbook of formal languages, vol. 3. chapter Languages, Automata, and Logic, pages 389–455. Springer-Verlag New York, Inc., New York, NY, USA, 1997.

[Tsa11] Todor Tsankov. The additive group of the rationals does not have an automatic presentation. *J. Symb. Log.*, 76(4) :1341–1351, 2011.

[Ukk92] Esko Ukkonen. Constructing suffix trees on-line in linear time. In *Proceedings of the IFIP 12th World Computer Congress on Algorithms, Software, Architecture - Information Processing '92, Volume 1*, pages 484–492, Amsterdam, The Netherlands, The Netherlands, 1992. North-Holland Publishing Co.

[Ukk95] Esko Ukkonen. On-Line Construction of Suffix Trees. *Algorithmica*, 14(3) :249–260, 1995.

[Via09] Victor Vianu. Automatic verification of database-driven systems : a new frontier. In *ICDT* [ICD09], pages 1–13.

[Vil92] Roger Villemaire. Joining k- and l-recognizable sets of natural numbers. In *STACS* [STA92], pages 83–94.

[Wal02] Igor Walukiewicz. Monadic second-order logic on tree-like structures. *Theor. Comput. Sci.*, 275(1-2) :311–346, 2002.

[Wei73] Peter Weiner. Linear pattern matching algorithms. In *SWAT '73 : Proceedings of the 14th Annual Symposium on Switching and Automata Theory*, pages 1–11, Washington, DC, USA, 1973. IEEE Computer Society.

[Yu98] Sheng Yu. *Handbook of Formal Languages*, volume 1, chapter Regular Languages, pages 41–110. Springer, 1998.

INDEX

émondé, 15
équivalence, 1, 17, 58
équivalent, 4, 7, 13–15, 17, 53, 58, 78, 81, 83, 85, 125, 130
état, 2, 3, 5–7, 11, 13–22, 29, 30, 37, 47, 49, 50, 54, 55, 57–59, 77, 78, 82, 83, 95, 111–120, 122, 124, 125
étiqueté, 8, 15, 16, 22, 55, 59, 92, 114, 115, 120
étiqueter, 17
étiquette, 14, 16, 17, 22, 55, 97, 111, 117, 118, 122
étoile, 12

accès, 26, 27
accepte, 22, 58
accessible, 14–16, 125
acyclique, 15, 92
algébrique, 3
algorithme, 6, 8–11, 17, 19, 35, 53, 54, 57, 60, 62–66, 70, 71, 73, 78, 85, 86, 110, 118, 122, 125, 130
algorithmique, 8
alphabet, 1–6, 11–13, 16–19, 21, 25, 27, 29, 30, 33, 36–38, 43, 45, 47, 49, 51, 54, 56, 89–92, 94, 114, 120, 122, 124, 129
application, 1, 2, 6, 9, 19, 20, 22, 42, 43, 47, 48, 53, 79, 84, 86, 115, 130
arbre, 2, 7, 8, 53, 63–65, 71–75, 77, 82–86, 130
arithmétique, 19, 27, 74, 75, 78
arrêter, 2
associative, 12
asymptotiquement, 7, 53, 78, 83, 85, 130
automata, 3, 4
automate, 1–4, 6–11, 13–17, 19, 21, 22, 30–32, 40, 41, 44, 53–60, 63–65, 71, 73, 74, 77, 78, 82–86, 89, 92, 94, 95, 97, 99, 103, 110–125, 129, 130
automaticité, 1, 5, 6, 19, 21, 25, 27, 30, 33, 38, 39, 42, 43, 45, 46, 51, 129, 130
automatique, 2, 3, 9, 10, 19, 20, 22, 25, 28, 30, 33, 38–40, 42–44, 49
automatisé, 3

bijectif, 21
bijection, 21, 33, 47, 48, 50, 53, 63, 74, 75, 77, 84, 85
bijective, 19, 20
binaire, 4, 7, 9, 10, 13, 18, 20, 45, 46, 49, 53, 63, 65, 71, 73, 83, 85, 89, 95, 97, 99, 103, 112, 122, 130
bio-informatique, 7–9, 89
biologie, 8

calcul, 3, 15–17, 28, 57, 65–67, 70–73, 82, 83, 85, 86, 91, 97, 117–119, 124, 125, 130
cardinal, 12, 39, 49
cartésien, 4
chemin, 8, 14, 16, 22, 55, 58, 63, 65, 75–77, 85
circuit, 15
clôture, 3
clone, 5, 27–32, 38–40
clos, 12
codage, 19, 21, 28–30, 33, 35–40, 43, 44, 47–51, 97, 129
combinaison, 14
combinatoire, 7, 74
compilation, 10

complémentation, 3, 124
complet, 7, 15, 17, 53, 63, 65, 71, 73, 83, 120, 125
complexité, 2, 7, 53, 73, 94, 103
comportement, 2, 3
composition, 4, 6, 45–47, 49, 54, 55, 59, 60, 62–71, 86
concaténation, 11, 12, 14
constante, 13, 32, 114
correcteur, 9
couple, 2, 4, 15, 16, 89, 120, 122
critère, 46, 89, 94
croissante, 7, 46, 47, 79

décidabilité, 1–6, 19, 21, 25–27, 29, 30, 32, 33, 37, 38, 42, 45, 46, 49–51, 129, 130
décidable, 3–5, 19, 20, 22, 25, 29, 32, 37, 42, 45, 49, 51, 129
décider, 42
décompose, 45, 66, 86
définissabilité, 1
définissable, 4, 18–20, 29, 37, 47, 48, 50, 51
dénombrable, 4, 5, 25, 33, 38, 45, 47, 51, 129
déterminiser, 16, 110, 122
déterminisme, 16, 118
déterministe, 4, 6, 7, 13, 16, 17, 21, 53, 54, 56, 58, 62, 63, 66, 83, 86, 92, 110, 114, 122, 125
descendant, 16, 122
description, 10, 14, 15, 113
dictionnaire, 1, 8, 9, 89–92, 94, 95, 99, 103, 110
différence, 9, 38–40, 42, 72, 89
différentiel, 28, 30
differentia, 33, 37
discret, 2
distance, 9, 89–91
distinct, 4, 15, 17, 25, 26, 33–35, 38, 39, 55, 117, 129
division, 35, 47
document, 2, 8, 9
dynamique, 66, 71, 73, 115

ensemble, 1–5, 7, 8, 11–22, 25, 27, 30, 33, 38, 39, 47–49, 51, 53–55, 58, 59, 63–65, 73, 75–77, 83, 84, 89–92, 97, 111, 112, 114, 115, 118, 122, 125, 129
erreur, 8, 9, 99, 103
espace, 7–10, 66, 71, 85, 89, 91, 94, 111, 130
exclusive, 6, 42, 43
exhaustive, 7
expérimentalement, 9, 82, 89, 94, 95, 130
expérimentales, 1, 129
expérimentaux, 7, 53, 89, 94, 103
expressivité, 3, 4

factoriel, 65, 66
fermeture, 3, 4, 12, 118, 119
Fibonacci, 43
fonction, 4, 10, 13, 14, 16, 17, 19, 34, 35, 37, 38, 42, 50, 54, 55, 71, 73, 77–79, 82, 83, 111, 115–117, 119, 120, 122, 124, 125, 130
formalisme, 3–5, 18
formule, 4, 18–22, 26, 27, 30, 51, 55, 79, 85

Hamming, 9, 89–92, 94
heuristique, 7, 9, 53, 86, 130
homogène, 7, 16, 53–55, 86, 130

implémentation, 11, 122
index, 8, 9, 17
infini, 1–5, 8, 11, 13, 18, 19, 21, 25, 27, 30, 33, 38, 42, 43, 45, 47, 49, 51, 81, 129
injective, 19, 22
inondé, 7
insertion, 9
interprétable, 19–21, 29, 37, 47, 48, 50
interprétation, 5, 18, 20, 21, 29, 37, 47, 49, 50
intersection, 4, 12
inverse, 20, 21, 46, 47, 54, 55
isomophisme, 17, 125
itéré, 5
itératif, 65

lettre, 4, 6, 8, 9, 11–13, 15, 16, 18, 22, 25–28, 30, 31, 33–35, 38–41, 47, 49–51, 89–92, 95, 97, 114, 115, 119, 120, 122, 129
Levenstein, 9, 90
logarithmique, 8, 73
logique, 1–5, 8, 11, 16, 18–21, 25–27, 29, 30, 33, 37, 38, 46, 49, 51, 122, 129, 130

mémoire, 8–10, 14, 26, 71, 85, 110
mathématicien, 1
mathématique, 1, 11, 129

matrice, 10, 57, 59, 60, 62, 63, 65–70, 114
minimal, 7, 17, 34, 53, 56, 58, 65, 71–73, 75, 77, 78, 82–86, 92, 95, 125, 130
minimaux, 7, 53, 56, 65, 66, 71–73, 82, 83
minimisation, 1, 6, 7, 10, 17, 85, 86, 113, 125, 130
minimiser, 7, 68–70, 110, 125
monadique, 1, 4–6, 18–21, 25, 27, 29, 37, 45, 46, 49, 51, 129, 130
monotonement, 46, 47, 49
monotonie, 47
mot, 1–5, 7–9, 11–16, 18, 19, 21, 22, 25–41, 43–51, 58, 89–92, 94, 95, 97, 99, 103, 110, 114, 118, 119, 124, 129, 130
motif, 1, 7–9, 103
multiplication, 47
multiplicité, 6, 114

neutre, 11
non-finaux, 17, 124
normalisé, 15
noyau, 9

occurrences, 8, 9, 42, 65

Pascal, 7, 74–77, 84, 85
Pebble, 3
prédicat, 4–6, 18, 20, 21, 25–30, 32, 33, 38–40, 42, 48–51, 129
préfixe, 5, 11, 25–29, 31, 33, 37, 44, 48, 50, 51, 129
prétraitement, 8
premier, 1, 4–6, 18–21, 25, 27–34, 36–42, 44–46, 49–51, 53, 62, 63, 78, 79, 84, 85, 130
processus, 2, 3, 26, 27, 60, 63
programme, 2, 8, 71
projection, 4
protéine, 8
protocole, 10
puissance, 20

quadratique, 7
quadruplet, 45
quantifier, 18, 89
quasi-linéaire, 6, 53
quintuplet, 13

récursif, 62, 63, 65, 66
récursivement, 12, 90, 91
rédaction, 10, 111, 130
régulière, 8, 17, 19
réseau, 10
résiduel, 12
racine, 8, 63, 85
rationalité, 28, 33, 36
rationnel, 3, 6, 8, 12, 13, 17, 19–21, 28, 29, 36, 37, 53, 54, 56, 130
reconnaissabilité, 1
reconnaissable, 14, 19, 21, 22, 30–32, 40, 44
requête, 7–9

séquence, 5, 7, 8, 17, 18, 22, 27–29, 35, 37, 49, 58
schéma, 10
semi-structuré, 2
signature, 5, 18–20, 42
singleton, 20, 47, 48
spontané, 10, 113
standard, 15
structure, 1, 2, 4–6, 8–10, 13, 14, 18–22, 25, 27–30, 32, 33, 36–40, 42–51, 63, 73, 89, 103, 114, 122, 129, 130
substitution, 9
suffixe, 8, 11
supprime, 2, 16, 111
symbole, 6, 11, 13, 18–21, 27, 35, 36, 38, 54, 56, 89, 92, 94, 114
synchrone, 19, 21, 118
système, 2–4, 7, 9, 10, 26, 53–55, 57–60, 62–66, 71, 74, 77, 82, 83, 111, 130

tâche, 9, 10, 111, 130
tête, 21, 117
taille, 1, 6, 7, 13, 27, 28, 33, 54, 55, 59, 64–67, 70, 73, 77, 78, 89, 91, 92, 94, 95, 97, 99, 103, 114, 129
technique, 8, 92
temporelle, 2, 73
temporisé, 2
temps, 2, 6, 8, 9, 37, 53, 65, 66, 71, 73, 85, 89, 91, 92, 94, 99, 103, 130
terme, 14, 90
terminal, 22, 114
terminaux, 22, 114
ternaire, 28, 29, 37

texte, 1, 7–10, 89, 113, 116
textuelle, 7
Timed, 2
tolérance, 8
transformation, 10, 111, 112, 114
transition, 1, 6, 7, 10, 13–17, 21, 22, 53–55, 57–60, 62–65, 67, 71, 73, 74, 77, 78, 82–86, 92, 95, 97, 111–120, 122, 124, 130
transposition, 10
triangle, 7, 74–77, 84, 85
triangulaire, 7, 53, 55–60, 63, 66, 67, 84–86, 130
triplet, 21

unaire, 4, 13
universalité, 3, 4, 45, 46
universelle, 20, 21, 47, 48
uplet, 13, 18, 19, 21, 22, 45

vérification, 2, 3, 41
variable, 3, 18, 21, 114, 115, 118
variante, 4, 6, 8, 17
voisin, 9, 39–41, 89–92, 94, 95, 97, 99, 110, 130
volume, 10, 89, 130

zéro, 11, 14, 21, 28, 35, 36, 38, 49, 51, 74, 85

ABRÉVIATIONS

CFS	**C**ommon **F**ollow **S**ets
CTL	**C**omputation **T**ree **L**ogic
FO	**F**irst **O**rder Logic
FTP	**F**ile **T**ransfer **P**rotocol
LCP	**L**ongest **C**ommon **P**refix
LTL	**L**inear **T**emporal **L**ogic
MSO	**M**onadic **S**econd **O**rder Logic
SO	**S**econd **O**rder Logic
WMSO	**W**eak **M**onadic **S**econd **O**rder Logic
XML	e**X**tensible **M**arkup **L**anguage

الْحَمْدُ لِلّهِ الْمَلِكُ الْمَحْمُودُ الْوَدُودُ مُصَوِّرُ كُلِّ مَوْلُودٍ و عَالم الَا سَرَار و مُدْرِكهَا أَمَّا بَعْد. يَتَنَاوَلُ هَذَا الْعَمَلُ بِشَكْل أَسَاسِي نَظَرِيَّة التَّشْغِيلِ الْآلِي، الْمَنْطِق الرِّيَاضِي وَ تَطْبِيقَاتِهمَا. فِي الْجُزْء الْأَوَّل، تَمَّ اِسْتِخْدامُ آلِيَّاتٍ مُنْتَهِيَةٍ لِإِثْبَاتِ آلِيَّةِ الْعَدِيدِ مِنَ التَّرَاكِيبِ الْمَنْطِقِيَّةِ الْقَائِمَةِ عَلَى مَجْمُوعَةِ كَلِمَاتٍ تَامَّةٍ مَكْتُوبَةٍ فِي أَبْجَدِيَّةٍ لَانِهَائِيَّةِ الْحُروف. نَتَائِجُ هَذِهِ الْآلِيَّةِ تَسْمَحُ بِالْاِسْتِدْلَالِ عَلَى الْقُدْرَةِ عَلَى اقرار النَّظرِيَاتِ الْمَنْطِقِيَّةِ الْمُتَعَلِّقَةِ بِهذِه التَّراكِيب. تَمَّ ايضاً طرح نَتَائِج أُخْرَى ذَاتُ صِلَةٍ فِي عَدَم امكانية اقرار نَظريات أُخْرَى. فِي الْجُزْءِ الثَّانِي، تَمَّ تَعْمِيمُ مَفْهُوم الْمَجْمُوعَاتِ الْمُتَتَالِيَة الْمُشْتَرَكَة لِلتَّعْبِيرِ النَّمَطِي عَلَى الاليَاتِ الْمُتَجَانِسَةِ الْمُنْتَهِيَةِ. عَلَى أَسَاس هَذَا الْمَفْهُوم وَمَعَ فِئَة مُعَيَّنَةٍ مِنَ الإشجار الثُّنَائِيَّة، تَمَّ اِسْتِنْبَاط خوارزمية فَعَّالَة لِخَفض وَتَقْلِيل عَدَدِ التَّحَوُّلَاتِ للآلاتِ الْمُثَلَّثِيَّة. أَخيرا، تمَّ تَقْدِيم دارسَة تَجْرِيبِيَّة تَمَّ فِيهَا اِسْتِخْدامُ آلِيَّاتٍ مُتَخَصِّصَةٍ بِالْكَلِمَاتِ الْجُزْئِيَّةِ لِحَلِّ مُشكلةِ الْبَحْثِ عَنْ نَمط تَقْرِيبِي فِي الْقَوَامِيسِ.

خلاصه: مشکلات مطرح شده در این پژوهش به طور عمده در زمینه نظریه ماشینها (اتوماتها)، منطق ریاضی و کاربردهای آنان است. در نخستین قسمت، از ماشینهای (اتوماتهای) محدود برای نشان دادن مکانیزم هایی با ساختار منطقی بروی کلمات محدود که با الفبای نامحدود نوشته شده است، استفاده می شود. نتیجه حاصل از این مکانیزم اجازه وضع تعیین پذیری از تئوری های منطقی مرتبط را به ما می دهد. و همچنین دیگر نتایج تعیین پذیری / تعیین ناپذیری نیز نمایش داده شده است. در دومین قسمت، به مفهوم مجموعه دنباله مشترک یک عبارت منظم با اتوماتهای محدود همگن پرداخته شده است. بر اساس این مفهوم و یک کلاس خاص از درخت دودویی، یک الگوریتم کارآمد برای به کاهش / حداقل رساندن تعداد انتقالات اتوماتهای مثلثاتی توسعه یافته است. و در نهایت، به معرفی یک مطالعه تجربی برای استفاده از حفره های اتوماتها در جستجو تقریبی در واژه نامه ای از کلمه پرداخته است.

Résumé

Les problèmes traités et les résultats obtenus dans ce travail s'inscrivent essentiellement dans le domaine de la théorie des automates, la logique mathématique et leurs applications.

Dans un premier temps on utilise les automates finis pour démontrer l'automaticité de plusieurs structures logiques sur des mots finis écrits dans un alphabet infini dénombrable. Ceci nous permet de déduire la décidabilité des théories logiques associées à ces structures. On a considéré par exemple la structure $\mathfrak{S} = (\Sigma^*; \prec, \mathfrak{clone})$ où Σ^* désigne l'ensemble des mots finis sur l'alphabet infini dénombrable Σ, $\prec$ désigne la relation de préfixe et $\mathfrak{clone}$ désigne le prédicat qui est vrai pour un mot se terminant par deux lettres identiques. On a démontré l'automaticité de la structure $\mathfrak{S}$ et la décidabilité de sa théorie du premier ordre et de sa théorie monadique du second ordre. On a aussi considéré des extensions de la structure $\mathfrak{S}$ obtenues en ajoutant des prédicats comme $\sim$ qui est vrai pour deux mots de même longueur. Nous avons en particulier démontré la $\mathfrak{M}$-automaticité de la structure $\mathfrak{T} = (\Sigma^*; \prec, \mathfrak{clone}, \sim)$, d'où la décidabilité de sa théorie du premier ordre. On a par ailleurs étudié des structures qui comportent le prédicat $\mathfrak{diff}$ qui est vrai pour un mot dont les lettres sont toutes distinctes. En particulier on a démontré l'automaticité de la structure $\mathfrak{S}_4 = (\Sigma^*; \prec, \mathfrak{clone}, \mathfrak{diff})$ et la décidabilité de sa théorie du premier ordre et de sa théorie monadique du second ordre. On a également obtenu, par interprétation logique, des résultats de décidabilité et des résultats d'indécidabilité pour plusieurs variantes des structures $\mathfrak{T}$ et $\mathfrak{S}_4$, ainsi que pour des familles de structures appelées *structure d'applications exclusives* et *structure de décomposition*.

Dans un deuxième temps on s'est intéressé au problème de la réduction du nombre de transitions dans les automates finis. On a commencé par étendre le concept de *Common Follow Sets* d'une expression régulière aux automates finis homogènes. On a montré comment établir une liaison assez directe entre des systèmes de CFS spécifiques et les arbres binaires complets. Ce lien est prouvé en utilisant un objet combinatoire appelé *triangle d'Ératosthène - Pascal*. Cette correspondance permet de transformer la valeur qui nous intéresse (le nombre de transitions) en une valeur assez naturelle associée aux arbres (le poids d'un arbre). En effet, construire un automate ayant un minimum de transitions revient à trouver un arbre de poids minimal. On a montré, d'une part, que ce nombre de transitions est asymptotiquement équivalent à $n(\log_2 n)^2$ (la borne inférieure). D'autre part, les tests expérimentaux montrent que pour les petites valeurs de n, les automates minimaux en nombre de transitions coïncident (en nombre et en taille) avec ceux obtenus par notre construction. Cela nous mène à suggérer que notre réduction est finalement une minimisation pour les automates triangulaires.

Dans un dernier temps on a présenté une étude expérimentale concernant l'application des automates à trous dans le domaine de la recherche approchée de motif dans les dictionnaires de mots. Contrairement aux complexités théoriques, temps de recherche et espace de stockage exponentiels, nos expérimentations montrent la linéarité de l'automate à trous.

Abstract

This work deals mainly with automata theory, mathematical logic and their applications.

In the first part, we use finite automata to prove the automaticity of several logical structures over finite words written in a countable infinite alphabet. These structures involve predicates like $\prec$, $\mathfrak{clone}$ and $\mathfrak{diff}$, where $x \prec y$ holds if x is a strict prefix of y, $\mathfrak{clone}(x)$ holds when the two last letters of x are equal, and $\mathfrak{diff}(x)$ holds when all letters of x are pairwise distinct. The automaticity results allow to deduce the decidability of logical theories associated with these structures. Other related decidability/undecidability results are obtained by logical interpretation.

In the second part, we generalize the concept of Common Follow Sets of a regular expression to homogeneous finite automata. Based on this concept and a particular class of binary trees, we devise an efficient algorithm to reduce/minimize the number of transitions of triangular automata. On the one hand, we prove that the produced reduced automaton is asymptotically minimal, in the sense that for an automaton with n states, the number of transitions in the reduced automaton is equivalent to $n(\log_2 n)^2$, which corresponds at the same time to the upper and the lower known bounds. On the other hand, experiments reveal that for small values of n, all minimal automata are exactly those obtained by our reduction, which lead us to conjecture that our construction is not only a reduction but a minimization.

In the last part, we present an experimental study on the use of special automata on partial words for the approximate pattern matching problem in dictionaries. Despite exponential theoretical time and space upper bounds, our experiments show that, in many practical cases, these automata have a linear size and allow a linear search time.