UNIVERSITÉ DU QUÉBEC À CHICOUTIMI

MÉMOIRE PRÉSENTÉ À
L'UNIVERSITÉ DU QUÉBEC À MONTRÉAL
COMME EXIGENCE PARTIELLE
DE LA MAÎTRISE EN INFORMATIQUE
OFFERTE À
L'UNIVERSITÉ DU QUÉBEC À CHICOUTIMI
EN VERTU D'UN PROTOCOLE D'ENTENTE
AVEC L'UNIVERSITÉ DU QUÉBEC À MONTRÉAL

PAR
ERIC LAVOIE
B. SC. A.

ALGORITHME DE CALCUL DU MONOÏDE DÉRIVÉ D'UNE BOUCLE FINIE

AOÛT 2006

Bibliothèque Paul-Émile-Boulet

UQAC Université du Québec à Chicoutimi

Mise en garde/Advice

Afin de rendre accessible au plus grand nombre le résultat des travaux de recherche menés par ses étudiants gradués et dans l'esprit des règles qui régissent le dépôt et la diffusion des mémoires et thèses produits dans cette Institution, **l'Université du Québec à Chicoutimi (UQAC)** est fière de rendre accessible une version complète et gratuite de cette œuvre.

Motivated by a desire to make the results of its graduate students' research accessible to all, and in accordance with the rules governing the acceptation and diffusion of dissertations and theses in this Institution, the **Université du Québec à Chicoutimi (UQAC)** is proud to make a complete version of this work available at no cost to the reader.

L'auteur conserve néanmoins la propriété du droit d'auteur qui protège ce mémoire ou cette thèse. Ni le mémoire ou la thèse ni des extraits substantiels de ceux-ci ne peuvent être imprimés ou autrement reproduits sans son autorisation.

The author retains ownership of the copyright of this dissertation or thesis. Neither the dissertation or thesis, nor substantial extracts from it, may be printed or otherwise reproduced without the author's permission.

Résumé

Notre recherche se situe dans le cadre de la théorie algébrique des langages. Il est bien connu que les semigroupes finis peuvent être vus comme des automates finis. Ainsi, ils permettent de reconnaître exactement les langages réguliers. D'une façon similaire, les groupoïdes finis sont associés aux automates à pile et aux langages hors-contextes.

Il a été démontré qu'un type particulier de groupoïdes, les boucles finies, ne reconnaissent que des langages réguliers. Mais les groupoïdes et les boucles sont des structures algébriques non associatives, ce qui rend leur étude très complexe. Cependant, il a été démontré que l'on peut associer à chaque boucle finie un unique monoïde fini nommé le monoïde dérivé de la boucle. Ce monoïde préserve plusieurs propriétés de la boucle. En particulier, il reconnaît l'ensemble des langages reconnus par la boucle concernée, ce qui permet d'utiliser la richesse et l'élégance de la théorie des semigroupes dans l'étude des boucles finies.

Dans ce travail, nous démontrerons que le monoïde dérivé d'une boucle finie est calculable. Par la suite, nous présenterons différentes versions d'un algorithme ayant pour objectif de construire le monoïde dérivé d'une boucle finie. Finalement, nous exposerons et analyserons brièvement quelques résultats calculés sur les boucles d'ordre 8 et moins à l'aide de ces différentes versions de l'algorithme.

Remerciements

Je tiens tout d'abord à exprimer mes sincères remerciements à François Lemieux, mon directeur de recherche, pour son implication à la réalisation de ce travail de recherche et pour nos nombreuses discussions sur la vie. Sa grande disponibilité, sa très grande franchise et son enthousiasme débordant m'ont permis de terminer ce projet avec succès.

Je veux souligner l'apport du personnel administratif et du corps professoral du Département d'informatique et de mathématique (DIM) de l'UQAC pour leur aide et leurs enseignements. Je dois également souligner le support financier du Conseil de Recherche en Sciences Naturelles et en Génie du Canada (CRSNG).

Je veux souligner l'appui quotidien de tous les étudiants qui ont travaillé au Groupe de Recherche en Informatique (GRI) au cours des quatre années que j'y ai passées. Plus spécifiquement, j'adresse un merci particulièrement désorganisé à Pierre. Je veux également remercier Jérôme, mon ami et colocataire de bureau. Merci pour les nombreux conseils sur l'écriture de ce mémoire et merci pour l'humour vaudevillesque dont tu sais faire preuve.

Je veux remercier tous les gens qui s'impliquent et se sont impliqués au sein du Mouvement des Associations Générales Étudiantes de l'UQAC (MAGE-UQAC). Plus

particulièrement, je veux remercier Fred, Jeff, Martin et Valérie pour leur sincère amitié et pour l'ensemble des connaissances qu'ils m'ont transmises.

Sur un plan plus personnel, je veux remercier mes amis pour m'avoir permis de souffler pendant les moments plus difficiles. Je veux également remercier mes parents, Chantal et Gilles, et ma soeur Johanne. Merci de m'avoir encouragé tout au long de mes études, votre soutient fut inestimable. Finalement, je veux remercier chaleureusement ma copine Carole. Merci d'être là pour partager mes joies et mes peines, merci de me supporter dans mes décisions et de m'encourager dans mes projets, merci d'être ce que tu es.

Table des matières

Liste des Figures

Liste des Tableaux

Liste des Algorithmes

Liste des Fonctions

Chapitre 1

Introduction

La progression fulgurante qu'a connue l'informatique depuis les années cinquante n'aurait jamais été possible sans le développement de l'informatique théorique. Les concepts fondamentaux qu'elle a établis permettent aujourd'hui aux ordinateurs d'effectuer des tâches qui ont changé notre mode de vie. Une des branches de l'informatique théorique, appelée la théorie des automates, a d'ailleurs contribué à la création d'outils tels que les compilateurs qui sont utilisés dans la programmation des langages de hauts niveaux. Cette théorie possède également de nombreuses applications dans d'autres domaines que l'informatique, notamment en bio-informatique, en linguistique et en mathématiques.

La théorie des automates étudie en fait les langages formels et les modèles théoriques de calculs qui servent à reconnaître ces langages. L'automate fini est un des modèles de calcul qui a été largement étudié (voir [19, 32, 16]). C'est ainsi que nous savons que les automates finis reconnaissent exactement les langages rationnels (langages réguliers) (voir [20]). Par contre, il existe des questions relatives aux langages réguliers auxquelles les chercheurs sont incapables de répondre avec la seule utilisation des automates finis. Par exemple, ils sont incapables, en utilisant seulement ces outils, de déterminer si un langage régulier est sans étoile (voir 2.1). L'ajout d'outils plus raffinés à la théorie des automates est donc devenu une nécessité.

C'est ainsi que M.P. Schützenberger introduisit l'utilisation des *monoïdes finis* (ensemble fini muni d'une opération binaire associative fermée sur l'ensemble et contenant un élément identité, voir Section 2.3.1) en théorie des automates (voir [29]). Il considère alors un monoïde fini comme une machine pouvant reconnaître des langages. À partir d'un automate fini et d'un langage L reconnu par celui-ci, il calcul un monoïde que l'on nomme le *monoïde syntactique* de L. L'intérêt de ce monoïde réside dans les deux faits suivants : le monoïde syntactique de L est unique et il est le plus petit monoïde reconnaissant le langage L (voir Section 2.3.2). Ainsi, l'utilisation du monoïde syntactique de L permet de déterminer si un langage régulier est sans étoile (voir Section 2.3.2), démontrant par le fait même que les propriétés combinatoires des langages sont intimement liées à leurs propriétés algébriques (voir [30]).

Comme nous le verrons, les monoïdes finis reconnaissent exactement les mêmes langages que les automates finis, i.e. les langages réguliers. Par la suite, Eilenberg généralisa les travaux de Schützenberger et établit dans [16] les bases d'une théorie qui allait conduire à la classification des monoïdes finis selon les langages qu'ils reconnaissent. D'autres travaux ont également démontré que l'utilisation du monoïde comme modèle de calcul ne se limite pas à l'étude des langages réguliers (voir [22, 4]).

Devant ces nombreux résultats obtenus sur les monoïdes, certains chercheurs se sont demandés si l'utilisation d'une structure algébrique non associative, les *groupoïdes finis* (ensemble fini muni d'une opération binaire fermée sur l'ensemble, voir Section 2.4.1), pouvait constituer un modèle de calcul plus puissant. En effet, nous savons maintenant que les groupoïdes finis reconnaissent exactement les mêmes langages que les automates à pile, i.e. les langages hors-contextes ([25], voir aussi [5]). Pour pouvoir classifier les langages hors-contextes, il est intéressant de mieux comprendre le fonctionnement de la non-associativité sur des groupoïdes moins complexes et plus structurés. C'est ainsi

que certains chercheurs ont commencé à étudier les *boucles finies* (ensemble fini muni d'une opération binaire fermée sur l'ensemble telle que la table de multiplication forme un carré latin, voir Section 2.4.2) et les langages qu'elles reconnaissent (voir [12, 21]). Un premier résultat important est obtenu dans [7], où il est démontré que les boucles finies reconnaissent exactement une classe de langages réguliers bien connue, soit celle des langages réguliers ouverts.

Tout comme cela a été fait pour les monoïde finis, il serait également intéressant de classifier les boucles finies selon la complexité des langages qu'elles reconnaissent. Certains travaux ont d'ailleurs été entrepris en ce sens (voir [8]). Un des outils pouvant faciliter cette classification est le monoïde dérivé d'un groupoïde (voir Chapitre 3), un concept initialement introduit dans [6]. Une propriété importante du monoïde dérivé d'un groupoïde G est qu'il reconnaît tous les langages reconnus par G. En utilisant cet outil sur les boucles finis, il devient possible d'utiliser la richesse de la théorie des monoïdes dans l'étude des boucles.

Ce travail possède deux objectifs principaux. Le premier est de démontrer que le monoïde dérivé d'une boucle finie est calculable. Le deuxième est de trouver un algorithme efficace permettant de calculer le monoïde dérivé des boucles contenant peu d'éléments (8 et moins). Notre second objectif comprend seulement les boucles d'ordre 8 et moins, car la taille du monoïde dérivé peut être doublement exponentielle par rapport à l'ordre de la boucle (voir Chapitre 5). De plus, le nombre de boucles à traiter (classes d'isomorphismes) pour un ordre donné croît également de façon exponentiel. Ainsi, il existe 23 746 classes d'isomorphismes parmi les boucles d'ordre 7, mais plus de 106 000 000 pour les boucles d'ordre 8 et plus de 9 365 milliards pour les boucles d'ordre 9 (voir [24]). Finalement, il est important de noter que, comme nous le verrons au Chapitre 4, le deuxième objectif a seulement été partiellement atteint.

L'atteinte de ces objectifs nécessite d'abord l'introduction de nombreuses notions sur les structures algébriques. Ainsi, nous verrons au Chapitre 2 quelques brèves définitions sur les relations binaires. Par la suite, nous donnerons la définition formelle d'un monoïde et nous ferons une brève étude de leurs propriétés. Puis nous définirons la reconnaissance d'un langage par un monoïde et nous étudierons les principaux résultats s'y rattachant. La dernière section de ce chapitre est consacrée aux définitions relatives aux groupoïdes et aux boucles, ainsi qu'à la définition de la reconnaissance d'un langage par ces structures algébriques non associatives.

Le Chapitre 3 est consacré à l'étude du monoïde dérivé. Nous introduirons d'abord le concept de monoïde dérivé d'un groupoïde et nous présenterons plusieurs propriétés de celui-ci. Puis, nous démontrerons que le monoïde dérivé d'une boucle finie est fini et nous démontrerons qu'il est possible de le calculer en prouvant qu'il n'est nécessaire que d'effectuer une quantité finie de calcul pour l'obtenir. Nous verrons également qu'il est difficile d'évaluer précisément sa taille, mais nous donnerons tout de même une borne supérieure. Le reste de ce chapitre est consacré à la présentation de deux versions d'un algorithme permettant de calculer le monoïde dérivé d'une boucle finie. L'analyse des temps d'exécutions de ces deux versions de l'algorithme nous apprendra qu'elles sont inefficaces.

Ainsi, le Chapitre 4 se concentre sur les solutions possibles pour améliorer l'algorithme présenté au Chapitre 3. Nous présenterons trois solutions servant à réduire la quantité de calcul à effectuer. Ces nouvelles versions sont plus efficaces que les deux premières, mais dans la plupart des cas, nous devons pour cela sacrifier la certitude de l'obtention du monoïde dérivé. Nous verrons que le monoïde obtenu par ces versions n'est tout de même pas dénué d'intérêt car il est en pratique unique et il possède plusieurs propriétés connexes au monoïde dérivé. Il existe d'ailleurs plusieurs facteurs

permettant de croire que ce monoïde correspond toujours au monoïde dérivé de la boucle concernée.

Le Chapitre 5 se penche sur l'analyse des résultats. La première section présente les monoïdes dérivés calculés. La deuxième contient les monoïdes "pseudo-dérivés", c'est-à-dire les monoïdes obtenus en pratique par les versions de l'algorithme présentées au Chapitre 4 et tel que nous ne sommes pas en mesure de démontrer qu'ils correspondent au monoïde dérivé. Dans chacun des cas, nous fournissons plusieurs propriétés sur les boucles et les monoïdes présentés. Finalement, la conclusion nous permettra de poser plusieurs questions que les travaux effectués au cours de ce mémoire nous ont apportées.

CHAPITRE 2

PRÉLIMINAIRES

2.1 Introduction

Nous considérons dans ce chapitre que le lecteur est familier avec les notions de bases en théorie des langages (langages réguliers, expressions régulières, automates finis, langages hors-contexte, automates à pile). Sinon, les définitions nécessaires peuvent être trouvées dans [32] ou dans [19]. Lorsqu'on parle de langages formels, une question naturelle est de savoir si, étant donné un langage L et un mot w, le mot w appartient ou non au langage L. Si un algorithme M peut répondre à cette question correctement, on dit que M *reconnaît* le langage L. Il existe plusieurs modèles de calcul pouvant être utilisés comme machine pour reconnaître des langages, selon les particularités combinatoires de ceux-ci (voir [31, 23, 3, 33, 14]). Par exemple, un de ces modèles de calcul très répandus est l'automate fini (voir exemple à la Figure 2.1) et nous savons que les automates finis reconnaissent exactement les langages rationnels (langages réguliers) (voir [20]).

Malgré la richesse de la théorie des automates, certaines questions sont longtemps demeurées sans réponse avec la seule utilisation de ces outils. Par exemple, on dit qu'un langage L est sans étoile s'il existe une expression régulière définissant L et n'utilisant pas l'opérateur * (opération de Kleene), mais pouvant utiliser l'opérateur unaire de complémentation ($\neg$). Durant de nombreuses années, il a été impossible de répondre à

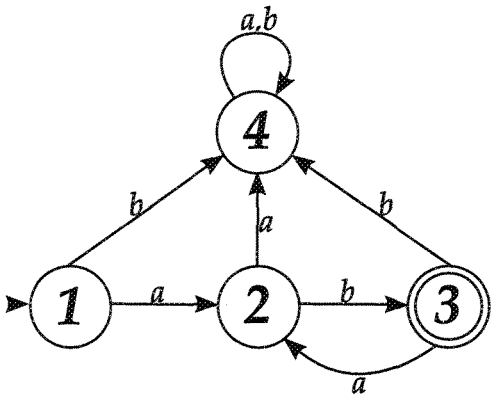

Figure 2.1: Automate fini $\mathcal{B} = \{Q, \Sigma, \delta\}$, où $Q = \{1, 2, 3, 4\}$ est l'ensemble fini des états, $\Sigma = \{a, b\}$ est l'alphabet fini et δ est l'application de transition définit graphiquement. En utilisant l'état 1 comme état de départ et l'ensemble d'états $F = \{3\}$ comme états finaux, l'automate $\mathcal{B}$ reconnaît le langage $L_{(ab)} = ab(ab)^* \subset \Sigma^*$, contenant tous les mots non vides composés uniquement d'une répétition du mot ab.

la question suivante : soit L un langage régulier, est-ce que L est un langage régulier sans étoile. En fait, on ne savait même pas si la réponse à cette question était décidable. Pour certains langages, comme pour les langages exprimés par une expression régulière n'utilisant pas l'étoile ou les langages finis, il est facile de répondre à cette question. Il est par contre plus difficile de savoir si le langage régulier $L_{(ab)} = ab(ab)^*$, reconnu par l'automate fini présenté à la Figure 2.1, est sans étoile.

Il fût nécessaire d'attendre l'introduction d'un nouveau modèle de calcul, le semigroupe fini (voir Section 2.3), pour être capable de déterminer si un langage régulier est sans étoile. En effet, M.P. Schützenberger démontre dans [30] que les langages sans étoile possèdent une caractéristique algébrique commune qu'il est possible de vérifier. Avant de pouvoir énoncer exactement ce théorème et déterminer si le langage $L_{(ab)}$ est sans étoile, nous devons familiariser le lecteur avec quelques notions de bases sur les structures algébriques. Ainsi, à la Section 2.2 nous allons énoncer quelques définitions sur les relations binaires. Puis, à la Section 2.3 nous allons définir la notion de semigroupe et énoncer quelques-unes de ses propriétés. Les structures algébriques non associatives,

présentées à la Section 2.4, nous permettrons finalement d'établir clairement l'objectif de ce mémoire.

2.2 Relations binaires

Soient A et B, deux ensembles. Une relation $\mathcal{R} : A \to B$ est un sous-ensemble du produit cartésien $A \times B$. Si $a \in A$ et $b \in B$, alors $(a, b) \in \mathcal{R}$ est dénoté par $a\mathcal{R}b$. Si $A = B$, on dit que $\mathcal{R}$ est une relation sur A. Soient les relations binaires $\mathcal{R} : A \to A$ et $\varphi : A \to B$, voici quelques propriétés applicables à ces relations binaires.

$\mathcal{R}$ est *réflexive* $\Leftrightarrow$ $\forall a \in A,\ a\mathcal{R}a.$

$\mathcal{R}$ est *symétrique* $\Leftrightarrow$ $\forall a_1, a_2 \in A$, si $a_1\mathcal{R}a_2$, alors $a_2\mathcal{R}a_1$.

$\mathcal{R}$ est *transitive* $\Leftrightarrow$ $\forall a_1, a_2, a_3 \in A$, si $a_1\mathcal{R}a_2$ et $a_2\mathcal{R}a_3$, alors $a_1\mathcal{R}a_3$.

$\mathcal{R}$ est une relation d'*équivalence* $\Leftrightarrow$ $\mathcal{R}$ est *réflexive*, *symétrique* et *transitive*.

φ est *injective* $\Leftrightarrow$ $\forall a_1, a_2 \in A$, s'il existe $b \in B$ tel que $(a_1, b) \in \varphi$ et tel que $(a_2, b) \in \varphi$, alors $a_1 = a_2$.

φ est *surjective* $\Leftrightarrow$ $\forall b \in B$, $\exists a \in A$ tel que $(a, b) \in \varphi$.

φ est une *fonction* $\Leftrightarrow$ $\forall a \in A$, il existe au plus un élément $b \in B$ tel que $(a, b) \in \varphi$. On dénote alors cet élément $\varphi(a)$ et on dit que b est l'image de a dans φ.

φ est une *application* $\Leftrightarrow$ $\forall a \in A$, il existe exactement un élément $b \in B$ tel que $\varphi(a) = b$.

$$\varphi \text{ est } \textit{bijective} \quad \Leftrightarrow \quad \varphi \text{ est une } \textit{application injective} \text{ et } \textit{surjective}.$$

Cela implique nécessairement que $|A| = |B|$.

Soit S, un ensemble. On nomme *loi de composition interne* une application $\cdot$ de $(S \times S)$ dans S. Voici quelques propriétés possibles pour l'application $\cdot$.

$$\begin{aligned}
\cdot \text{ est } \textit{associative} &\quad \Leftrightarrow \quad \forall x, y, z \in S,\ ((x \cdot y) \cdot z) = (x \cdot (y \cdot z)) \\
\cdot \text{ est } \textit{commutative} &\quad \Leftrightarrow \quad \forall x, y \in S,\ (x \cdot y) = (y \cdot x) \\
\textit{Annulation à gauche} &\quad \Leftrightarrow \quad \forall x, y, z \in S,\ (x \cdot y) = (x \cdot z) \text{ implique } y = z \\
\textit{Annulation à droite} &\quad \Leftrightarrow \quad \forall x, y, z \in S,\ (y \cdot x) = (z \cdot x) \text{ implique } y = z
\end{aligned}$$

Soit A, un ensemble fini et soient $\mathcal{F} : A \to A$ et $\mathcal{G} : A \to A$, deux applications sur A. L'opération de composition des applications $\circ$ est l'application $(\mathcal{F} \circ \mathcal{G}) : A \to A$ définit par $(\mathcal{F} \circ \mathcal{G})(a) = \mathcal{F}(\mathcal{G}(a))$, $\forall a \in A$.

2.3 Semigroupes et monoïdes

Cette section est divisée en deux parties. Premièrement, nous définissons ce qu'est un semigroupe et nous énonçons plusieurs propriétés des semigroupes. Quelques exemples concrets sont également présentés pour favoriser la compréhension du lecteur. Deuxièmement, nous allons discuter de l'utilisation du semigroupe fini en théorie des automates et de la notion de reconnaissance d'un langage par un semigroupe. Notons que les résultats présentés dans cette section sont des résultats classiques de la littérature sur les semigroupes finis. Aussi, certaines propositions seront énoncées sans que les preuves ne soient détaillées (une référence sera par contre fournie).

2.3.1 Définitions et propriétés

Un semigroupe est un couple $(S, \cdot)$ où S est un ensemble et $\cdot$ est une loi de composition interne associative sur S. Notons que lorsqu'il n'y a aucune ambiguïté possible, un semigroupe $(S, \cdot)$ est simplement dénoté S. On dit qu'un semigroupe $(S, \cdot)$ est fini si l'ensemble S est fini. Un élément e de S est appelé *identité* si pour tout $x \in S$, $e \cdot x = x \cdot e = x$. On nomme *monoïde* un semigroupe contenant un élément identité. Remarquons qu'un monoïde ne peut posséder deux éléments identités car si e et f sont deux éléments identités du monoïde tel que $e \neq f$, il est impossible de définir le résultat de la multiplication de e par f sans obtenir une contradiction (voir [27]). Si la loi de composition interne d'un monoïde $(M, \cdot)$ supporte l'annulation à gauche et à droite, $(M, \cdot)$ est alors appelé un *groupe*.

Exemple 2.3.1. *Une des façons pratique de représenter un semigroupe fini est l'utilisation de sa table de Cayley, une généralisation du concept de table de multiplication. Les propriétés d'annulation à gauche et à droite sont facilement vérifiables à l'aide de la table de Cayley d'une structure algébrique. En effet, l'annulation à gauche signifie qu'il ne peut y avoir deux éléments identiques sur une même ligne alors que l'annulation à droite signifie qu'il ne peut y avoir deux éléments identiques sur une même colonne. La Figure 2.2 présente la table de Cayley du monoïde* $(M_1, \wedge)$, *où* $M_1 = \{0, 1\}$, *représentant l'opération booléenne de conjonction. Dans ce monoïde, on remarque que l'élément 1 est l'identité puisque* $0 \wedge 1 = 1 \wedge 0 = 0$ *et* $1 \wedge 1 = 1$. *On remarque également que* M_1 *n'est pas un groupe puisqu'il ne supporte ni l'annulation à gauche, ni l'annulation à droite.*

Dans certains domaines de la théorie des semigroupes, la distinction entre un semigroupe et un monoïde est très importante. Dans notre cas, nous nous concentrerons

$\wedge$	0	1
0	0	0
1	0	1

Figure 2.2: Table de Cayley du monoïde $(M_1, \wedge)$, où $M_1 = \{0, 1\}$.

plutôt sur les monoïdes finis. Ainsi, bien que la plupart des définitions qui suivent ont leur équivalent dans la théorie des semigroupes en général, nous les exprimerons par rapport aux monoïdes.

Définition 2.3.2 (Morphisme de monoïdes). *Soient $(M, \cdot)$ et $(N, \bullet)$, deux monoïdes où $1_M \in M$ et $1_N \in N$ sont respectivement l'identité de M et N. L'application $\varphi : M \to N$ est un* morphisme *de monoïdes si et seulement si $\forall m_1, m_2 \in M$, $\varphi(m_1 \cdot m_2) = \varphi(m_1) \bullet \varphi(m_2)$ et si $\varphi(1_M) = 1_N$. Si φ est également une bijection, on dira alors que φ est un* isomorphisme *de M dans N.*

Pour le reste de cette section, $(M, \cdot)$ représente un monoïde fini. Soit $(N, \bullet)$, un monoïde fini. Il peut exister de nombreuses relations entre M et N. Nous allons décrire celles qui sont les plus importantes dans le cadre de notre travail. Nous présenterons également quelques exemples concrets de monoïdes pour appuyer les définitions formelles.

Si N est un sous-ensemble de M contenant l'identité, alors $(N, \cdot)$ est un *sous-monoïde* de $(M, \cdot)$ si et seulement si $\forall n_1, n_2 \in N$, $(n_1 \cdot n_2) \in N$. Si pour tout $a \in M$, il existe $n \in \mathbb{N}^+$ tel que $a^n = a^{n+1}$, alors on dit que le monoïde M est *apériodique*. Il est démontré dans [27] que cela équivaut aux monoïdes ne contenant aucun sous-monoïde non trivial, c'est-à-dire contenant plus d'un élément, qui est un groupe. Soit $P \subseteq M$, alors P *génère* M si et seulement si pour tout $m \in M$, il existe $p_1, p_2, \ldots p_k \in P$, tel que $p_1 \cdot p_2 \cdots \cdot p_k = m$.

$\circ$	**1**	a	b	**aa**	**ab**	**ba**
1	**1**	a	b	**aa**	**ab**	**ba**
a	a	aa	ab	aa	aa	a
b	b	ba	aa	aa	b	aa
aa	**aa**	aa	aa	**aa**	**aa**	**aa**
ab	**ab**	a	aa	**aa**	**ab**	**aa**
ba	**ba**	aa	b	**aa**	**aa**	**ba**

Figure 2.3: Table de Cayley du monoïde $(M_2, \circ)$, où $M_2 = \{1,a,b,aa,ab,ba\}$.

Exemple 2.3.3. *La Figure 2.3 présente la table de Cayley du monoïde* $(M_2, \circ)$. *On remarque que l'ensemble* $M_2' = \{1, aa, ab, ba\}$, *muni de l'opération* $\circ$, *forme un sous-monoïde du monoïde* M_2. *En effet, l'opération* $\circ$ *est fermée pour l'ensemble* M_2', *c'est-à-dire qu'il n'est possible que d'obtenir des éléments de l'ensemble* M_2' *si l'on multiplie entre eux deux éléments de cet ensemble. Par contre, le sous-monoïde* M_2' *n'est pas un groupe et il est facile de vérifier que le monoïde* M_2 *est apériodique, puisqu'il suffit de vérifier que pour tout* $a \in M_2$, *il existe* n *tel que* $a^n = a^{n+1}$. *L'ensemble* $P_2 = \{1, a, b\}$, *sous-ensemble de* M_2, *est un ensemble générateur pour le monoïde* M_2, *puisque* $aa = a \circ a$, $ab = a \circ b$, $ba = b \circ a$ *et que* $a, b \in P_2$.

S'il existe un morphisme surjectif $\tau : M \to N$, alors on dira que N est un *quotient* de M. Finalement, on dira que N *divise* M (dénoté $N \prec M$), si N est quotient d'un sous-monoïde de M.

Exemple 2.3.4. *La Figure 2.4 présente le monoïde* $(M_3, \diamond)$. *On définit l'application* $\tau : M_3 \to M_1$, *où* M_1 *est le monoïde définit à la Figure 2.2, par*

$$\tau(a) = 0 \ , \ \tau(b) = 0$$
$$\tau(c) = 1 \ , \ \tau(1) = 1$$

Il est facile de vérifier que τ *est un morphisme de* M_3 *vers* M_1. *De plus,* τ *est*

$\diamond$	a	b	c	1
a	a	a	a	a
b	a	b	a	b
c	a	a	c	c
1	a	b	c	1

Figure 2.4: Table de Cayley du monoïde $(M_3, \diamond)$, où $M_3 = \{1, a, b, c\}$.

un morphisme surjectif puisque tous les éléments de M_1 sont l'image d'au moins un éléments de M_3, ce qui signifie que M_1 est un monoïde quotient de M_3.

Avant de pouvoir discuter du lien existant entre les automates finis et les monoïdes finis et de définir la reconnaissance d'un langage par un monoïde, nous devons introduire le concept de monoïde de transformation. Nous introduirons également les concepts de congruences et de monoïde libre qui nous serons utiles tout au long de ce travail.

Définition 2.3.5 (Monoïde de transformation). *Soit E, un ensemble, on dénote par $T(E)$ l'ensemble des applications de la forme $\mathcal{R} : E \to E$. Remarquons que l'opérateur de composition des applications $(\circ)$ est une opération fermée et associative sur les éléments de $T(E)$, donc $(T(E), \circ)$ forme un monoïde. On nomme* monoïde de transformation *sur E tout sous-monoïde de $(T(E), \circ)$.*

Remarquons que si l'ensemble E est fini, alors tous monoïde de transformation sur E sera également fini.

Définition 2.3.6 (Congruence). *Une congruence sur un monoïde M est une relation d'équivalence $\sim$ qui est compatible à gauche et à droite avec l'opération du monoïde, c'est-à-dire tel que pour tout $m_1, m_2, m_3 \in M$, $m_1 \sim m_2$ implique que $(m_1 \cdot m_3) \sim (m_2 \cdot m_3)$ ainsi que $(m_3 \cdot m_1) \sim (m_3 \cdot m_2)$.*

La congruence $\sim$ sépare naturellement les éléments du monoïde M en classes de congruences, deux éléments $m_1, m_2 \in M$ faisant partie de la même classe de congruence si et seulement si $m_1 \sim m_2$. On dénote l'ensemble des classes induites par $\sim$, par $M/\sim$. De part la définition d'une congruence, l'ensemble quotient $M/\sim$ est alors muni d'une structure de monoïde (voir [27]).

Définition 2.3.7 (Monoïde libre). *Étant donné le monoïde M, on définit récursivement l'ensemble M^+ comme suit : pour tout $m \in M$, m fait partie de M^+ ; si $u, v \in M^+$, alors $uv \in M^+$; rien d'autre ne fait partie de M^+. Cet ensemble contient alors tous les mots non vides qu'il est possible de former en utilisant l'ensemble M comme alphabet. M^+ contient donc une infinité d'éléments.*

Prenant deux mots u, v de l'ensemble M^+, nous définissons la multiplication de u par v par la concaténation des deux mots, c'est-à-dire uv (observons que le produit est associatif). En ajoutant ε, le mot vide, nous obtenons l'ensemble M^*. Celui-ci, muni de l'opération de concaténation, forme alors un monoïde que nous appelons le monoïde libre *de M.*

2.3.2 Reconnaissance d'un langage par un semigroupe

Nous avons vu au début de ce chapitre que la seule utilisation des automates finis n'a pas été suffisante pour permettre aux chercheurs de caractériser les langages réguliers sans étoile. Nous avons également mentionner que les monoïdes peuvent être utilisés comme modèle de calcul servant à la reconnaissance des langages. Dans cette section, nous allons voir qu'un automate fini peut être représenté par un monoïde fini (monoïde de transition d'un automate fini). Puis, nous allons définir la reconnaissance d'un langage par un monoïde. Nous allons finalement énoncer le théorème de Schützenberger et discuter plus en détails de la relation entre les monoïdes finis et les langages

réguliers.

Définition 2.3.8 (Monoïde de transition). *Soit $\mathcal{A} = (Q, \Sigma, \delta)$, un automate fini où Q est l'ensemble fini des états, Σ est l'alphabet et δ est l'application de transition $(Q \times \Sigma) \to Q$ de l'automate fini. Chaque lettre $a \in \Sigma$ induit ainsi une application $\delta_a^* : Q \to Q$ que l'on généralise, à l'aide de l'opération de composition des applications et de son associativité, à Σ^* pour obtenir $\delta_w^* : Q \to Q$, où $w \in \Sigma^*$. L'ensemble des applications δ_a^* $(a \in \Sigma)$ et l'application identité (δ_1^*) génèrent alors un monoïde de transformation sur l'ensemble Q que l'on appelle le* monoïde de transition *de l'automate $\mathcal{A}$ et que l'on dénote $M(\mathcal{A})$. Notons que les applications δ_w^* formant un monoïde de transition sont généralement notés directement par le mot w pour en simplifier la notation.*

Soient $q \in Q$ et $w \in \Sigma^*$, respectivement un état de l'automate $\mathcal{A}$ et un mot sur l'alphabet Σ, remarquons que le résultat de $\delta_w^*(q)$ est l'état atteint par l'automate $\mathcal{A}$ si le mot w est lu à partir de l'état q. Si $L \subseteq \Sigma^*$ est un langage reconnu par l'automate fini $\mathcal{A}$ en utilisant $q_0 \in Q$ comme état initial et $F \subseteq Q$ comme ensemble d'états finaux, alors $\forall w \in \Sigma^*$, nous avons $w \in L \iff \delta_w^*(q_0) \in F$. Nous allons maintenant expliquer comment le monoïde de transition de l'automate $\mathcal{A}$ peut reconnaître le langage L.

Sans perte de généralité, supposons que $w = w_1 w_2 \dots w_n$, où les w_i sont des lettres de l'alphabet Σ. Par définition du monoïde de transition $M(\mathcal{A})$, pour tout w_i, nous avons $\delta_{w_i}^* \in M(\mathcal{A})$. Puisque $M(\mathcal{A})$ est fermé par la loi de composition des applications, nous savons également que $\delta_{w_1}^* \circ \delta_{w_2}^* \circ \dots \delta_{w_n}^* = \delta_r^*$ où $\delta_r^* \in M(\mathcal{A})$. Finalement, nous savons, par définition, que δ_w^* définit la même application que δ_r^*. Il suffit alors de vérifier si l'état correspondant à $\delta_r^*(q_0)$ est inclus dans l'ensemble F pour savoir si le mot w fait partie du langage L. L'exemple suivant illustre la reconnaissance d'un langage par un monoïde de transition.

	1	2	3	4
δ_1^*	1	2	3	4
δ_a^*	2	4	2	4
δ_b^*	4	3	4	4
δ_{aa}^*	4	4	4	4
$\boldsymbol{\delta_{ab}^*}$	**3**	4	3	4
δ_{ba}^*	4	2	4	4

Figure 2.5: Liste des applications δ_w^* générées par l'ensemble $\{\delta_1^*, \delta_a^*, \delta_b^*\}$ et formant le monoïde de transition de $\mathcal{B}$ ($M(\mathcal{B})$), où $\mathcal{B}$ est l'automate fini présenté à la Figure 2.1.

Exemple 2.3.9. *La Figure 2.1 présente l'automate fini* $\mathcal{B} = \{Q, \Sigma, \delta\}$ *et la Figure 2.5 donne la liste des applications de la forme* δ_w^* *générées par l'ensemble d'applications* $\{\delta_1^*, \delta_a^*, \delta_b^*\}$ *et qui forment le monoïde de transition de l'automate* $\mathcal{B}$. *La suite du calcul montre que les applications* δ_{aaa}^*, δ_{aab}^*, δ_{abb}^* *et* δ_{baa}^* *définissent la même application que* δ_{aa}^*, *tandis que* δ_{aba}^* *définit la même application que* δ_a^* *et* δ_{bab}^* *la même que* δ_b^*. *Le monoïde de transition de l'automate* $\mathcal{B}$ *contient donc 6 éléments,* $M(\mathcal{B}) = \{1, a, b, aa, ab, ba\}$ *et sa table de Cayley est présentée à la Figure 2.3. Pour reconnaître le langage* $L_{(ab)}$, *l'ensemble d'éléments acceptant sera* $P = \{ab\}$ *puisque* δ_{ab}^* *est la seule application de* $M(\mathcal{B})$ *qui conduit de l'état initial 1 à un des états faisant partie de l'ensemble* $F = \{3\}$. *Ainsi, nous savons que le mot abab fait partie du langage* $L_{(ab)}$ *puisqu'en multipliant ses lettres dans la table de Cayley du monoïde de transition, nous obtenons que* $a \circ b \circ a \circ b = ab$ *et l'élément ab fait partie de l'ensemble acceptant* P. *Par contre, le mot baba ne fait pas partie du langage* $L_{(ab)}$ *puisque* $b \circ a \circ b \circ a = ba \notin P$.

Puisque le monoïde de transition d'un automate $\mathcal{A}$ peut reconnaître tous les langages reconnus par celui-ci et que les automates finis reconnaissent exactement les langages réguliers, nous savons que les monoïdes finis peuvent reconnaître tous les langages réguliers. Généralisons maintenant le concept de reconnaissance d'un langage par un monoïde.

Définition 2.3.10. *Soit A, un alphabet et soit $(M, \cdot)$, un monoïde. On dit que le monoïde M reconnaît un langage $L \subseteq A^*$ s'il existe un morphisme $\varphi : A^* \to M$ et un sous-ensemble P de M tel que $\forall w \in A^*$, $w \in L \iff \varphi(w) \in P$.*

Le théorème suivant établit clairement la puissance des monoïdes finis comme modèle de calcul.

Théorème 2.3.11 ([20]). *Les monoïdes finis reconnaissent exactement les mêmes langages que les automates finis, c'est-à-dire les langages réguliers.*

On se souvient qu'un automate fini est dit minimal pour un langage L s'il est le plus petit automate (au sens du nombre d'états) reconnaissant L (voir [19]). Le monoïde de transition de l'automate minimal d'un langage L se nomme le *monoïde syntactique* de L, dénoté $M(L)$. Dans [29], Schützenberger introduit ce concept et démontre que $M(L)$ est le plus petit monoïde (au sens de la relation de division) reconnaissant L. La dénomination *monoïde syntactique* d'un langage $L \subseteq A^*$, où A est un alphabet, est due à la relation suivante. Soit $\sim_L$, la congruence définie sur A^* par $u \sim_L v$ si et seulement si pour tout $x, y \in A^*$, $xuy \in L \iff xvy \in L$, alors on appelle $\sim_L$ une congruence syntactique et le monoïde syntactique de L correspond au monoïde quotient $M(L) = A^*/\sim_L$. Schützenberger remarqua également que les propriétés algébriques du monoïde syntactique d'un langage reflètent les propriétés combinatoires de celui-ci. Le théorème suivant est un exemple de ce lien et il permet de répondre à la question posée précédemment, i.e. soit L, un langage régulier, est-ce que L est un langage régulier sans étoile.

Théorème 2.3.12 ([30]). *Soit L, un langage régulier. Alors L est sans étoile si et seulement si $M(L)$ est apériodique (s'il ne possède pas de sous-monoïde non trivial formant un groupe).*

Exemple 2.3.13. *À la Section 2.1, nous nous sommes demandés si le langage $L_{(ab)}$, reconnu par l'automate $\mathcal{B}$ présenté à la Figure 2.1, est un langage régulier sans étoile. Nous savons que le monoïde de transition de cet automate (M_2, Figure 2.3) est apériodique. Puisque l'automate $\mathcal{B}$ est minimal pour $L_{(ab)}$, $M_2 = M(L_{(ab)})$ et le langage $L_{(ab)}$ est sans étoile. En effet, considérant que l'expressions Σ^* représente un langage sans étoile puisque $\Sigma^* = \neg\emptyset$, $L_{(ab)}$ peut s'exprimer ainsi : $L_{(ab)} = ab(ab)^* = \big((a\Sigma^* \cap \Sigma^*b) \cap \neg(\Sigma^*aa\Sigma^* \cup \Sigma^*bb\Sigma^*)\big)$ (voir [27]).*

Eilenberg généralise ce concept en établissant une correspondance entre certaines familles de langages réguliers, appelées *variétés* de langages réguliers et certaines familles de monoïdes finis, appelées *pseudo-variétés* de monoïdes finis (voir [16]). Les définitions formelles de ces familles dépassent le cadre de ce travail. Mentionnons tout de même qu'une variété de langages réguliers est un ensemble de langages réguliers fermé par les opérations booléennes, par morphismes inverses et par quotients. D'une façon similaire, une pseudo-variété de monoïde fini est un ensemble de monoïdes finis fermés par sous-monoïdes, par quotients et par produits directs finis. Les Théorèmes 2.3.12 et 2.3.11 représentent alors des cas particuliers de cette théorie. Voici l'énoncé du théorème des variétés d'Eilenberg.

Théorème 2.3.14 ([16]). *Il existe une bijection entre les variétés de langages et les pseudo-variétés de monoïdes finis.*

2.4 Groupoïdes

Dans la section précédente, nous avons vu qu'un monoïde est un couple $(M, \cdot)$ où $\cdot$ est une loi de composition interne associative. Nous avons également vu que les monoïdes finis reconnaissent exactement les langages réguliers. Il est alors naturel de

se demander si l'utilisation d'une structure algébrique non associative ne permettrait pas de "capturer" plus de langages. C'est ainsi qu'à la Section 2.4.1, nous introduirons une structure algébrique non associative, les groupoïdes finis, et nous généraliserons à ceux-ci le concept de reconnaissance d'un langage. Nous étudierons également quels types de langages les groupoïdes finis peuvent reconnaître. Finalement, la Section 2.4.2 est réservée à l'étude d'un type particulier de groupoïdes finis, soit les boucles finies.

2.4.1 Définitions et reconnaissance d'un langage

Un *groupoïde* est un couple $(G, \cdot)$ où G est un ensemble et $\cdot$ est une loi de composition interne sur G. Aucune restriction n'est imposée à la loi de composition interne d'un groupoïde. Un semigroupe est donc un groupoïde associatif. Bien que la présence d'un élément identité ne soit pas requise par la définition d'un groupoïde, nous considérerons que tous les groupoïdes discutés dans cette section en contiennent un.

Pour le reste de cette section, $(G, \cdot)$ est un groupoïde tel que $1 \in G$ est un élément identité. Puisque la loi de composition interne d'un groupoïde n'est pas obligatoirement associative, nous n'obtiendrons pas nécessairement le même élément selon l'ordre dans lequel nous multiplions les lettres d'un mot (parenthétisation). L'expression 2^G est l'ensemble contenant tous les sous-ensembles de G. Nous définissons alors la fonction $eval : G^* \to 2^G$ comme suit : si $x \in G^*$ est un mot sur G, alors $eval(x)$ donne l'ensemble des éléments de G qu'il est possible d'obtenir en multipliant les lettres de x dans G (en changeant la parenthétisation). Plus formellement :

$$eval(x) = \begin{cases} 1 & \text{Si } x = \varepsilon \\ x & \text{Si } x \in G \\ \bigcup\limits_{\substack{x = uv, \\ u,v \neq \varepsilon}} eval(u) \cdot eval(v) & \text{Autrement} \end{cases} \tag{2.1}$$

$\diamond$	1	2	3	4	5
1	1	2	3	4	5
2	2	1	1	2	4
3	3	5	3	2	1
4	4	5	2	2	3
5	5	3	3	2	1

Figure 2.6: Table de Cayley du groupoïde $(G_1, \diamond)$, où $G_1 = \{1, 2, 3, 4, 5\}$.

En fait, le groupoïde $(G, \cdot)$ est un monoïde si et seulement si $\forall w \in G^*$, $|eval(w)| = 1$. Voici maintenant un exemple du calcul de la fonction d'évaluation (*eval*).

Exemple 2.4.1. *Considérons* $(G_1, \diamond)$, *le groupoïde fini présenté à la Figure 2.6 et* $225, 542, 41352 \in G_1^*$, *des mots sur* G_1. *Voyons quels sont les éléments qu'il est possible d'obtenir à partir du mot* 225.

$$\begin{aligned} ((2 \diamond 2) \diamond 5) &= (1 \diamond 5) = 5 \\ (2 \diamond (2 \diamond 5)) &= (2 \diamond 4) = 2 \end{aligned}$$

Nous avons donc $eval(225) = \{2, 5\}$. *D'une façon similaire, nous obtenons les évaluations suivantes :* $eval(542) = \{1\}$ *et* $eval(41352) = \{1, 2, 5\}$. *Notons que puisque* $((2 \diamond 2) \diamond 5) \neq (2 \diamond (2 \diamond 5))$, *nous savons que la loi de composition interne* $\diamond$ *n'est pas associative et que* G_1 *n'est pas un monoïde.*

Considérant que les groupoïdes que nous utilisons possèdent tous un élément identité, la définition d'un morphisme de groupoïde est la même que celle d'un morphisme de monoïde (voir Définition 2.3.2). Si E et F sont deux ensembles finis, on dit qu'un morphisme de monoïdes libres $\varphi : E^* \to F^*$ est strictement alphabétique[1] si $\varphi(E) \subseteq F$.

[1]Cette terminologie provient de [9]

Nous pouvons maintenant généraliser le concept de reconnaissance d'un langage aux groupoïdes.

Définition 2.4.2 ([21]). *Soit A, un alphabet et $L \subseteq A^*$, un langage. On dit que le groupoïde G reconnaît L s'il existe un morphisme strictement alphabétique $\varphi : A^* \to G^*$ et un ensemble acceptant $F \subseteq G$ tel que*

$$L = \{w \in A^* \mid \big(eval(\varphi(w)) \cap F\big) \neq \emptyset\}$$

Le théorème suivant est une extension naturelle des travaux de Kleene, il établit clairement la puissance des groupoïdes finis comme modèle de calcul.

Théorème 2.4.3 ([25]). *Les groupoïdes finis reconnaissent exactement les mêmes langages que les automates à pile, c'est-à-dire les langages hors-contextes.*

Tout comme Eilenberg a classifié l'ensemble des monoïdes finis selon les langages qu'ils reconnaissent, il est intéressant de vérifier les propriétés combinatoires des langages reconnus par d'autres familles de groupoïdes finis.

2.4.2 Quasigroupes et boucles

Un *quasigroupe* est un groupoïde supportant à la fois l'annulation à gauche et l'annulation à droite. Une *boucle* est un quasigroupe possédant un élément identité. Les boucles sont l'une des premières structures algébriques non associatives à avoir été largement étudiées au niveau algébrique (voir [26, 15]). Dans cette section, nous ferons un survol rapide de la théorie des boucles, définissant seulement les éléments qui nous seront utiles pour le reste de ce document. Pour plus de détails, nous invitons le lecteur à consulter [1, 2, 10].

Pour le reste de cette section, $(B, \circ)$ représentera une boucle finie. On dit que $B' \subseteq B$ forme une sous-boucle de B si $(B', \circ)$ est une boucle, i.e. si $\circ$ est une loi de composition interne pour B' et si B' contient l'élément identité. La définition suivante présente un type particulier de sous-boucles : les sous-boucles normales.

Définition 2.4.4 (Sous-boucle normale [21]). *Une sous-boucle $(N, \circ)$ de $(B, \circ)$ est normale si $\forall x, y \in B$, elle satisfait les équations suivantes :*

$$xN = Nx,\ (Nx)y = N(xy),\ y(xN) = (yx)N \tag{2.2}$$

Notons que les Équations (2.2) impliquent que $x(Ny) = (xN)y$.

Puisque N contient l'identité, $\forall x \in B$, $x \in Nx$. Donc $y \in Nx$ implique que $y = nx$ pour un certain $n \in N$ et $Ny = N(nx) = (Nn)x = Nx$, ce qui signifie que la sous-boucle normale N sépare les éléments de la boucle en co-ensembles disjoints. De plus, de par les lois d'annulations, les co-ensembles posséderont tous $|N|$ éléments. Ces co-ensembles forment une boucle avec l'opération $(Nx)(Ny) = N(xy)$. Cela est formalisé par le théorème suivant qui nous est nécessaire dans la preuve du Lemme 2.4.6. Ce dernier lemme nous sera particulièrement utile au Chapitre 4.

Théorème 2.4.5 ([11]). *Si N est une sous-boucle normale de B, alors N défini un morphisme surjectif naturel $x \longrightarrow Nx$ de B dans la boucle quotient B/N.*

Lemme 2.4.6. *Soit $(B, \circ)$ une boucle finie et $N \subseteq B$ une sous-boucle normale de B. Si la boucle quotient B/N est associative et si $w \in B^*$, alors $|eval(w)| \leq |N|$.*

Preuve. Par le Théorème 2.4.5, nous savons qu'il existe un morphisme surjectif naturel $x \longrightarrow Nx$. Sans perte de généralités, supposons que $w = w_1 w_2 \dots w_k$ où $w_i \in B$. Puisque

$1 \in N$, nous savons que $eval(w) \subseteq eval\big((N \circ w_1) \circ (N \circ w_2) \cdots \circ (N \circ w_k)\big)$, mais puisque B/N est associatif, $eval\big((N \circ w_1) \circ (N \circ w_2) \cdots \circ (N \circ w_k)\big) = (N \circ w_1) \circ (N \circ w_2) \cdots \circ (N \circ w_k)$ et $|eval(w)| \leq |(N \circ w_1) \circ (N \circ w_2) \cdots \circ (N \circ w_k)| \leq |N|$ □

La définition de reconnaissance d'un langage par une boucle finie est exactement la même que pour les groupoïdes en général (voir Définition 2.4.2) car la loi de composition interne d'une boucle n'est pas associative. Le théorème suivant indique quels types de langages sont reconnus par les boucles finies.

Soient A, un alphabet. On définit la fermeture polynômial des langages à groupes comme une union finie de langages de la forme $L_0 a_1 L_1 a_2 \cdots L_{n-1} a_n L_n$, où les L_i sont des langages reconnus par des groupes finis et où les a_i sont des lettres de l'alphabet. Il est démontrer par Pin dans [28] que cela correspond exactement aux langages réguliers ouverts. Nous traitons de ces langages dans le lemme suivant.

Théorème 2.4.7 ([7]). *Les boucles finies reconnaissent exactement les langages réguliers ouverts.*

Notons qu'une preuve démontrant que les boucles ne reconnaissent que des langages réguliers peut être construite à l'aide des éléments que nous introduirons à la Section 3.4.

S'inspirant de ce dernier résultat, des chercheurs ont commencé à classifier les boucles finies selon le type de langages réguliers qu'elles reconnaissent. Le résultat suivant démontre l'intérêt de cette recherche et établit un parallèle intéressant entre les travaux de Schützenberger et la théorie des boucles finies.

Théorème 2.4.8 ([8]). *Les boucles apériodiques finies ne reconnaissent que des langages réguliers ouverts et sans étoile.*

Un outil introduit dans [6], appelé le monoïde dérivé d'un groupoïde (voir Chapitre 3), permet d'utiliser la richesse et l'élégance de la théorie des semigroupes dans l'étude des boucles finies et des langages qu'elles reconnaissent. Bien que son utilité théorique est démontrée dans [6], très peu de choses sont connues à propos du monoïde dérivé. L'objectif de ce mémoire est d'une part de démontrer que le monoïde dérivé d'une boucle finie est calculable et d'autre part de trouver un algorithme efficace permettant de le calculer. Comme nous le verrons au Chapitre 4, le deuxième objectif a été seulement partiellement atteint.

CHAPITRE 3

MONOÏDE DÉRIVÉ

3.1 Introduction

Ce chapitre est consacré à l'étude du monoïde dérivé d'un groupoïde fini. À la Section 3.2, nous définissons le concept de monoïde dérivé d'un groupoïde et nous donnons plusieurs propriétés qui nous seront utiles tout au long de ce travail. Ensuite, nous démontrons que le monoïde dérivé d'une boucle finie est calculable (Sections 3.3 et 3.4). Cela nous permettra également d'obtenir une borne supérieure sur son nombre d'éléments à la Section 3.5. Finalement, nous présentons et analysons deux versions d'un algorithme permettant de calculer le monoïde dérivé d'une boucle finie (Sections 3.6 et 3.7).

3.2 Définitions et propriétés

Définition 3.2.1 (Monoïde dérivé). *Soit G, un groupoïde contenant un élément identité et soient $u, v \in G^+$. On dit que $(u \sim v) \Leftrightarrow \forall x, y \in G^*$, $(eval(xuy) = eval(xvy))$. Puisque $\sim$ est une congruence syntactique définie sur les éléments de G^+, l'ensemble quotient $G^+/\sim$ est naturellement muni d'une structure de monoïde que l'on appellera ici le* monoïde dérivé de G *(dénoté $(D(G), \cdot)$).*

Pour le reste de cette section, $(G, \diamond)$ représentera un groupoïde fini contenant un élément identité et $(B, \circ)$ une boucle finie. Considérant que les éléments de G^* sont ordon-

nés en respectant l'*ordre militaire*[1], on nomme *représentant* d'une classe de congruence le plus petit élément de la classe. On dénote par $[u]$ le représentant du mot $u \in G^*$. Notons toutefois que pour des raisons de technicité, le représentant du mot vide est l'identité et l'on considérera l'identité comme un mot de longueur 0. On dit qu'un mot est un *irréductible* si tous ses segments stricts sont les représentants de leur classe de congruence.

Lemme 3.2.2. *Si* $(G, \diamond)$ *est associatif, alors* $(D(G), \cdot) = (G, \diamond)$

Preuve. Soit $u \in G^*$ et soit $b \in G$, tel que $b = u_1 \diamond u_2 \cdots \diamond u_{|u|}$, $u_i \in G$. Nous allons démontrer que $[u] = b$, c'est-à-dire que $\forall x, y \in G^*$, $eval(xuy) = eval(xby)$, ce qui nous permettra de prouver que $D(G)$ et G contiennent les mêmes éléments. Nous démontrerons ensuite que $\cdot$ correspond à $\diamond$.

Puisque $(G, \diamond)$ est associatif, nous obtenons le même résultat si nous multiplions les lettres d'un mot en changeant la parenthétisation, donc $eval(xuy) = x_1 \diamond x_2 \cdots \diamond x_{|x|} \diamond (u_1 \diamond u_2 \cdots \diamond u_{|u|}) \diamond y_1 \diamond y_2 \cdots \diamond y_{|u|} = x_1 \diamond x_2 \cdots \diamond x_{|x|} \diamond (b) \diamond y_1 \diamond y_2 \cdots \diamond y_{|u|} = eval(xby)$. Donc $D(G) = G$.

La multiplication est également identique puisque si $a, b \in G = D(G)$, nous savons que $a \cdot b = [ab]$ par définition et en utilisant le principe décrit dans le dernier paragraphe, $[ab] = a \diamond b$, ce qui conclut la preuve. □

Lemme 3.2.3. *Soient* $s, t \in G^*, u \in G^+$, *alors* $[sut] = [s[u]t]$.

Preuve. $eval(xsuty) = eval(xs[u]ty)$ par définition puisque $u \sim [u]$. □

[1]On ordonne d'abord les mots par ordre croissant de longueur, puis pour une même longueur on les classe de façon lexicographique.

Lemme 3.2.4. *Tous les représentants sont des irréductibles.*

Preuve. La preuve est par contradiction. Soit r, un représentant dont le segment strict u n'est pas le représentant de sa classe de congruence. Alors $r = sut \neq s[u]t$ et le Lemme 3.2.3 nous indique que $s[u]t$ est dans la même classe que sut. Par contre, $u \neq [u]$ puisqu'il n'est pas le représentant de sa classe. Donc $s[u]t$ est plus petit que sut et $sut = r$ n'est pas le représentant de sa classe. □

Définition 3.2.5 (Successeur de représentant). *On appelle* successeur de représentant *un mot de la forme ra où r est le représentant d'une classe de congruence de $D(G)$ et a est un élément du groupoïde G.*

Lemme 3.2.6. *Tous les irréductibles formés d'au moins deux lettres sont des successeurs de représentant.*

Preuve. Supposons que nous avons un irréductible ra où $r \in G^+$ et $a \in G$. Par définition d'un irréductible, le segment strict r est nécessairement un représentant. □

Corollaire 3.2.7. *S'il n'y a aucun représentant de longueur n, il n'y en aura aucun de longueur plus grande que n.*

Comme nous le verrons, un des impacts majeurs de ce corollaire est de savoir à quel moment du calcul du monoïde dérivé nous connaissons tous les représentants.

Pour savoir si deux mots u et v font partie de la même classe, nous devons comparer pour tout $x, y \in G^*$, les évaluations de xuy et xvy. On nomme alors x le contexte gauche d'évaluation et y le contexte droit d'évaluation. Nous parlerons généralement du couple (x, y) comme contextes d'évaluation.

Théorème 3.2.8. *Soit G, un groupoïde et $D(G)$, son monoïde dérivé. Si L est un langage reconnu par G, alors L est reconnu par $D(G)$.*

Preuve. Si $\{M, \bullet\}$ est un monoïde, on dit par définition (voir Section 2.3.2) qu'un langage $L' \subseteq M^*$ est reconnu par M si et seulement s'il existe un morphisme de monoïde $\varphi : M^* \rightarrow M$ (par exemple le résultat de la multiplication des lettres dans le monoïde) et s'il existe un sous-ensemble $E \subseteq M$ tel que $L' = \{w \in M^* \mid \varphi(w) \in E\}$.

Sans perte de généralité, considérons que $L \subseteq G^*$. Par définition (voir Section 2.4.1), on dit que le langage $L \subseteq G^*$ est reconnu par le groupoïde G si et seulement s'il existe un sous-ensemble $F \subseteq G$ tel que $L = \{w \in G^* \mid eval(w) \cap F \neq \emptyset\}$.

Puisque $G \subseteq D(G)$, nous savons que $L \subseteq G^* \subseteq D(G)^*$. Définissons comme ensemble acceptant $E = \{[w] \in D(G) \mid eval([w]) \cap F \neq \emptyset\}$. Sachant que par définition du monoïde dérivé $eval(w) = eval([w])$, $\forall w \in D(G)^*$, l'ensemble E et le morphisme de monoïde $\varphi : G^* \rightarrow D(G)$ définit par $\varphi(w) = [w]$, permettent à $D(G)$ de reconnaître le langage L. □

3.3 Monoïde dérivé fini pour les boucles finies

La première étape vers la présentation d'un algorithme pour le calcul du monoïde dérivé d'un groupoïde est de démontrer sous quelles conditions il est fini, c'est ce que fait le Lemme 3.3.1. Puis, nous prouverons dans le théorème suivant que ces conditions sont respectées pour les boucles finies.

Lemme 3.3.1. *Soit $G = (\{a_1, a_2, \ldots, a_k\}, \diamond)$, un groupoïde fini ne reconnaissant que des langages réguliers et contenant un élément identité, alors $D(G)$ est fini.*

Preuve. Considérons le langage $L_{a_i} = \{w \in G^* \mid a_i \in eval(w)\}$, où $1 \leq i \leq k$. Soit la congruence syntactique $\sim_{a_i}$ définie sur G^* par $u \sim_{a_i} v$ si et seulement si $\forall x, y \in$

$G^*, (xuy) \in L_{a_i} \iff (xvy) \in L_{a_i}$. Cette congruence divise les mots en classes qui forment le monoïde syntactique de L_{a_i}, définit par $M(L_{a_i}) = G^*/\sim_{a_i}$. Nous savons que $M(L_{a_i})$ est le plus petit monoïde reconnaissant L_{a_i} et qu'il est fini puisque l'ensemble des monoïdes finis reconnaissent l'ensemble des langages réguliers ([27]).

La congruence définissant le monoïde dérivé ($\sim$), énoncée précédemment par $u, v \in G^+$, $(u \sim v) \Leftrightarrow \forall x, y \in G^*,\ eval(xuy) = eval(xvy)$, peut alors être redéfinie comme suit :

$$\begin{aligned} u \sim v \iff & u \sim_{a_1} v \quad \wedge \\ & u \sim_{a_2} v \quad \wedge \\ & \vdots \\ & u \sim_{a_k} v \end{aligned}$$

Dans cette situation, $eval(xuy)$ et $eval(xvy)$ contiendront effectivement les mêmes éléments si et seulement si $u \sim v$. Les classes du monoïde dérivé correspondent donc aux intersections des classes de tous les monoïdes syntactiques des langages de la forme L_{a_i} reconnus par le groupoïde. Puisque tous les $M(L_{a_i})$ sont finis, qu'il y a un nombre fini de langages de la forme L_{a_i} et que l'opération booléenne de conjonction ($\wedge$) préserve le caractère fini, le monoïde dérivé $D(G) = G^*/\sim$ est fini. □

Théorème 3.3.2. *Le monoïde dérivé d'une boucle finie est fini.*

Preuve. Par le Lemme 2.4.7, nous savons que les boucles finis ne reconnaissent que des langages réguliers. Jumelant ce résultat au Lemme 3.3.1, cela démontre le résultat désiré. □

3.4 Longueur des contextes d'évaluation

Pour qu'il soit possible de calculer le monoïde dérivé d'une boucle finie B, le nombre de tâches à effectuer doit être limité. Puisque le Théorème 3.3.2 démontre que $D(B)$ est fini, il nous reste à démontrer que le nombre de contextes d'évaluation qu'il est nécessaire d'utiliser pour vérifier que deux mots font partie de la même classe de congruence est lui aussi fini. Dans cette section, nous démontrerons qu'une telle borne supérieure existe et qu'elle vaut deux fois l'ordre de la boucle moins un.

3.4.1 Arbres pointés : notations et définitions

Dans cette sous-section, nous introduirons plusieurs notions dont les arbres binaires et les arbres pointés, qui nous serons nécessaires pour prouver que la longueur des contextes d'évaluation peut être limitée.

Définition (Groupoïde libre). *Soit A, un alphabet. On dénote le groupoïde libre de A par $A^{(*)}$ et on le définit récursivement comme suit : $\{a\} \in A^{(*)}$, $\forall a \in A$ et si $a, b \in A^{(*)}$, $(ab) \in A^{(*)}$. De plus $\{\varepsilon\} \in A^{(*)}$, rien d'autre n'est dans $A^{(*)}$ (l'opération n'est pas associative car on ajoute des parenthèses lors de la concaténation). Puisque ε est le mot vide, $(a\varepsilon) = (a) = a$.*

Il existe plusieurs façons de représenter les éléments d'un groupoïde libre. Pour les besoins de ce document, nous vous en présentons trois différentes. Premièrement, un élément $T \in A^{(*)}$ peut être représenté par une suite de lettres de l'alphabet A dans laquelle sont incluses des parenthèses. Par exemple, si $A = \{a, b, c\}$, alors $T_1 = ((a((ba)b))c) \in A^{(*)}$ et $T_2 = ((a(ba))((ac)c)) \in A^{(*)}$. La structure des éléments du groupoïde libre, telle que décrite dans la définition, fait en sorte qu'il y a exactement $n - 1$ parenthèses ouvrantes et $n - 1$ parenthèses fermantes dans un mot de n lettres

représenté ainsi. De plus, il y a toujours exactement deux éléments entre chaque paire de parenthèses.

Deuxièmement, pour représenter les parenthèses nous pouvons utiliser la notation polonaise inverse (ce qui revient à éliminer les parenthèses ouvrantes) pour obtenir $T_1 = aba)b))c) \in A^{(*)}$ et $T_2 = aba))ac)c)) \in A^{(*)}$. Considérant n'importe quel préfixe d'un mot T représenté ainsi, il y aura toujours au moins une lettre de plus que le nombre de parenthèses fermantes. Il est également important de remarquer qu'il n'existe qu'une seule possibilité d'ajouter les parenthèses ouvrantes pour que cela demeure bien parenthétisé (pour qu'il y ait exactement deux éléments entre chaque paire de parenthèses). Notons que dans ces deux premières représentations, le mot vide est dénoté directement par ε.

Finalement, la dernière méthode de représentation est l'utilisation d'un arbre binaire. Dans ce cas, les nœuds internes représentent les parenthèses et les feuilles sont les lettres du mot. Ainsi, si t_i est un nœud interne de $T \in A^{(*)}$ et si t_{ig} et t_{ir} sont respectivement son enfant gauche et son enfant droit, le nœud interne t_i indique les parenthèses suivantes : $(t_{ig}t_{ir})$. En effectuant un parcours en profondeur de l'arbre, on retrouve donc le mot et sa parenthétisation. La Figure 3.1 nous présente T_1 et T_2 sous cette dernière représentation. Le mot vide ε est représenté par un arbre sans aucun nœud[2]. Dans ce document, nous utiliserons principalement la représentation sous la forme d'arbres binaires.

Définition (Arbre pointé). *Soient A, un alphabet et X, une variable. On appelle arbre pointé sur $A \cup \{X\}$ un arbre binaire contenant la variable X dans exactement une feuille. $A_X^{(*)}$ représente alors l'ensemble des arbres pointés sur $A \cup \{X\}$. On peut le définir récursivement comme suit. Soit T_0^X l'arbre ayant comme unique feuille X, alors*

[2]Pour faciliter la discussion, nous supposerons l'existence d'un arbre vide ne contenant aucun nœud.

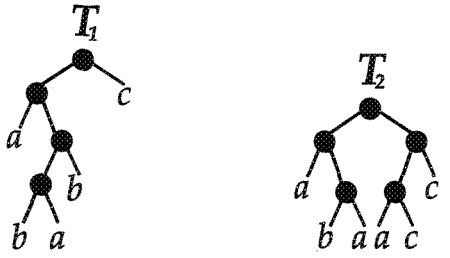

Figure 3.1: Représentation de $T_1 \in A^{(*)}$ et $T_2 \in A^{(*)}$ en notation d'arbre binaire

$T_0^X \in A_X^{(*)}$. *Soient* $T_1^X \in A_X^{(*)}$ *et* $T_2 \in A^{(*)}$, *si* T_g^X *représente l'arbre pointé ayant comme enfant gauche* T_1^X *et comme enfant droit* T_2 *et si* T_d^X *représente l'arbre pointé ayant comme enfant gauche* T_2 *et comme enfant droit* T_1^X, *alors* $T_g^X \in A_X^{(*)}$ *et* $T_d^X \in A_X^{(*)}$, *rien d'autre n'est dans* $A_X^{(*)}$.

Si $T_1^X, T_2^X \in A_X^{(*)}$, nous définissons l'opération $(T_1^X \cdot T_2^X)$ comme suit : on substitue la variable X présente dans T_1^X par l'arbre pointé T_2^X, ce qui nous donne un nouvel arbre pointé. De plus, si $T_1^X, T_2^X, \ldots, T_k^X$ sont des arbres pointés sur $A \cup \{X\}$, $T_1^X \cdot T_2^X \cdot \ldots \cdot T_k^X$ représente le même arbre pointé peu importe la parenthétisation utilisée (nous le dénoterons par $\prod_{i=1}^{k} T_i^X$), l'opération est donc associative. Considérant que l'ensemble est fermé sous l'opération et que celle-ci est associative, $(A_X^{(*)}, \cdot)$ forme un monoïde où l'identité T_0^X est représentée par l'arbre ayant comme unique nœud la variable X.

Nous pouvons aussi généraliser l'opération de substitution pour multiplier un arbre pointé sur $A \cup \{X\}$ par un arbre binaire sur A. Si $T_1^X \in A_X^{(*)}$ et $T_2 \in A^{(*)}$, $(T_1^X \cdot T_2)$ est calculé en substituant à la variable X de T_1^X, l'arbre binaire T_2. Le résultat de cette opération donnera toujours un arbre binaire dans $A^{(*)}$.

Voici maintenant deux définitions sur les arbres binaires et les arbres pointés qui nous permettrons de simplifier la notation dans les lemmes suivants. Dans ces deux définitions, $(G, \diamond)$ est un groupoïde et $T \in G^{(*)}$ est un arbre binaire sur G.

Définition (Frontière d'un arbre $(f(T))$). *Nous définissons récursivement la* frontière d'un arbre $T \in G^{(*)}$, *dénoté* $f(T)$, *comme suit. Pour tout* $g \in G$, $f(g) = g$. *Si* $T_g \in G^{(*)}$ *et* $T_d \in G^{(*)}$ *sont respectivement le sous-arbre gauche et le sous-arbre droit de* T *et si* $f(T_g) = u$ *et* $f(T_d) = v$, $f(T) = f\big((T_g T_d)\big) = uv$. *Autrement dit, la frontière de* T *est obtenue en ignorant les parenthèses. La frontière d'un arbre pointé se calcule de la même manière, mais la variable* X *sera présente dans le mot obtenu.*

Définition (Valeur d'un arbre $(v(T))$). *Nous définissons la valeur de l'arbre* $T \in G^{(*)}$, *dénoté* $v(T)$, *comme un homomorphisme* $v : G^{(*)} \to G$ *donnant la valeur de l'évaluation de* T *dans le groupoïde* $(G, \diamond)$ *en respectant les parenthèses induites par l'arbre. La valeur d'un arbre pointé donnera une expression à évaluer dans laquelle nous retrouvons la variable* X.

Dans le reste de la section, $(B, \circ)$ est une boucle finie et T^X est un arbre pointé sur $B \cup \{X\}$. Si T_1, T_2 sont deux arbres binaires sur B, alors

$$v(T_1) = v(T_2) \Rightarrow v(T^X \cdot T_1) = v(T^X \cdot v(T_1)) = v(T^X \cdot v(T_2)) = v(T^X \cdot T_2) \qquad (3.1)$$

L'inverse de l'Implication (3.1) est également vrai, mais cela est moins trivial. C'est ce qui est prouvé dans le lemme suivant.

Lemme 3.4.1. $v(T^X \cdot T_1) = v(T^X \cdot T_2) \Rightarrow v(T_1) = v(T_2)$.

Preuve. La preuve est par induction sur la hauteur de l'arbre T^X. Le cas de base est lorsque la hauteur est de 0, il n'y a dans ce cas que la racine comme nœud et elle représente la variable X. Nous avons alors $v(T^X \cdot T_1) = v(X \cdot T_1) = v(T_1)$ et $v(T^X \cdot T_2) = v(X \cdot T_2) = v(T_2)$. Donc dans ce cas $v(T_1) = v(T_2)$.

Supposons maintenant que la hauteur de T^X est n où $n \geq 1$. Puisqu'un arbre pointé contient exactement une fois la variable X, un seul des sous-arbres de la racine de T^X sera pointé. Sans perte de généralité, supposons que c'est celui de droit (l'autre cas étant symétrique) et appelons T_g le sous-arbre gauche et T_d^X le sous-arbre droit.

Puisque la valeur d'un arbre représente son évaluation en respectant les parenthèses qu'il induit, nous savons que $v(T^X) = v\big((T_g T_d^X)\big) = v(T_g) \circ v(T_d^X)$. Cela signifie que pour $i = 1, 2$, $v(T^X \cdot T_i) = v(T_g) \circ v(T_d^X \cdot T_i)$. Mais si $v(T_g) \circ v(T_d^X \cdot T_1) = v(T_g) \circ v(T_d^X \cdot T_2)$, alors $v(T_d^X \cdot T_1) = v(T_d^X \cdot T_2)$ par la loi de la cancellation de $(B, \circ)$. La hauteur de T_d^X étant de $n - 1$, $v(T_1) = v(T_2)$ par hypothèse d'induction. □

3.4.1.1 Modification d'un arbre binaire

Bien que provenant de [12], cette partie est incluse dans cet ouvrage car les deux lemmes qui y sont prouvés sont essentiels à la poursuite de ce travail. Ils nous permettrons de démontrer que la longueur des contextes d'évaluation nécessaires pour calculer le monoïde dérivé d'une boucle finie est bornée supérieurement. De plus, la Remarque 3.4.4 constitue une de nos contributions sur ces travaux.

Prenons un arbre binaire T sur B pouvant être exprimé sous la forme $T = (\prod_{i=1}^{k} T_i^X) \cdot R$, où $k \geq |B|$ et où $f(T) = w \in B^+$. Il est important de noter que nous désirons que k soit plus grand ou égal à $|B|$ pour pouvoir utiliser le principe du trou de pigeon lors d'une démonstration ultérieure. Si nous remplaçons T_i^X par $\overline{T_i^X}$, un arbre pointé ayant la même frontière que T_i^X (mais pas nécessairement la même valeur), nous ne modifions pas la frontière de T. Un exemple de modification est présenté aux Figures 3.2 et 3.3 où l'arbre T_i^X est modifié pour obtenir $\overline{T_i^X}$, un arbre pointé ayant la même frontière. Nous pouvons ainsi modifier plusieurs sous-arbres T_i^X et ce, sans altérer $f(T)$ (tant que nous conservons l'ordre des arbres pointés dans T).

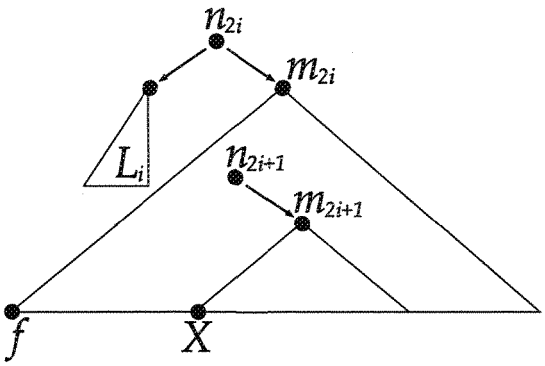

Figure 3.2: Représentation de l'arbre T_i^X

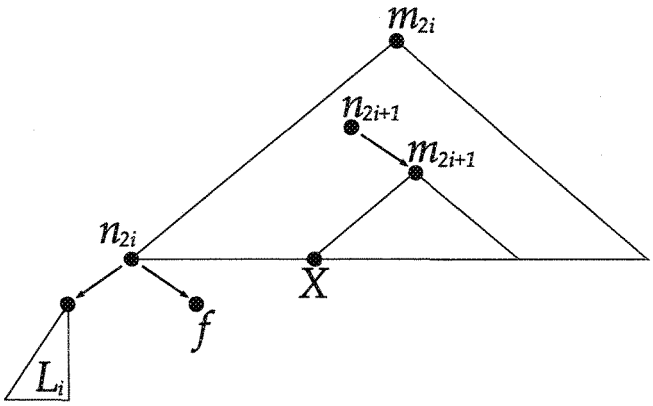

Figure 3.3: Représentation de l'arbre $\overline{T_i^X}$

Si $1 \leq x \leq y \leq k$, alors nous définissons $S(x,y)$ comme suit :

$$S(x,y) = \left(\prod_{i=1}^{x-1} T_i^X \cdot \prod_{i=x}^{y} \overline{T_i^X} \cdot \prod_{i=y+1}^{k} T_i^X \right) \cdot R. \tag{3.2}$$

Puisque dans la définition de $S(x,y)$ l'ordre des arbres pointés n'est pas modifié, il est facile de remarquer que la frontière de $S(x,y)$ est toujours la même que celle de T. De plus, nous pouvons ignorer le premier terme si $x = 1$ et l'avant-dernier si $y = k$.

Lemme 3.4.2. *Il existe deux entiers a, b tel que $1 \leq a \leq b \leq k$ et tel que $v(T) = v(S(a,b))$.*

Preuve. Tout d'abord, définissons $S(1,0) = T$. Puisque $k \geq |B|$, il existe, par le principe du trou de pigeon, deux entiers a et b tel que $1 \leq a \leq b \leq k$ et tel que :

$$\begin{aligned} v\left(S(1,a-1)\right) &= v\left(\prod_{i=1}^{a-1} \overline{T_i^X} \cdot \prod_{i=a}^{b} T_i^X \cdot \prod_{i=b+1}^{k} T_i^X \cdot R \right) \\ &= v\left(\prod_{i=1}^{a-1} \overline{T_i^X} \cdot \prod_{i=a}^{b} \overline{T_i^X} \cdot \prod_{i=b+1}^{k} T_i^X \cdot R \right) \\ &= v\left(S(1,b)\right) \end{aligned}$$

À partir de l'égalité précédente et par le Lemme 3.4.1, nous obtenons l'égalité suivante :

$$v\left(\prod_{i=a}^{b} T_i^X \cdot \prod_{i=b+1}^{k} T_i^X \cdot R\right) = v\left(\prod_{i=a}^{b} \overline{T_i^X} \cdot \prod_{i=b+1}^{k} T_i^X \cdot R\right)$$

Finalement par l'Implication (3.1),

$$\begin{aligned} v(T) &= v\left(\prod_{1}^{a-1} T_i^X \cdot \prod_{i=a}^{b} T_i^X \cdot \prod_{i=b+1}^{k} T_i^X \cdot R\right) \\ &= v\left(\prod_{1}^{a-1} T_i^X \cdot \prod_{i=a}^{b} \overline{T_i^X} \cdot \prod_{i=b+1}^{k} T_i^X \cdot R\right) \\ &= v(S(a,b)) \end{aligned}$$

□

Si T est un arbre et π un chemin dans T, on définit la *longueur à droite* de π comme son nombre d'arcs à droite. La *profondeur à droite* de T représente la longueur à droite maximale de l'arbre. D'ailleurs, si nous observons attentivement les Figures 3.2 et 3.3, nous pouvons remarquer que la profondeur à droite de la variable X est moins grande dans $\overline{T_i^X}$ que dans T_i^X.

Lemme 3.4.3. *Soient B, une boucle d'ordre q et T, un arbre binaire sur B dont la profondeur à droite est $d > 2q$. Il existe alors un arbre T'' qui possède la même frontière et la même valeur que T, mais dont la profondeur à droite $d'' \leq 2q$.*

Preuve. Définissons la fonction $N_k : B^{(*)} \to \mathbb{N}$ tel que $N_k(T)$ est le nombre de chemins de longueur à droite plus grand ou égal à k dans T. Nous allons démontrer qu'il existe un arbre T' possédant la même frontière et la même valeur que T, mais dont $N_d(T') <$

$N_d(T)$. Le résultat final pourra alors être obtenu par des applications successives de ce principe.

Supposons que π est un chemin de longueur à droite d dans T. Sans perte de généralité, nous pouvons supposer que le premier arc de π est un arc à droite. Sinon considérons le plus grand sous-arbre de T ayant cette propriété.

Soient $(n_0, m_0), (n_1, m_1), \ldots, (n_{2q}, m_{2q})$, les $2q+1$ premiers arcs à droite dans π et soit Q_i, le sous-arbre de T dont la racine est n_i. $\forall i,\ 0 \leq i \leq q-1$, définissons T_i^X comme l'arbre pointé sur $B \cup \{X\}$ obtenu de Q_{2i} en remplaçant Q_{2i+2} par X. Soit $R = Q_{2q}$, il est alors évident que $\forall i < q,\ Q_{2i} = T_i^X \cdot Q_{2i+2}$ et $T = Q_0 = T_0^X \cdot Q_2 = T_0^X \cdot T_1^X \cdot Q_4 = \cdots = \prod_{i=0}^{q-1} T_i^X \cdot Q_{2q}$. En résumé, nous avons :

$$T = \prod_{i=0}^{q-1} T_i^X \cdot R$$

La construction est telle que le chemin menant à X dans T_i^X contient exactement deux arcs à droite : (n_{2i}, m_{2i}) et (n_{2i+1}, m_{2i+1}). La variable X est alors une feuille contenue dans le sous-arbre droit de T_i^X, mais elle ne peut être la feuille la plus à gauche de ce sous-arbre (voir Figure 3.2).

Appelons L_i le sous-arbre gauche de T_i^X et f la feuille la plus à gauche du sous-arbre droit. Définissons $\overline{T_i^X}$ comme l'arbre pointé ayant m_{2i} comme racine et où la feuille f est remplacée par un nœud ayant L_i comme enfant gauche et f comme enfant droit (voir Figure 3.3).

Il est évident que $\overline{T_i^X}$ possède la même frontière que T_i^X. De plus, il est important de remarquer que la longueur à droite de tout chemin menant à une feuille dans L_i demeure inchangée et que la longueur à droite de tout chemin menant à une feuille dans R_i^X est diminué de 1, sauf pour le chemin menant à f dont la longueur à droite demeure inchangée.

La construction que nous venons d'effectuer nous permet maintenant d'utiliser le Lemme 3.4.2. Il existe donc deux entiers $0 \leq a \leq b \leq q-1$ tel que $v(T') = v(T)$ où T' est défini comme suit :

$$T' = \prod_{i=0}^{a-1} T_i^X \cdot \prod_{i=a}^{b} \overline{T_i^X} \cdot \prod_{i=b+1}^{q-1} T_i^X \cdot R$$

T' est nécessairement différent de T puisque, selon la définition, il y aura au moins un T_i^X qui sera remplacé par un $\overline{T_i^X}$.

Donc si d_i représente la profondeur à droite de T_i, alors $N_{d_i}(\overline{T_i^X}) \leq N_{d_i}(T_i^X)$. Maintenant prenons un chemin p dans T menant à une feuille w_i, ce chemin possède un équivalent p' dans T' qui mènera également à w_i. Ce chemin sera identique sauf pour le segment passant au travers T_i^X qui sera modifié si l'on remplace T_i^X par $\overline{T_i^X}$ et comme il a été dit précédemment, la longueur à droite de p' ne sera pas plus grande que celle de p. Par contre, la longueur à droite du chemin correspondant à π dans T' sera strictement plus petite que celle de π. Nous pouvons donc en conclure que $N_d(T') < N_d(T)$. □

Remarque 3.4.4. *Dans la preuve du Lemme 3.4.3 de Caussinus et Lemieux (voir [12]), il est exigé d'avoir une profondeur à droite plus grande que $2q$, le dernier arc à droite $(2q+1)$ étant utilisé pour uniformiser la preuve. De part l'utilisation que nous ferons de ce lemme, il est très important pour nous d'optimiser cette valeur. Considérant que ce dernier arc n'est pas requis, nous exigerons un arbre T d'une profondeur à droite $d \geq 2q$ qui sera transformé à un arbre T'' ayant la même frontière et la même valeur que T et ayant une profondeur $d'' < 2q$.*

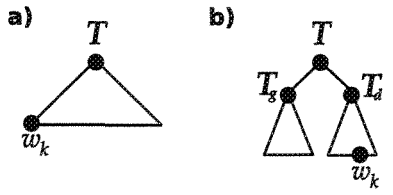

Figure 3.4: Position, selon le cas, de la lettre w_k dans l'arbre binaire T

3.4.2 Borne supérieure pour la longueur des contextes d'évaluation

Dans cette sous-section, $B = (B, \circ)$ représente toujours une boucle finie et $T \in B^{(*)}$ un arbre binaire. Nous allons ici démontrer que la longueur des deux contextes d'évaluation (gauche et droit) qu'il est nécessaire d'utiliser pour calculer $D(B)$ est bornée supérieurement par deux fois l'ordre de la boucle moins un.

Considérant que T est exprimé en notation polonaise inverse, définissons les fonctions $\mathcal{N}_l : B^{(*)} \to \mathbb{N}$ et $\mathcal{N}_p : B^{(*)} \to \mathbb{N}$, tel que $\mathcal{N}_l(T)$ donne le nombre de lettres et $\mathcal{N}_p(T)$ donne le nombre de parenthèses fermantes dans T.

Lemme 3.4.5. *Soit $T \in B^{(*)}$. Si $f(T) = w_1 w_2 \dots w_n$ et si T_k est le préfixe de T (en notation polonaise inverse) se terminant par w_k, la profondeur à droite de w_k dans T (en notation d'arbre binaire) équivaut alors à $\mathcal{N}_l(T_k) - \mathcal{N}_p(T_k) - 1$.*

Preuve. La preuve est par induction mathématique sur la profondeur p de w_k dans T. Le cas de base est quand p est égal à 0. Cela signifie alors que w_k est la feuille la plus à gauche dans la représentation en arbre binaire de T, donc $k = 1$ (voir Figure 3.4a). Nous avons alors $\mathcal{N}_l(T_k) - \mathcal{N}_p(T_k) - 1 = 1 - 0 - 1 = 0$.

Si $p > 0$, w_k peut se situer à n'importe quelle feuille, sauf celle à l'extrême gauche. Sans perte de généralité, nous pouvons supposer que le premier arc du chemin menant de la racine à w_k est un arc à droite (voir Figure 3.4b), sinon considérons le plus grand sous-arbre de T contenant w_k et ayant cette propriété. En notation polonaise inverse,

nous avons alors $T = T_g T_d$). Si T_{d_k} est le préfixe de T_d se terminant par w_k, nous devons donc démontrer que $\mathcal{N}_l(T_g T_{d_k}) - \mathcal{N}_p(T_g T_{d_k}) - 1 = p$.

$$\begin{aligned}
\mathcal{N}_l(T_g T_{d_k}) - \mathcal{N}_p(T_g T_{d_k}) - 1 &= (\mathcal{N}_l(T_g) + \mathcal{N}_l(T_{d_k})) - \\
&\quad (\mathcal{N}_p(T_g) + \mathcal{N}_p(T_{d_k})) - \\
&\quad 1 \\
&= (\mathcal{N}_l(T_g) - \mathcal{N}_p(T_g) - 1) + \\
&\quad (\mathcal{N}_l(T_{d_k}) - \mathcal{N}_p(T_{d_k}) - 1) + 1
\end{aligned}$$

Puisque T_g représente un arbre binaire plein de m $(1 \leq m < k)$ feuilles, nous savons qu'il contient exactement $m - 1$ nœuds internes, donc $m - 1$ arcs à droite. Considérant qu'un arc à droite représente une parenthèse fermante en notation polonaise inverse, cela nous donne l'équation suivante :

$$(\mathcal{N}_l(T_g) - \mathcal{N}_p(T_g) - 1) = m - (m-1) - 1 = 0$$

Nous avons également, par hypothèse d'induction, l'équation suivante :

$$(\mathcal{N}_l(T_{d_k}) - \mathcal{N}_p(T_{d_k}) - 1) = p - 1$$

Puisque $0 + (p-1) + 1 = p$, cela conclut la preuve. □

Lemme 3.4.6. *Il existe un automate non déterministe, fonctionnant avec une pile bornée par* $2|B| - 1$, *nous permettant de vérifier si l'évaluation d'un mot* $w \in B^*$ *contient au moins un élément commun avec un sous-ensemble* $F \subseteq B$ *de la boucle* $(eval(w) \cap F \neq \emptyset)$.

Preuve. L'Algorithme 3.1 donne la description informelle de l'automate à pile non

déterministe. Remarquez qu'il est plus habituel d'exprimer le non-déterminisme d'un automate à pile sous la forme de choix non déterministes effectués à chaque itération. L'algorithme présenté effectue plutôt l'ensemble des choix non déterministes au début et agit ensuite comme un automate déterministe. Les deux solutions sont équivalentes, mais la seconde nous permet d'expliquer plus facilement le comportement de l'automate.

Algorithme 3.1 : `Automate à pile non déterministe`$(w = w_1 \dots w_n)$ (Lemme 3.4.6)

```
Placer de façon non déterministe les (n -1) parenthèses fermantes
(notation polonaise inverse) à l'intérieur du mot w.

Soit T, le mot w parenthétisé.
Lire le premier caractère de T; le définir comme Valeur_Courante;

Tant qu'il y a des caractères à lire ou
 que la pile n'est pas vide, faire

    Si Valeur_Courante est une lettre
        Empiler la Valeur_Courante sur la pile,
        lire le prochain caractère dans T, et le
        définir comme la nouvelle Valeur_Courante;

    Sinon (Valeur_Courante est une parenthèse fermante)
        Dépiler les deux éléments sur le dessus de la
        pile (s, t), les multiplier dans B et définir
        le résultat comme nouvelle Valeur_Courante;

Si la Valeur_Courante est incluse dans l'ensemble F, accepter;
Sinon, rejeter;
```

Comme mentionné dans la description informelle, les choix non déterministes consistent à placer les $n-1$ parenthèses fermantes à l'intérieur du mot w. Nous obtenons alors $T \in B^{(*)}$ tel que $f(T) = w$. Nous devons maintenant démontrer que la taille de la pile équivaut à la profondeur à droite de T, c'est à dire maximum $2|B| - 1$.

Remarquons d'abord que si le caractère lu dans T est une lettre, la taille de la pile

de l'automate augmentera de 1. Elle diminuera plutôt de 1 si le caractère lu est une parenthèse fermante (sauf pour la dernière où elle diminue de 2).

Supposons que l'automate vient de traiter la lettre w_k, où $1 \leq k \leq n$, appelons T_k le préfixe de T dont le dernier caractère est w_k. La taille de la pile à cet instant est égale à $\mathcal{N}_l(T_k) - \mathcal{N}_p(T_k) - 1$, qui par le Lemme 3.4.5 équivaut à la profondeur à droite de w_k dans T. La taille maximale de la pile de l'automate est donc égale à la profondeur à droite de T qui par le Lemme 3.4.3 est borné supérieurement par $2|B| - 1$. □

Nous allons maintenant utiliser l'automate défini dans le lemme précédent pour prouver que tous les mots de longueur plus petite ou égale à deux fois l'ordre de la boucle sont suffisants comme contextes gauches d'évaluation. Nous utiliserons ensuite ce fait pour limiter la taille des contextes droits d'évaluation.

Lemme 3.4.7. *Soient $r_1, r_2 \in B^+$ et soient $x, y \in B^*$ tel que $eval(xr_1y) \neq eval(xr_2y)$. Il existe alors un $x' \in B^*$ tel que $|x'| < 2|B|$ et tel que $eval(x'r_1y) \neq eval(x'r_2y)$.*

Preuve. Sans perte de généralité, nous pouvons supposer qu'il existe un $a \in B$ tel que $a \in eval(xr_1y)$ et $a \notin eval(xr_2y)$. Soit l'automate à pile non déterministe décrit dans le lemme précédent avec $w = xr_1y$ et $F = \{a\}$.

Si, au moment exact où nous venons de lire le premier élément de r_1, le contenu de la pile est $a_1a_2 \ldots a_{k-1}a_k$ où les $a_i \in B$ et où a_k est le dernier élément ajouté, alors nous définissons $x' = a_ka_{k-1} \ldots a_2a_1$ comme le mot donnant la transposée du contenu de la pile. Puisque celle-ci est bornée supérieurement par $2|B| - 1$, $|x'| < 2|B|$.

Si $a \in eval(xr_1y)$, alors $a \in eval(x'r_1y)$ puisque nous n'avons qu'à empiler tous les éléments de x' pour nous retrouver exactement dans le même état (contenu de la pile, valeur courante et le mot restant à lire) qu'au début de la lecture de r_1 dans $eval(xr_1y)$.

Supposons maintenant que $a \in eval(x'r_2y)$, alors $a \in eval(xr_2y)$ car x' est une

évaluation partielle de x, tout ce qui peut être obtenu en évaluant x' peut donc l'être en évaluant x. Mais nous savons que $a \notin eval(xr_2y)$, donc $a \notin eval(x'r_2y)$. En résumé, nous avons $a \in eval(x'r_1y)$ et $a \notin eval(x'r_2y)$ et $|x'| < 2|B|$, ce qui conclut la preuve. □

Soit $\overleftarrow{B} = (S, \bullet)$, la transposée de la boucle $B = (S, \circ)$ définie comme suit : si $a \circ b = c$, alors $b \bullet a = c$. De plus, si $x = a_1a_2 \ldots a_{n-1}a_n \in B^*$, on notera par $\overleftarrow{x} = a_na_{n-1} \ldots a_2a_1$ l'équivalent de x dans $\overleftarrow{B}^*$. Nous identifions l'évaluation du mot x dans la boucle B par $eval_B(x)$ et l'évaluation du mot $\overleftarrow{x}$ dans la boucle $\overleftarrow{B}$ par $eval_{\overleftarrow{B}}(\overleftarrow{x})$.

Lemme 3.4.8. *Soit $x \in B^+$ alors $eval_B(x) = eval_{\overleftarrow{B}}(\overleftarrow{x})$.*

Preuve. La preuve se fera par induction mathématique sur la longueur de x. Dans le cas de base, $|x| = 1$, i.e. $x \in B$. Nous avons alors $eval_B(x) = \{x\} = eval_{\overleftarrow{B}}(\overleftarrow{x})$.

Si la longueur de x est plus grande que 1, nous savons par définition que

$$eval_B(x) = \bigcup_{\substack{x = uv, \\ u,v \neq \varepsilon}} eval_B(u) \circ eval_B(v) \tag{3.3}$$

et que

$$eval_{\overleftarrow{B}}(\overleftarrow{x}) = \bigcup_{\substack{\overleftarrow{x} = \overleftarrow{uv} = \overleftarrow{v}\,\overleftarrow{u}, \\ u,v \neq \varepsilon}} eval_{\overleftarrow{B}}(\overleftarrow{v}) \bullet eval_{\overleftarrow{B}}(\overleftarrow{u}) \tag{3.4}$$

Puisque $u, v \neq \varepsilon$, nous savons par hypothèse d'induction que $eval_{\overleftarrow{B}}(\overleftarrow{v}) = eval_B(v)$ et que $eval_{\overleftarrow{B}}(\overleftarrow{u}) = eval_B(u)$. Selon la définition de la transposée de B donnée plus tôt, $eval_B(v) \bullet eval_B(u) = eval_B(u) \circ eval_B(v)$, ce qui permet de transformer l'équation

(3.4) en l'équation (3.5).

$$eval_{\overleftarrow{B}}(\overleftarrow{x}) = \bigcup_{\substack{\overleftarrow{x} = \overleftarrow{uv} = \overleftarrow{v}\,\overleftarrow{u}, \\ u,v \neq \varepsilon}} eval_B(u) \circ eval_B(v) \tag{3.5}$$

$$= \bigcup_{\substack{x = uv, \\ u,v \neq \varepsilon}} eval_B(u) \circ eval_B(v) \tag{3.6}$$

$$= eval_B(x) \tag{3.7}$$

□

Théorème 3.4.9. *Soient $x, y \in B^*$ et $r_1, r_2 \in B^+$. Si $eval(xr_1y) \neq eval(xr_2y)$, alors il existe $x', y' \in B^*$ tel que $|x'| < 2|B|$ et $|y'| < 2|B|$ et tel que $eval(x'r_1y') \neq eval(x'r_2y')$.*

Preuve. Nous savons déjà par le Lemme 3.4.7 que la longueur du contexte gauche d'évaluation est bornée supérieurement par $2|B| - 1$, c'est-à-dire qu'il existe un x' tel que $eval_B(x'r_1y) \neq eval_B(x'r_2y)$ et tel que $|x'| < 2|B|$.

Considérons maintenant $eval_{\overleftarrow{B}}(\overleftarrow{y}\,\overleftarrow{r_1}\,\overleftarrow{x'})$. Le contexte gauche d'évaluation est alors $\overleftarrow{y}$, selon le Lemme 3.4.7 il existe un $\overleftarrow{y'} \in \overleftarrow{B}^*$ tel que $eval_{\overleftarrow{B}}(\overleftarrow{y'}\overleftarrow{r_1}\overleftarrow{x'}) \neq eval_{\overleftarrow{B}}(\overleftarrow{y'}\overleftarrow{r_2}\overleftarrow{x'})$ et que $|\overleftarrow{y'}| < 2|B|$. En utilisant le Lemme 3.4.8, nous savons que $eval_{\overleftarrow{B}}(\overleftarrow{y'}\overleftarrow{r_1}\overleftarrow{x'}) = eval_B(x'r_1y')$ et que $eval_{\overleftarrow{B}}(\overleftarrow{y'}\overleftarrow{r_2}\overleftarrow{x'}) = eval_B(x'r_2y')$, donc $eval(x'r_1y') \neq eval(x'r_2y')$. Puisque l'opération de transposition préserve la longueur du mot, $|y'| < 2|B|$. □

3.5 Borne supérieure pour $|D(B)|$

Au début de ce chapitre, nous mentionnons que deux mots $u, v \in B^*$ font partie de la même classe de congruence du monoïde dérivé si pour tout $x, y \in B^*$, $eval(xuy) = eval(xvy)$. Suite au Théorème 3.4.9, nous savons qu'il n'est nécessaire d'avoir cette égalité que pour tout $x, y \in Ctx_{(<2|B|)}$ où $Ctx_{(<2|B|)} = \{w \in B^* \mid |w| < 2|B|\}$.

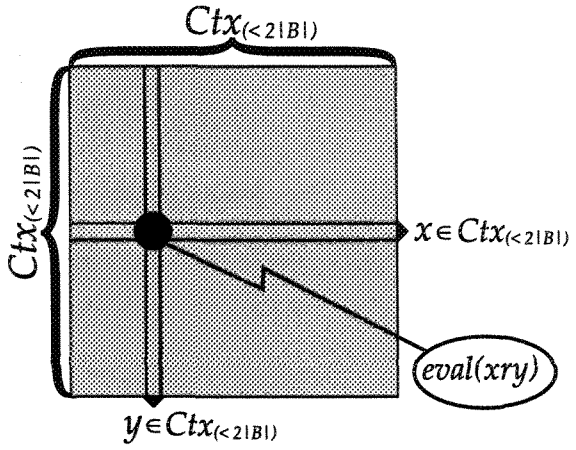

Figure 3.5: Table des évaluations de la classe $r \in D(B)$ pour tout contextes d'évaluation nécessaires.

La Figure 3.5 présente une classe de congruence du monoïde dérivé comme une table d'évaluation où les lignes représentent les contextes gauches d'évaluation et les colonnes les contextes droits d'évaluation. Nous retrouvons à la jonction de la ligne x et de la colonne y l'ensemble d'évaluation de xry où r est un mot faisant partie de la classe de congruence concernée. Notons que les éléments de l'ensemble $Ctx_{(<2|B|)}$ sont utilisés comme contextes d'évaluation (lignes et colonnes).

Le fait de représenter une classe de congruence du monoïde dérivé de cette façon nous permet d'observer qu'une borne supérieure grossière peut être donnée. En effet, il existe un nombre limité de tables différentes. Pour chacune des $|Ctx_{(<2|B|)}|^2$ cases, il y a $2^{|B|}$ ensembles possibles, ce qui nous donne un total de $\left(2^{|B|}\right)^{|Ctx_{(<2|B|)}|^2} = 2^{|B|\cdot|Ctx_{(<2|B|)}|^2}$ tables différentes.

Nous pouvons également exprimer le nombre d'éléments inclus dans l'ensemble $Ctx_{(<2|B|)}$ en fonction de la taille de la boucle puisque celui-ci est composé de tous les mots de longueur $2|B|$ et moins. Il est à noter que l'identité n'est pas utilisée comme lettre lors de la construction des mots puisqu'elle n'a aucune influence sur l'ensemble d'évaluation obtenu. Cela signifie donc que nous devons calculer le nombre de mots de

$2|B|-1$ caractères et moins avec $|B|-1$ lettres différentes.

$$\begin{aligned} |Ctx_{(<2|B|)}| &= \sum_{i=0}^{2|B|-1} (|B|-1)^i \\ &= \frac{(|B|-1)^{2|B|}-1}{|B|-2} \end{aligned}$$

Si nous conservons l'identité comme lettre, nous pouvons compléter les mots plus petits que $2|B|-1$ à l'aide de l'identité pour qu'ils aient une longueur de $2|B|-1$. Nous voyons ainsi que l'ensemble des mots de longueur $2|B|-1$ avec l'identité contient tous les contextes nécessaires au moins une fois (la plupart sont répétés plusieurs fois). Cela nous permet d'obtenir une borne plus grossière, mais plus élégante. Nous avons donc

$$|Ctx_{(<2|B|)}| < |B|^{2|B|-1}$$

$$\begin{aligned} |D(B)| &\leq 2^{|B|\left(\frac{(|B|-1)^{2|B|}-1}{|B|-2}\right)^2} \\ &< \left(2^{|B|}\right)^{\left(|B|^{2|B|-1}\right)^2} \\ &= 2^{|B|^{4|B|-1}} \end{aligned}$$

En pratique, nous pensons que cette borne n'est pas réaliste. Nous conjecturons plutôt que la longueur des représentants, tout comme celle des contextes d'évaluation, est bornée supérieurement par deux fois la longueur de la boucle moins un. Cela signifierait donc une nouvelle borne supérieure de $(|B|-1)^{2|B|-1}$ éléments pour le monoïde dérivé. Pour le moment, nous sommes par contre dans l'incapacité de démontrer cette conjecture.

3.6 Algorithme original ($Alg.v1$)

Utilisant les résultats obtenus au Théorème 3.4.9 (la longueur des contextes d'évaluation est bornée supérieurement), représentés à la Figure 3.5, et au Théorème 3.3.2 ($D(B)$ fini pour une boucle B finie), nous pouvons maintenant présenter un algorithme dont nous savons que le temps d'exécution est fini.

La fonction principale de l'algorithme se nomme $\texttt{Calculer\_Repr}_{Alg.v1}(\ )$. Elle a pour objectif de calculer l'ensemble des éléments formant le monoïde dérivé d'une boucle finie donnée. Voici ses éléments d'entrées/sorties :

ENTRÉE : La boucle $B = (\{a_1, a_2, \ldots, a_n\}, \circ)$.

SORTIE : La liste des représentants du monoïde dérivé de B.

Au cours de son exécution, l'algorithme conserve en mémoire une structure de données, soit la liste des représentants trouvés jusqu'à maintenant ($Repr$). $Suiv(u)$ donne l'élément suivant le mot u dans $Repr$ et si u est le dernier représentant de la liste, $Suiv(u)$ retourne `NULL`. La liste des représentants est initialisée avec tous les mots d'une lettre puisque $eval(a_i) = eval(a_j) \iff a_i = a_j$. En effet, puisque si B est un groupe, $D(B) = B$ (voir Lemme 3.2). En tout, trois fonctions sont nécessaires pour le calcul.

La fonction $\texttt{Calculer\_Ctx}_{Alg.v1}(B, |B|)$ (Fonction 3.3) a pour tâche de calculer la liste $Ctx_{(< 2|B|)}$, celle-ci contient tous les mots de longueur $2|B| - 1$ et moins, soit l'ensemble des contextes d'évaluation nécessaires selon le Théorème 3.4.9. Elle n'est appelée qu'une seule fois au début de $\texttt{Calculer\_Repr}_{Alg.v1}$ pour initialiser la liste des contextes d'évaluation.

La fonction $\texttt{Trouver\_Repr}_{Alg.v1}(u)$ (Fonction 3.2) vérifie s'il existe au plus un $v \in Repr$ tel que $\forall x, y \in Ctx_{(< 2|B|)}$, $eval(xuy) = eval(xvy)$. Si un tel v existe, il est

retourné comme représentant de u. S'il ne fait partie d'aucune des classes présentes, la fonction retourne le mot entré (u) puisqu'il constituera le représentant d'une nouvelle classe de congruence que l'on devra ajouter.

La fonction $\texttt{Calculer\_Repr}_{Alg.v1}(B, |B|)$ (Fonction 3.1), calcule la liste *Repr* tel que pour chaque pair $u, v \in Repr$ $(u \neq v)$, $\exists x, y \in Ctx_{(< 2|B|)}$ tel que $eval(xuy) \neq eval(xvy)$. Elle traite chacun des représentants auxquels sont ajoutés, tour à tour, chacun des éléments de B pour former un nouveau mot. On teste ensuite pour savoir si ce nouveau mot (ua_i) fait partie de l'une des classes déjà représentées ou bien s'il constitue le représentant d'une nouvelle classe (voir $\texttt{Trouver\_Repr}_{Alg.v1}$). L'exécution de la fonction se termine lorsqu'il n'y a plus de représentants qui n'ont pas été traités puisque nous savons alors par le Corollaire 3.2.7 qu'il n'y aura plus de nouveaux représentants.

Fonction 3.1 : $\texttt{Calculer\_Repr}_{Alg.v1}(B, |B|)$

```
1:  Repr ←< a_1, a_2, ..., a_n >
2:  Ctx_(< 2|B|) ← Calculer_Ctx_Alg.v1(B, |B|)
3:  u ← a_1
4:
5:  Tant que u ≠ NULL faire
6:    Pour i ← 1 à n faire
7:      v ← Trouver_Repr_Alg.v1(ua_i)
8:      Si v = ua_i alors
9:        Repr → ajouter(ua_i)
10:   u ← suiv(u)
11:
12: retourner Repr
```

Fonction 3.2 : $\texttt{Trouver\_Repr}_{Alg.v1}(u)$

```
Si ∃v ∈ Repr tel que ∀x, y ∈ Ctx_(< 2|B|),
   eval(xuy) = eval(xvy) alors
  retourner v      //car v est le représentant de u
Sinon
  retourner u      //car u est son propre représentant
```

Fonction 3.3 : $\texttt{Calculer\_Ctx}_{Alg.v1}(B, |B|)$

1: $Ctx_{(< 2|B|)} \rightarrow ajouter(\varepsilon)$
2:
3: **Pour** $k \leftarrow 1$ à $2n-2$ **faire**
4: **Pour chacun** des n^{k-1} derniers mots (u) de la liste $Ctx_{(< 2|B|)}$ **faire**
5: **Pour** $i \leftarrow 1$ à n **faire**
6: $Ctx_{(< 2|B|)} \rightarrow ajouter(ua_i)$
7:
8: **retourner** $Ctx_{(< 2|B|)}$

3.6.1 Rectitude de l'algorithme original

Pour prouver que l'algorithme présenté calcule effectivement le monoïde dérivé, nous devons s'assurer que chacune des classes données par l'algorithme sont distinctes. Nous devons également s'assurer que tous les représentants sont calculés et qu'il s'agit réellement des représentants du monoïde dérivé (plus petits éléments de leur classe de congruence).

Lemme 3.6.1 (Invariant). *Après chaque itération de la boucle "pour" de la fonction* $\texttt{Calculer\_Repr}_{Alg.v1}$ *(Fonction 3.1), les classes sont toutes distinctes, i.e. :*

$$\forall u, v \in Repr,\ u \neq v,$$
$$\exists x, y \in Ctx_{(< 2|B|)},\ \text{tel que}$$
$$eval(xuy) \neq eval(xvy)$$

Preuve. La preuve est par induction sur le nombre d'éléments dans *Repr*. Au départ, il y a n éléments dans la liste, soit chacune des lettres de B. Ces éléments font effectivement partie de classes distinctes car si $x = y = \varepsilon, eval(xay) = a \neq b = eval(b) = eval(xby)$ où $a, b \in Repr$ et $a \neq b$.

Le seul endroit où un élément est ajouté à la liste des représentants, c'est à la

ligne 9 de la fonction $\texttt{Calculer\_Repr}_{Alg.v1}$. Par hypothèse d'induction, nous savons que l'invariant est vrai pour tout $v_1, v_2 \in Repr, v_1 \neq v_2 \neq ua_i$. Puisque nous ajoutons ua_i, il nous suffit maintenant de vérifier que la relation est vraie entre ua_i et tous les autres représentants. Or, si ua_i est ajouté à la liste des représentants, c'est parce que la fonction $\texttt{Trouver\_Repr}_{Alg.v1}$ n'a pas trouvé de $v \in Repr$, tel que $v \neq ua_i$ et tel que $\forall x, y \in Ctx_{(< 2|B|)}$, $eval(xua_iy) = eval(xvy)$. Par conséquent $\forall v \in Repr$, $v \neq ua_i$, $\exists x, y \in Ctx_{(< 2|B|)}$ tel que $eval(xua_iy) \neq eval(xvy)$, ce qui conclut la preuve. □

Lemme 3.6.2. *Pour tous les successeurs de représentant ra* ($\forall r \in Repr$, $\forall a \in B$) *ou bien* $ra \in Repr$, *ou bien* $\exists v \in Repr$ *tel que* $\forall x, y \in Ctx_{(< 2|B|)}$, $eval(xray) = eval(xvy)$.

Preuve. L'exécution de la fonction $\texttt{Calculer\_Repr}_{Alg.v1}$ se termine lorsqu'elle a traité chacun des éléments de la liste des représentants. Pour chacun de ces éléments (r), elle lui ajoute tour à tour chacune des lettres de la boucle (a). La fonction $\texttt{Trouver\_Repr}_{Alg.v1}$ vérifie alors s'il existe un $v \in Repr$ tel que pour tout $x, y \in Ctx_{(< 2|B|)}$, $eval(xray) = eval(xvy)$. Si oui, alors v est dans la même classe que ra, sinon cela signifie que ra est le premier élément d'une nouvelle classe puisque $\forall v \in Repr$, $\exists x, y \in Ctx_{(< 2|B|)}$ tel que $eval(xray) \neq eval(xvy)$. □

Lemme 3.6.3. $|Repr| = |D(B)|$.

Preuve. L'Invariant 3.6.1 nous permet d'affirmer que $|Repr| \leq |D(B)|$ puisqu'il est vérifié que toutes les classes de $Repr$ sont distinctes.

La preuve est par induction sur la longueur des éléments de $Repr$. Le cas de base est simple puisque nous savons déjà que $Repr$ contient tous les représentants de longueur 1, car ce sont toutes les lettres de la boucle.

Nous voulons vérifier que *Repr* contient le même nombre de mots de longueur l que le nombre de représentants de longueur l. Nous savons, par hypothèse d'induction, que c'est vrai pour $l-1$ et par les Lemmes 3.2.4 (tous les représentants sont des irréductibles), 3.2.6 (tous les irréductibles sont des successeurs de représentant) et 3.6.2 (tous les successeurs de représentant on été traités), nous savons que *Repr* contient le même nombre de mots de longueur l que le nombre de représentants de longueur l. □

Théorème 3.6.4. *Suite à l'exécution de la fonction* $\texttt{Calculer\_Repr}_{Alg.v1}(B, |B|)$, *le groupoïde* $(Repr, \bullet)$, *où* $u \bullet v = \texttt{Trouver\_Repr}_{Alg.v1}(uv)$, *est le monoïde dérivé de* B $\big((Repr, \bullet) = (D(B), \cdot)\big)$.

Preuve. La preuve est en deux parties, nous devons d'abord prouver que *Repr* contient les mêmes éléments que $D(B)$, puis nous devons prouver que $(u \circ v) = (u \cdot v)$, $\forall u, v \in Repr$.

Nous savons déjà que *Repr* et $D(B)$ contiennent le même nombre d'éléments (voir Lemme 3.6.3), nous devons par contre s'assurer que $\forall r \in Repr$, r est le plus petit élément de sa classe ($[r] = r$). Nous prouverons par contradiction que c'est le cas. Soit $r \in Repr$, le plus petit élément de *Repr* tel que $[r] \neq r$. Cela signifie donc qu'il existe un élément $r' \in B^*$ tel que r' est plus petit que r selon l'ordre militaire et que $[r] = r'$. Selon les Lemmes 3.2.4 et 3.2.6, nous savons que r' est un successeur de représentant et r en est également un puisqu'il est le plus petit élément de *Repr* à ne pas être un représentant. Mais la fonction $\texttt{Calculer\_Repr}_{Alg.v1}$ traite tous les successeurs de représentant et ce en ordre militaire croissant, donc r' a été traité avant r et nous avons $r' \in Repr$ et $r \notin Repr$.

Ensuite nous devons prouver que $\texttt{Trouver\_Repr}_{Alg.v1}(uv) = (u \cdot v)$, $\forall u, v \in Repr$. Rappelons-nous que $(u \cdot v) = r$ tel que $r \in D(B)$ et $\forall x, y \in Ctx_{(< 2|B|)}$, $eval(xuvy) =$

$eval(xry)$. Or, c'est exactement le calcul effectué par la fonction $\texttt{Trouver\_Repr}_{Alg.v1}$ pour $r \in Repr = D(B)$. Ce qui conclut la preuve. □

3.6.2 Analyse des temps d'exécution

De toute évidence, les temps d'exécution des Fonctions 3.1 ($\texttt{Calculer\_Repr}_{Alg.v1}$) et 3.2 ($\texttt{Trouver\_Repr}_{Alg.v1}$) dépendent du nombre d'éléments du monoïde dérivé. Nous avons obtenu à la Section 3.5 une borne supérieure grossière pour $|D(B)|$, mais nous ne connaissons pas d'expression algébrique nous permettant d'obtenir plus précisément ce nombre. Aussi, nous exprimerons l'analyse du temps d'exécution des algorithmes en fonction de $|D(B)|$ et de $|B|$ et non uniquement en fonction de $|B|$.

3.6.2.1 $\texttt{Calculer\_Ctx}_{Alg.v1}(B, |B|)$

Le rôle de cette fonction est d'ajouter tous les mots qui sont nécessaires comme contextes d'évaluation à l'ensemble $Ctx_{(<2|B|)}$. En considérant que la construction et l'ajout d'un mot à l'ensemble prennent un temps constant, nous aurons un temps d'exécution égal à la quantité de contextes nécessaires. Comme nous l'avons vu à la Section 3.5, $|Ctx_{(<2|B|)}| = \frac{(|B|-1)^{2|B|}-1}{|B|-2} \leq |B|^{2|B|-1}$. Le temps d'exécution de la fonction $\texttt{Calculer\_Ctx}_{Alg.v1}$ est donc dans l'ordre de $\theta\left(\frac{(|B|-1)^{2|B|}-1}{|B|-2}\right)$ ou plus simplement, il est borné supérieurement par $O(|B|^{2|B|-1})$.

3.6.2.2 $\texttt{Trouver\_Repr}_{Alg.v1}(u)$

Les opérateurs logiques $\exists$ et $\forall$ nécessitent tout deux une boucle parcourant l'ensemble des éléments concernés pour effectuer leur tâche. Nous aurons donc besoin d'une boucle pour parcourir l'ensemble des éléments $v \in Repr$ et de deux autres imbriquées pour parcourir l'ensemble des éléments $x, y \in Ctx_{(<2|B|)}$. Sachant que $|Repr| = |D(B)|$

(voir Lemme 3.6.3), nous devons alors effectuer $|D(B)| \cdot |Ctx_{(< 2|B|)}| \cdot |Ctx_{(< 2|B|)}| \leq |D(B)| \cdot |B|^{4|B|-2}$ exécutions du test $eval(xuy) = eval(xvy)$ en pire cas. En meilleur cas, l'élément v recherché est le premier de la liste et nous avons moins de $|B|^{4|B|-2}$ tests à effectuer.

La fonction $eval(x_1 x_2 \ldots x_{|x|})$ peut être implémentée à l'aide d'un algorithme de programmation dynamique dont la récurrence est $T[i,j] = eval(x_i \ldots x_j) = \bigcup_{i \leq k \leq j} T[i,k] \times T[k,j]$, où $\times$ est le produit point à point des ensembles d'évaluation dans B. La multiplication point à point ce fait en temps $|B|^2$ et l'algorithme effectue $|x|^3$ multiplications. Le temps d'exécution de $eval(x)$ est donc dans l'ordre de $\theta(|x|^3 \cdot |B|^2)$.

Il est impossible d'estimer le temps d'exécution de l'ensemble des tests $eval(xuy) = eval(xvy)$ effectués par la fonction $\texttt{Trouver\_Repr}_{Alg.v1}$, car bien que nous connaissons la longueur maximale de x et y, nous ne connaissons aucune borne supérieure sur la longueur des représentants u et v. Par contre, nous conjecturons à la Section 3.5 que la longueur des représentants est bornée supérieurement par $2|B| - 1$. Considérant que $|B|$ est négligeable par rapport à $|D(B)|$, nous obtenons alors que le temps d'exécution de la fonction $\texttt{Trouver\_Repr}_{Alg.v1}$ est dans l'ordre de $O\big(|D(B)| \cdot |Ctx_{(< 2|B|)}|^2\big)$, ce qui équivaut à $O\big(|D(B)| \cdot (\frac{(|B|-1)^{2|B|}-1}{|B|-2})^2\big)$.

3.6.2.3 $\texttt{Calculer\_Repr}_{Alg.v1}(B, |B|)$

C'est la fonction principale de l'algorithme original et son travail est effectué en grande partie en appelant les fonctions $\texttt{Calculer\_Ctx}_{Alg.v1}$ et $\texttt{Trouver\_Repr}_{Alg.v1}$. La fonction débute d'ailleurs par le calcul de l'ensemble $Ctx_{(< 2|B|)}$, ce qui demandera un temps de $O(|B|^{2|B|-1})$. La suite de l'algorithme consiste en $|D(B)| \cdot |B|$ exécutions de la fonction $\texttt{Trouver\_Repr}_{Alg.v1}$, ce qui nous donne un minimum de $|D(B)|^2 \cdot |Ctx_{(< 2|B|)}|^2 \cdot |B|^3$ exécutions du test $eval(xuy) = eval(xvy)$ en pire cas. En considé-

rant toujours que $|B|$ est négligeable par rapport à $|D(B)|$, le temps d'exécution de $\texttt{Calculer\_Repr}_{Alg.v1}$ est dans l'ordre de $O\big(|D(B)|^2 \cdot |Ctx_{(<2|B|)}|^2\big)$, ce qui équivaut à $O\big(|D(B)|^2 \cdot (\frac{(|B|-1)^{2|B|}-1}{|B|-2})^2\big)$.

3.7 Utilisation d'une fonction de réduction pour la multiplication ($Alg.v2$)

Une des lacunes importantes de l'algorithme original ($Alg.v1$) est l'inefficacité de sa fonction de multiplication de deux représentants. En effet, celle-ci est simulée à l'aide de la fonction $\texttt{Trouver\_Repr}_{Alg.v1}$ qui doit effectuer un nombre exponentiel de tests pour obtenir le résultat. Froidure et Pin apportent une idée intéressante dans [17] pour calculer la multiplication de deux mots sur un semigroupe. Ils utilisent une fonction de réduction se servant de règles produites par l'algorithme. Dans le cas du monoïde dérivé, nous pouvons utiliser cette idée et conserver en mémoire, durant le calcul des représentants, un ensemble de règles ($Regles$) associant tous les successeurs de représentant à leur propre représentant (notez que le représentant d'un mot ra n'est pas nécessairement r). Dans l'algorithme original (Section 3.6), cette information est calculée mais simplement inutilisée. La Fonction 3.4 présente $\texttt{Calculer\_Repr}_{Alg.v2}$. Cette nouvelle version (les lignes 3, 10, 12 et 13 ont été ajoutées à la version originale) permet de conserver en mémoire l'ensemble $Regles$. Cela permettra de calculer la table de Cayley du monoïde dérivé de façon beaucoup plus efficace. Les fonctions $\texttt{Trouver\_Repr}_{Alg.v2}$ (Fonction 3.5) et $\texttt{Calculer\_Ctx}_{Alg.v2}$ (Fonction 3.6) sont identiques à leur version originale.

La fonction $\texttt{Mult}_{Alg.v2}(u, v)$ (Fonction 3.7) prend en entrée deux représentants (u et v) et retourne le résultat de la multiplication des deux éléments dans le monoïde dérivé en se basant sur les $Regles$. Soit $v = v_1 v_2 \dots v_{|v|}$ où $v_i \in B$, puisqu'il y a une règle pour

Fonction 3.4 : `Calculer_Repr`$_{Alg.v2}(B, |B|)$

1: $Repr \leftarrow < a_1, a_2, \ldots, a_n >$
2: $Regles \leftarrow < (a_1, a_1), (a_2, a_2), \ldots, (a_n, a_n) >$
3: $Ctx_{(< 2|B|)} \leftarrow$ `Calculer_Ctx`$_{Alg.v2}(B, |B|)$
4: $u \leftarrow a_1$
5:
6: **Tant que** $u \neq$ `NULL` **faire**
7: **Pour** $i \leftarrow 1$ à n **faire**
8: $v \leftarrow$ `Trouver_Repr`$_{Alg.v2}(ua_i)$
9: **Si** $v = ua_i$ **alors**
10: $Regles \rightarrow ajouter((ua_i, ua_i))$
11: $Repr \rightarrow ajouter(ua_i)$
12: **Sinon**
13: $Regles \rightarrow ajouter((ua_i, v))$
14: $u \leftarrow suiv(u)$
15:
16: **retourner** *Repr*

Fonction 3.5 : `Trouver_Repr`$_{Alg.v2}(u)$

1: *//La fonction* `Trouver_Repr`$_{Alg.v2}$ *est identique à la fonction* `Trouver_Repr`$_{Alg.v1}$
2: **Si** $\exists v \in Repr$ **tel que** $\forall x, y \in Ctx_{(< 2|B|)}$, $eval(xuy) = eval(xvy)$ **alors**
3: **retourner** v *//car v est le représentant de u*
4: **Sinon**
5: **retourner** u *//car u est son propre représentant*

Fonction 3.6 : `Calculer_Ctx`$_{Alg.v2}(B, |B|)$

1: *//La fonction* `Calculer_Ctx`$_{Alg.v2}$ *est identique à la fonction* `Calculer_Ctx`$_{Alg.v1}$
2: $Ctx_{(< 2|B|)} \rightarrow ajouter(\varepsilon)$
3:
4: **Pour** $k \leftarrow 1$ à $2n - 2$ **faire**
5: **Pour chacun** des n^{k-1} derniers mots (u) de la liste $Ctx_{(< 2|B|)}$ **faire**
6: **Pour** $i \leftarrow 1$ à n **faire**
7: $Ctx_{(< 2|B|)} \rightarrow ajouter(ua_i)$
8:
9: **retourner** $Ctx_{(< 2|B|)}$

chaque successeur de représentant, il existe un $v' \in Repr$ tel que $(uv_1, v') \in Regles$. Nous remplaçons alors uv_1 par v' (multiplication de u par v_1). En répétant cette opération k fois, nous multiplions chacune des lettres de v et nous obtenons le résultat de la multiplication de u par v.

Fonction 3.7 : $\texttt{Mult}_{Alg.v2}(u, v)$

1: $X \leftarrow u$
2:
3: **Pour** $i \leftarrow 1$ à $|v|$ **faire**
4: **Soit** $Y \in Repr$ tel que $(Xv_i, Y) \in Regles$
5: $X \leftarrow Y$ *//Y est le représentant de Xv_i (multiplication de X par v_i)*
6: *//$Y = Xv_i$ si Xv_i est un représentant*
7: **retourner** X

3.7.1 Rectitude de l'algorithme

Puisque les modifications apportées à $\texttt{Calculer\_Repr}_{Alg.v1}$ ne touchent pas la liste *Repr*, mais qu'il s'agit simplement de conserver dans la liste *Regles* des informations déjà calculées, les preuves effectuées à la Section 3.6.1 sur la rectitude de l'algorithme original sont encore valables pour cette nouvelle version. Nous savons donc que toutes les classes sont distinctes et que $Repr = D(B)$. Nous devons maintenant prouver le bon fonctionnement de la fonction de multiplication de deux représentants.

Lemme 3.7.1. *Pour tous les successeurs de représentant ra ($\forall r \in Repr, \forall a \in B$), $\exists v \in Repr$ tel que $(ra, v) \in Regles$ et tel que $\forall x, y \in Ctx_{(< 2|B|)}$, $eval(xray) = eval(xvy)$.*

Preuve. Le Lemme 3.6.2 nous indique que tous les successeurs de représentant ont été traités par la fonction $\texttt{Calculer\_Repr}_{Alg.v2}$ et que lors de chaque traitement, nous avons ou bien $v = ua_i$ (ligne 9) et ua_i représente une nouvelle classe ou bien $v \neq ua_i$ (ligne 12) et $\forall x, y \in Ctx_{(< 2|B|)}$, $eval(xua_iy) = eval(xvy)$. Dans les deux cas, la règle (ua_i, v) est ajoutée et celle-ci respecte l'énoncé du présent Lemme. □

Lemme 3.7.2. *Suite à l'exécution de la fonction* $\texttt{Calculer\_Repr}_{Alg.v2}(B, |B|)$, $(Repr, \circ)$, *où* $u \circ v = \texttt{Mult}_{Alg.v2}(u, v)$, *est le monoïde dérivé de* B $\big((Repr, \circ) = (D(B), \cdot)\big)$.

Preuve. Nous savons déjà que l'ensemble *Repr* contiendra les mêmes éléments que dans la version antérieure de l'algorithme et que l'opération $\cdot$ est associative. Nous devons maintenant démontrer que $\forall u, v \in Repr$, $(u \cdot v) = \texttt{Mult}_{Alg.v2}(u, v)$.

Puisque les lettres de la boucle B sont les générateurs du monoïde dérivé, nous pouvons exprimer v sous la forme d'une multiplication de lettres de B. Nous obtenons alors $v = (((v_1 \cdot v_2) \cdot v_3) \cdot \ldots v_k)$ où $v_i \in B \subseteq Repr$. Donc $(u \cdot v) = \big(u \cdot (((v_1 \cdot v_2) \cdot v_3) \cdot \ldots v_k)\big)$. Mais puisque l'opération $\cdot$ est associative, $\big(u \cdot (((v_1 \cdot v_2) \cdot v_3) \cdot \ldots v_k)\big) = \big((((u \cdot v_1) \cdot v_2) \cdot v_3) \cdot \ldots v_k\big)$. Par définition, nous avons :

$$(u \cdot v_1) = [uv_1] = \big(r_1 \in D(B) \mid \forall x, y \in Ctx_{(< 2|B|)}, eval(xuv_1y) = eval(xr_1y)\big)$$

Ainsi que

$$(u \circ v_1) = \texttt{Mult}_{Alg.v2}(u, v_1) = \big(r_2 \in Repr \mid (uv_1, r_2) \in Regles\big)$$

Mais par le Lemme 3.7.1, nous savons que $eval(xuv_1y) = eval(xr_2y)$, $\forall x, y \in Ctx_{(< 2|B|)}$ et puisque $D(B) = Repr$, $r_1 = r_2$. Nous avons donc $(u \cdot v_1) = (u \circ v_1)$. Avec des applications successives de ce principe, nous obtenons $\big((((u \cdot v_1) \cdot v_2) \cdot v_3) \cdot \ldots v_k\big) = \big((((u \circ v_1) \circ v_2) \circ v_3) \circ \ldots v_k\big) = \texttt{Mult}_{Alg.v2}(u, v)$, ce qui conclut la preuve. □

3.7.2 Analyse des temps d'exécution

L'analyse des temps d'exécution des fonctions $\texttt{Calculer\_Ctx}_{Alg.v2}(B, |B|)$ et $\texttt{Trouver\_Repr}_{Alg.v2}(u)$ est la même que celle faite sur leur version précédente $(Alg.v1)$, car ces deux fonctions n'ont pas été modifiées d'une version à l'autre.

Pour la fonction $\texttt{Calculer\_Repr}_{Alg.v2}(B, |B|)$, nous considérerons que l'ajout d'un élément à l'ensemble *Regles* s'effectue en temps constant, tout comme nous avions considéré que l'ajout d'un élément à l'ensemble *Repr* se faisait en temps constant. Puisque la seule différence entre $\texttt{Calculer\_Repr}_{Alg.v1}$ et $\texttt{Calculer\_Repr}_{Alg.v2}$ est l'ajout des règles, le temps d'exécution de cette dernière reste lui aussi inchangé.

3.7.2.1 $\texttt{Mult}_{Alg.v2}(u, v)$

Le temps d'exécution de cette fonction dépendra beaucoup de la structure de donnée utilisée pour représenter l'ensemble *Regles*. Dans le meilleur des cas, nous choisissons une structure utilisant le *hachage* pour la recherche d'éléments et le temps de recherche est logarithmique par rapport à la taille de la clef. Puisque l'algorithme effectue $|v|$ recherches dans *Regles* avec une clef ayant une taille maximale de $|uv|$, le temps d'exécution de la fonction $\texttt{Mult}_{Alg.v2}(u, v)$ est donc dans l'ordre de $O(|v| \cdot \log(|uv|))$.

On remarque que le coût de la multiplication de u par v est beaucoup plus petit en utilisant la fonction $\texttt{Mult}_{Alg.v2}(u, v)$ dont le temps d'exécution est dans l'ordre de $O(|v| \cdot \log(|uv|))$ qu'en utilisant la fonction $\texttt{Trouver\_Repr}_{Alg.v2}(uv)$, dont le temps d'exécution est dans l'ordre de $O\big(|D(B)| \cdot |Ctx_{(< 2|B|)}|^2\big)$.

CHAPITRE 4

ÉTUDE DE SOLUTIONS POUR RÉDUIRE LA QUANTITÉ DE CONTEXTES D'ÉVALUATION UTILISÉS

4.1 Introduction

Au Chapitre 3, nous avons démontré que le monoïde dérivé d'une boucle finie est calculable et nous avons présenté deux versions d'un algorithme (Sections 3.6 et 3.7) permettant de le calculer. Mais nous avons vu, lors de l'analyse du temps d'exécution de ces deux versions (Sections 3.6.2 et 3.7.2), que cela prends un temps plus qu'exponentiel par rapport à l'ordre de la boucle pour calculer tous les éléments du monoïde dérivé (le temps d'exécution de `Calculer_Repr`$_{Alg.v2}$ est dans l'ordre de $O(|D(B)|^2 \cdot |Ctx_{(<2|B|)}|^2)$, où $|Ctx_{(<2|B|)}| = \frac{(|B|-1)^{2|B|}-1}{|B|-2}$).

En pratique, nous sommes intéressés à calculer les monoïdes dérivés non triviaux, c'est-à-dire ceux calculés à partir d'une boucle non associative car, par le Lemme 3.2, $D(B) = B$ si B est une boucle associative. Puisque la plus petite boucle non associative est d'ordre 5 (voir [15]), il est en pratique impossible de calculer son monoïde dérivé à l'aide de l'algorithme original, car nous devons alors effectuer $|Ctx_{(<2|B|)}|^2 = \left(\frac{4^{10}-1}{3}\right)^2 \approx 1,2 \cdot 10^{11}$ comparaisons d'évaluations pour affirmer que deux mots font partie de la même classe de congruence. Cela prendrait donc plusieurs années, voire plusieurs décennies pour calculer son monoïde dérivé.

La cause principale de ce temps exponentiel est le nombre exponentiel d'éléments dans l'ensemble $Ctx_{(<2|B|)}$, soit près de $|B|^{2|B|-1}$ éléments (voir Section 3.5). Pour que

nous puissions calculer le monoïde dérivé d'une boucle, nous devons donc diminuer la quantité de contextes d'évaluation utilisés. Cet objectif n'a pas été entièrement atteint dans cet ouvrage. Nous verrons dans ce chapitre trois méthodes différentes pour réduire la quantité de contextes d'évaluations utilisés, mais nous ne pouvons prouver que le résultat obtenu $\big((Repr, \bullet)$, où $u \bullet v = \texttt{mult}(uv)\big)$ par ces versions de l'algorithme est toujours le monoïde dérivé et ce même lorsque la loi de composition interne $(\bullet)$ est associative. Par contre, lorsque $(Repr, \bullet)$ forme un monoïde, nous démontrerons sur celui-ci plusieurs propriétés similaires à celles du monoïde dérivé. Cela nous amène à affirmer que les idées présentées dans ce chapitre constituent un pas dans la bonne direction.

La solution la plus naturelle pour réduire la quantité de contextes d'évaluations, est de réduire leur taille maximale permise ($Alg.v3$). Par exemple, nous pouvons n'utiliser comme contextes d'évaluations que tous les mots dont la longueur est plus petite ou égale à 2. Nous verrons à la Section 4.2 qu'avec cette technique, nous obtenons un groupoïde $\big((Repr, \bullet)\big)$ et qu'en faisant varier la longueur maximale autorisée selon la boucle utilisée, cela nous permet d'obtenir un monoïde. Dans certains cas particuliers, nous pouvons prouver que ce dernier correspond au monoïde dérivé. Une autre solution consiste à utiliser comme ensemble de contextes d'évaluation l'ensemble $Repr$, contenant tous les représentants trouvés ($Alg.v4$). Comme nous le verrons à la Section 4.3, cela nous permettra généralement d'obtenir un monoïde $\big((Repr, \bullet)\big)$. Nous verrons finalement à la Section 4.4 un algorithme beaucoup plus efficace utilisant une hybridation des deux premières solutions ($Alg.v5$). Empiriquement, peu importe laquelle de ces trois solutions nous avons utilisée, le monoïde obtenu à partir d'une boucle donnée a toujours été le même.

L'Algorithme 4.1 résume grossièrement les particularités de chacune des versions

de l'algorithme présentées dans ce chapitre et dans le chapitre précédent. Nous voyons que pour les trois premières versions, l'ensemble de contextes d'évaluation ne change pas durant le calcul, une seule itération de la boucle sera donc effectuée. Cela a pour conséquence que les règles établies ne sont jamais modifiées. Pour la version $Alg.v4$, nous avons $Ctx_i = Repr$ à chaque début d'itération. Cette version se termine lorsqu'il n'y a pas eu de nouveaux représentants pendant toute une itération. Finalement, l'objectif de la cinquième version est de calculer un monoïde en utilisant le moins de contextes d'évaluation possible. Les représentants sont toujours utilisés comme contextes d'évaluations, mais ils sont ajoutés graduellement et ne seront pas nécessairement tous utilisés. Son exécution se termine lorsqu'un monoïde est trouvé ou lorsque tous les représentants ont été utilisés comme contextes d'évaluation.

Algorithme 4.1 : Algorithme générique

$Ctx_0 =$ ***Ctx_ Initiaux***
$i = 0$

Tant que ***Condition*** **faire**
Calculer tous les représentants possibles en utilisant Ctx_i et les ajouter à $Repr$.
$Ctx_{i+1} = Ctx_i \cup$ ***Nouveaux_ Ctx***
$i++$

//NOTE : On considère que $Ctx_{-1} = \emptyset$

	Ctx_ Initiaux	***Condition***	***Nouveaux_ Ctx***
$Alg.v1$	$\{u \in B^* \mid \ \lvert u\rvert < 2\lvert B\rvert\}$	$i < 1$	$\emptyset$
$Alg.v2$	$\{u \in B^* \mid \ \lvert u\rvert < 2\lvert B\rvert\}$	$i < 1$	$\emptyset$
$Alg.v3$	$\{u \in B^* \mid \ \lvert u\rvert \leq LMC\}$	$i < 1$	$\emptyset$
$Alg.v4$	$Repr = B$	$(Ctx_i \neq Ctx_{i-1})$	$Repr$
$Alg.v5$	$a_1(identité)$	$(\neg$Associatif $\bullet)$ **ET** $(Ctx_i \neq Ctx_{i-1})$	$\{r \in Repr \mid \ \lvert r\rvert = i\}$

Puisque les analyses effectuées sur les algorithmes $Alg.v1$ et $Alg.v2$ nous ont déjà permis de déterminer que la grande lenteur de leurs calculs est due à la trop grande

quantité de contextes d'évaluations utilisés, nous ne ferons pas l'analyse des temps d'exécution des trois versions de l'algorithme présentées dans ce chapitre. Nous effectuerons plutôt quelques comparaisons pratiques.

4.2 Utilisation de contextes d'évaluation de petites tailles ($Alg.v3$)

Nous avons vu qu'une solution naturelle pour diminuer la quantité de contextes d'évaluation utilisés lors du calcul du monoïde dérivé d'une boucle finie, est de n'utiliser que des mots de petites tailles. Nous verrons dans cette section une nouvelle version de l'algorithme, élaborée à partir de la deuxième version ($Alg.v2$, Section 3.7), implantant cette idée. On se souvient qu'il est possible de représenter une classe de congruence du monoïde dérivé comme une table d'évaluation où les lignes représentent les contextes gauches d'évaluation et les colonnes les contextes droits d'évaluation. Nous retrouvons alors à la jonction de la ligne x et de la colonne y l'ensemble d'évaluation de xry où r est un mot faisant partie de la classe de congruence concernée. La Figure 4.1 permet de visualiser la différence entre le calcul d'une classe selon la deuxième et la troisième version de l'algorithme ($Alg.v2$ et $Alg.v3$) en utilisant cette notation.

Il est important de noter que ce nouvel algorithme ne fournit pas toujours un monoïde pour une longueur maximale de contextes d'évaluations donnée, il s'agit alors d'augmenter cette valeur jusqu'à l'obtention d'un monoïde (au maximum la longueur sera de $2|B|-1$). Puisque nous ne sommes plus certains de toujours obtenir un monoïde à la suite de l'exécution de l'algorithme, nous aurons besoin d'une nouvelle fonction permettant de tester l'associativité du groupoïde obtenu, dénoté $(Repr, \bullet)$, où $u \bullet v = \texttt{Mult}_{Alg.v3}(u, v)$. Dans le cas où un monoïde est obtenu, nous étudierons plus en détails ses propriétés à la Section 4.2.1.

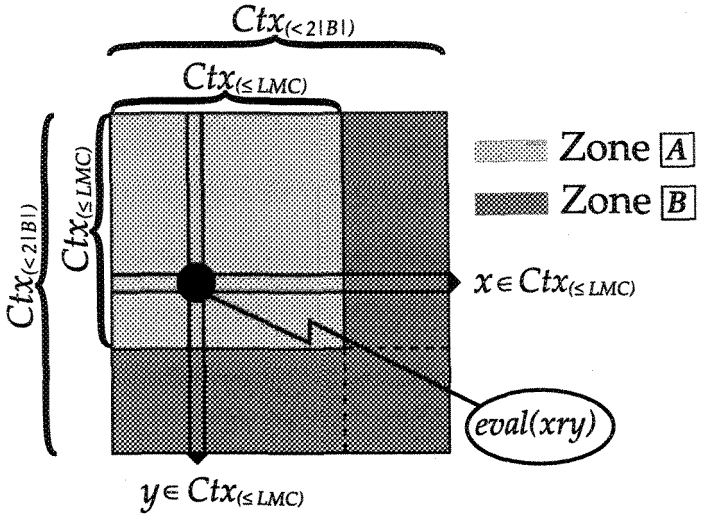

Figure 4.1: Différence entre la figuration d'une classe dont le représentant est r selon l'algorithme $Alg.v3$ (Zone A) et l'algorithme $Alg.v2$ (Zone A et B).

Dans les versions antérieures de l'algorithme, nous savions que $\forall r \in Repr$, $[r] = r$. Autrement dit, nous savions que chaque élément faisant partie de l'ensemble *Repr* était effectivement le plus petit mot de sa classe de congruence. Dans cette nouvelle version, cela est vrai uniquement lorsque $|Repr| = |D(B)|$, ce qui n'est pas toujours le cas. En général, bien que les éléments de l'ensemble *Repr* représentent toujours des classes d'équivalences, nous ne pouvons démontrer qu'ils sont toujours les plus petits mots de leur classe. Un mot r sera plutôt inclus dans l'ensemble *Repr* s'il est le premier de sa classe d'équivalence à être calculé par l'algorithme $Alg.v3$. Pour des raisons pratiques, nous utiliserons tout de même la dénomination "représentant" pour parler des éléments de l'ensemble *Repr*.

Voyons maintenant en détail la description de chacune des fonctions de cette nouvelle version de l'algorithme. Notons que dans cette troisième version, la fonction $\texttt{Mult}_{Alg.v3}(u, v)$ (Fonction 4.4) n'a pas été modifiée par rapport à sa dernière version. Pour ce qui est de la fonction $\texttt{Trouver\_Repr}_{Alg.v3}(u)$ (Fonction 4.2), elle effectue la

même tâche que sa version précédente, mais en utilisant l'ensemble $Ctx_{(\leq LMC)}$ plutôt que l'ensemble $Ctx_{(< 2|B|)}$ comme contextes d'évaluations.

La Fonction 4.5 présente $\texttt{Verifier\_Asso}_{Alg.v3}(B, Repr)$ qui permet de tester l'associativité du groupoïde obtenu $((Repr, \bullet))$. Nous utilisons un test d'associativité qui a été suggéré par Dr. F. W. Light en 1949, technique que l'on retrouve dans [13]. Il est dit dans ce test que si $(G, \diamond)$ est un groupoïde généré par B, alors il est suffisant de vérifier que $((x \diamond a) \diamond y) = (x \diamond (a \diamond y))$, $\forall x, y \in G$ et $\forall a \in B$ pour démontrer l'associativité de S. Dans notre cas, $(Repr, \bullet)$ est un groupoïde généré par la boucle B. La fonction $\texttt{Verifier\_Asso}_{Alg.v3}$ vérifie donc l'associativité de $(Repr, \bullet)$ selon la méthode de Light, ce qui nous permet de passer d'un temps cubique par rapport à $|Repr|$ à un temps quadratique (si on considère que $|B|$ est négligeable par rapport à $|Repr|$).

Les fonctions $\texttt{Calculer\_Repr}_{Alg.v3}(B, |B|, LMC)$ (Fonction 4.1) et $\texttt{Calculer\_Ctx}_{Alg.v3}(B, |B|, LMC)$ (Fonction 4.3) ont un nouveau paramètre d'entrée, soit LMC. Il indique la longueur maximale des contextes d'évaluation contenus dans l'ensemble $Ctx_{(\leq LMC)}$. Nous savons que LMC est nécessairement plus petit que $2|B|$ puisque que nous avons vu qu'il est inutile d'utiliser des contextes d'évaluation plus grands que $2|B| - 1$ (voir Section 3.4). Remarquons que si $LMC = 2|B| - 1$, cela équivaut à l'algorithme $Alg.v2$. Ainsi, à la fin de l'exécution de la fonction $\texttt{Calculer\_Ctx}_{Alg.v3}$, nous avons $Ctx_{(\leq LMC)} = \{w \in B^* \mid |w| \leq LMC\}$.

Les changements apportés à la fonction $\texttt{Calculer\_Repr}_{Alg.v3}(B, n, LMC)$ ne concernent pas la boucle "Tant que" servant à calculer l'ensemble $Repr$. Ils concernent dans un premier temps l'ensemble des contextes d'évaluations utilisés, qui contiendra maintenant tous les mots de longueur plus petite ou égale à LMC. En second lieu, la fonction retournera l'ensemble $Repr$ seulement si $(Repr, \bullet)$ est associatif, sinon cela signifie qu'elle n'a pas été en mesure de calculer un monoïde à l'aide des contextes d'évaluation

de longueur plus petite ou égale à LMC et elle retournera alors `NULL`.

Fonction 4.1 : $\texttt{Calculer\_Repr}_{Alg.v3}(B, |B|, LMC)$

```
1: Repr ←< a_1, a_2, ..., a_n >
2: Regles ←< (a_1, a_1), (a_2, a_2), ..., (a_n, a_n) >
3: Ctx_(≤ LMC) ← Calculer_Ctx_Alg.v3 (B, |B|, LMC)
4: u ← a_1
5:
6: Tant que u ≠ NULL faire
7:    Pour i ← 1 à n faire
8:       v ← Trouver_Repr_Alg.v3(ua_i)
9:       Si v = ua_i alors
10:         Regles → ajouter((ua_i, ua_i))
11:         Repr → ajouter(ua_i)
12:      Sinon
13:         Regles → ajouter((ua_i, v))
14:   u ← suiv(u)
15:
16: Si Verifier_Asso_Alg.v3(B, Repr) = VRAI alors
17:    retourner Repr
18: Sinon
19:    retourner NULL
```

Fonction 4.2 : $\texttt{Trouver\_Repr}_{Alg.v3}(u)$

```
Si ∃v ∈ Repr tel que ∀x, y ∈ Ctx_(≤ LMC),
   eval(xuy) = eval(xvy) alors
   retourner v        //car v est le représentant de u
Sinon
   retourner u        //car u est son propre représentant
```

En pratique, le temps d'exécution de cette version de l'algorithme est généralement acceptable lorsque le paramètre LMC est plus petit que 5. À partir de 5, le nombre de contextes d'évaluation augmente très rapidement et le temps d'exécution devient trop long. Lorsque LMC est égal à 7 ou plus, et ce indépendamment de l'ordre de la boucle, le nombre de contextes d'évaluation est beaucoup trop grand et le calcul d'un monoïde en utilisant cette version devient alors trop onéreux.

Fonction 4.3 : $\texttt{Calculer\_Ctx}_{Alg.v3}(B, |B|, LMC)$

1: $Ctx_{(\leq LMC)} \rightarrow ajouter(\varepsilon)$
2: $k \leftarrow 1$
3:
4: **Tant que** $(k < LMC)$ **ET** $(k < 2n-2)$ **faire**
5: **Pour chacun** des n^{k-1} derniers mots (u) de la liste $Ctx_{(\leq LMC)}$ **faire**
6: **Pour** $i \leftarrow 1$ à n **faire**
7: $Ctx_{(\leq LMC)} \rightarrow ajouter(ua_i)$
8:
9: **retourner** $Ctx_{(\leq LMC)}$

Fonction 4.4 : $\texttt{Mult}_{Alg.v3}(u, v)$

1: *//La fonction* $\texttt{Mult}_{Alg.v3}$ *est identique à la fonction* $\texttt{Mult}_{Alg.v2}$
2: $X \leftarrow u$
3:
4: **Pour** $i \leftarrow 1$ à $|v|$ **faire**
5: **Soit** $Y \in Repr$ tel que $(Xv_i, Y) \in Regles$
6: $X \leftarrow Y$ *//*Y *est le représentant de* Xv_i *(multiplication de* X *par* v_i*)*
7: *//*$Y = Xv_i$ *si* Xv_i *est un représentant*
8: **retourner** X

Fonction 4.5 : $\texttt{Verifier\_Asso}_{Alg.v3}(B, Repr)$

1: **Si** $\exists x, y \in Repr$ **ET** $\exists a \in B$ **tel que**
$\left(\texttt{Mult}_{Alg.v3}(\texttt{Mult}_{Alg.v3}(x, a),\ y)\right) \neq \left(\texttt{Mult}_{Alg.v3}(x,\ \texttt{Mult}_{Alg.v3}(a, y))\right)$ **alors**
2: **retourner** FAUX
3: **Sinon**
4: **retourner** VRAI

4.2.1 Propriétés du monoïde obtenu

Sans perte de généralité, nous pouvons considérer que les Lemmes 3.6.1 et 3.7.1 sont toujours valides pour cette version de l'algorithme. En effet, la méthode de calcul des représentants (boucle "Tant que" de la fonction $\texttt{Calculer\_Repr}_{Alg.v3}$) n'y est pas modifiée, seul le nombre de contextes d'évaluation utilisés est différent et cela n'influence pas les preuves de ces deux lemmes. Le premier lemme est un invariant de la boucle principale de $\texttt{Calculer\_Repr}_{Alg.v3}$, indiquant que les représentants ont tous des tables d'évaluation différentes, i.e. $\forall r, s \in Repr\ (r \neq s)$, $\exists x, y \in Ctx_{(\leq LMC)}$ tel que $eval(xry) \neq eval(xsy)$. Le second mentionne qu'il existe une règle pour chaque successeur de représentant, i.e. $\forall r \in Repr$, $\forall a \in B$, $\exists v \in Repr$ tel que $(ra, v) \in Regles$ et tel que $\forall x, y \in Ctx_{(\leq LMC)}$, $eval(xray) = eval(xvy)$.

D'autres propriétés du monoïde obtenu par l'algorithme $Alg.v3$ sont démontrées dans les énoncés suivants. En particulier, sachant que $(Repr, \bullet)$ forme un monoïde, le Lemme 4.2.5 démontre qu'il est suffisant que $\forall w \in B^*$, $eval(w) = eval(w_1 \bullet w_2 \cdots \bullet w_k)$, où $w = w_1 w_2 \ldots w_k$ $(w_i \in B)$, pour démontrer que tout monoïde obtenu par l'algorithme est effectivement le monoïde dérivé. Bien que nous sommes capable de prouver que $\forall w \in B^*$, $eval(w) \subseteq eval(w_1 \bullet w_2 \cdots \bullet w_k)$, il est important de noter qu'il nous est présentement impossible de démontrer que $\forall w \in B^*$, $eval(w) \supseteq eval(w_1 \bullet w_2 \cdots \bullet w_k)$. La finalisation de cette preuve, jumelée au Lemme 4.2.5, permettrait donc de démontrer immédiatement la rectitude de cette troisième version de l'algorithme. En attendant, la Remarque 4.2.1, le Lemme 4.2.2 et le Théorème 4.2.3 présentent les conditions suffisantes où nous savons que le monoïde calculé est effectivement le monoïde dérivé $\big((Repr, \bullet) = (D(B), \cdot)\big)$.

Remarque 4.2.1. *Si $LMC = 2|B| - 1$, le monoïde calculé par la fonction* $\texttt{Calculer\_Repr}_{Alg.v3}(B, |B|, LMC)$ *est le même que celui calculé par l'algorithme $Alg.v2$, c'est-à-dire le monoïde dérivé.*

Lemme 4.2.2. *Si $LMC = 0$ et si suite à l'exécution de la fonction* `Calculer_Repr`$_{Alg.v3}(B, |B|, LMC)$ $(Repr, \bullet)$ *est associatif, alors la boucle $(B, \circ)$ est associative et $(Repr, \bullet) = (B, \circ) = (D(B), \cdot)$.*

Preuve. Puisque $LMC = 0$, le seul contexte d'évaluation qui est utilisé est le mot vide. Soient $a, b \in B \subseteq Repr$, alors $eval(\varepsilon ab\varepsilon) = eval(ab) = a \circ b$. Considérant que l'algorithme traite les successeurs de représentant par ordre croissant de longueur et que tous ceux de longueur 2 ne seront pas des représentants ($[ab] = a \circ b$), l'algorithme ne traitera pas les mots de longueur 3 et $(Repr, \bullet) = (B, \circ)$. Puisque $(B, \circ)$ est associatif, par le Lemme 3.2, $(Repr, \bullet)$ est le monoïde dérivé de B. □

Théorème 4.2.3. *Soient $(B, \circ)$ une boucle finie et $N \subseteq B$ une sous-boucle normale de B. Si B/N est associatif et si $LMC = 2|N| - 1$, alors suite à l'exécution de la fonction* `Calculer_Repr`$_{Alg.v3}(B, |B|, LMC)$, $(Repr, \bullet)$ *est le monoïde dérivé de B $\big((Repr, \bullet) = (D(B), \cdot)\big)$.*

Preuve. L'énoncé du théorème signifie en fait que la longueur des contextes d'évaluation nécessaire pour calculer le monoïde dérivé de B est bornée supérieurement, dans ce cas précis, par $2|N| - 1$. En effet, grâce au Lemme 2.4.6, nous savons qu'un arbre binaire T sur B ayant une frontière $w \in B^*$, pourra prendre au maximum $|N|$ valeurs différentes si on en change la structure interne (parenthétisation). L'utilisation du principe du trou de pigeon dans la preuve du Lemme 3.4.2 ne nécessite alors que $|N|$ arbres binaires différents ($k \geq |N|$). Nous obtenons le résultat voulu en transposant ce fait dans les énoncés et les preuves concernés (Lemmes 3.4.3, 3.4.6 et 3.4.7, la Remarque 3.4.4 et le Théorème 3.4.9). □

Si l'ordre de la boucle est un nombre premier, l'utilisation du théorème précédent est inutile puisque pour qu'une sous-boucle N soit normale, $|N|$ doit diviser $|B|$ (voir

[11]). En pratique, l'utilisation de ce théorème nous permet de calculer le monoïde dérivé de plusieurs boucles d'ordre 6 et 8. Plus précisément nous avons ainsi calculer le monoïde dérivé de 14 boucles d'ordre 6 et de 139 boucles d'ordre 8 (voir Chapitre 5).

Le lemme suivant démontre que le monoïde calculé par l'algorithme $Alg.v3$ ne peut pas contenir plus de classes d'équivalences que le monoïde dérivé. Il nous sera notamment utile dans la preuve du Lemme 4.2.5.

Lemme 4.2.4. *Suite à l'exécution de la fonction* $\texttt{Calculer\_Repr}_{Alg.v3}(B, |B|, LMC)$, *on a* $|Repr| \leq |D(B)|$.

Preuve. Nous savons par le Lemme 3.6.1, que $\forall r, s \in Repr\ (r \neq s), \exists x, y \in Ctx_{(\leq LMC)}$ tel que $eval(xry) \neq eval(xsy)$. Mais puisque $Ctx_{(\leq LMC)} \subset B^*$, $[r] \neq [s]$ et $D(B)$ contient au minimum le même nombre d'éléments que $Repr$. □

Soit $w \in B^*$ tel que $w = w_1 w_2 \dots w_n$ $(w_i \in B)$, lorsque $(Repr, \bullet)$ forme un monoïde, nous définissons $[\![w]\!]$ comme le représentant de w dans $(Repr, \bullet)$, c'est-à-dire $w_1 \bullet w_2 \cdots \bullet w_n$. Puisque la loi de composition interne $(\bullet)$ est associative, la parenthétisation importe peu. L'objectif de cette notation est simplement d'alléger le lemme suivant qui dit que $eval(w) = eval([\![w]\!])$, $\forall w \in B^*$, est une condition suffisante pour démontrer que tout monoïde calculé par cette version de l'algorithme est effectivement le monoïde dérivé de B.

Lemme 4.2.5. *Suite à l'exécution de la fonction* $\texttt{Calculer\_Repr}_{Alg.v3}(B, |B|, LMC)$, *si* $(Repr, \bullet)$ *est associatif et si* $\forall w \in B^*$, $eval(w) = eval([\![w]\!])$, *alors* $(Repr, \bullet)$ *est un monoïde isomorphe au monoïde dérivé de* B $\big((Repr, \bullet) \cong (D(B), \cdot)\big)$.

Preuve. Soient $u, v \in B^*$, nous allons démontrer que si $[\![u]\!] = [\![v]\!]$, alors $[u] = [v]$. Pour tout $x, y \in B^*$, nous avons par associativité $[\![xuy]\!] = [\![x]\!] \bullet [\![u]\!] \bullet [\![y]\!]$. Mais puisque $[\![u]\!] =$

$[\![v]\!]$, nous avons également $[\![xuy]\!] = [\![x]\!] \bullet [\![u]\!] \bullet [\![y]\!] = [\![x]\!] \bullet [\![v]\!] \bullet [\![y]\!] = [\![xvy]\!]$. En utilisant la propriété mentionnée dans l'énoncé du lemme ($\forall w \in B^*$, $eval(w) = eval([\![w]\!])$), nous avons $eval(xuy) = eval([\![xuy]\!]) = eval([\![xvy]\!]) = eval(xvy)$. Comme cela est vrai $\forall x, y \in B^*$, $[u] = [v]$ par définition du monoïde dérivé. Cela signifie que $(D(B), \cdot)$ est une union de classes de congruences de $(Repr, \bullet)$, donc que $|D(B)| \leq |Repr|$. En jumelant ce fait et le Lemme 4.2.4 ($|Repr| \leq |D(B)|$), nous obtenons que $|Repr| = |D(B)|$.

Soit l'application $\varphi : Repr \rightarrow D(B)$, définit par $\varphi(r) = [r]$. Nous allons démontrer que φ est un isomorphisme de $(Repr, \bullet)$ dans $(D(B), \cdot)$, c'est-à-dire que $\forall r, s \in Repr$, $\varphi(r \bullet s) = \varphi(r) \cdot \varphi(s)$.

$$\varphi(r \bullet s) = \varphi(r) \cdot \varphi(s) \iff [r \bullet s] = [r] \cdot [s] \tag{4.1}$$

$$\iff [r \bullet s] = [rs] \tag{4.2}$$

$$\iff \forall x, y \in B^*,\ eval([\![xrsy]\!]) = eval(xrsy) \tag{4.3}$$

Puisque nous savons que $\forall w \in B^*$, $eval(w) = eval([\![w]\!])$ et que $xrsy \in B^*$, l'équation (4.3) est vrai, ce qui conclut la preuve. □

Parmi tous les tests pratiques effectués, il n'a jamais été possible d'obtenir deux monoïdes différents pour une même boucle. Ainsi, les mots de longueurs 2 et moins sont suffisants pour calculer un monoïde à partir d'une boucle d'ordre 5, mais le même monoïde est obtenu si l'on utilise comme contextes d'évaluation tous les mots plus petits que 3, 4 ou 5 (voir Chapitre 5).

4.3 Utilisation des représentants pour l'ensemble Ctx ($Alg.v4$)

Une deuxième solution pour réduire la quantité de contextes d'évaluations utilisés, est d'utiliser uniquement les éléments de l'ensemble $Repr$. Cette idée est principalement

motivée par le lemme et la remarque qui suivent. Notons que pour la quatrième version de l'algorithme, comme pour $Alg.v3$, un élément $r \in Repr$ n'est pas nécessairement le plus petit élément de sa classe d'équivalence, mais bien le premier mot de sa classe d'équivalence à être calculé par l'algorithme.

Lemme 4.3.1. *Si nous avons l'ensemble des représentants du monoïde dérivé, il n'est nécessaire que d'utiliser ceux-ci comme contextes d'évaluation pour déterminer si deux mots font partie de la même classe d'équivalence (le représentant du mot vide étant l'identité).*

Preuve. Par définition du monoïde dérivé, si $x, y \in B^*$ et si $u \in B^+$, alors

$$eval(xuy) = eval([x]uy) = eval([x]u[y])$$

□

Remarque 4.3.2. *Dans le lemme précédent, x peut être remplacé par n'importe quel mot $x' \in B^*$ tel que $[x'] = [x]$. En effet, selon la définition du monoïde dérivé, si $u, y \in B^*$, alors $\forall x, x' \in B^*$ tel que $[x'] = [x]$, $eval(xuy) = eval(x'uy)$. Nous pouvons utiliser le même argument pour remplacer le contexte droit d'évaluation (y).*

Évidemment, pendant le calcul du monoïde dérivé nous ne possédons pas la liste complète des représentants, mais bien une liste partielle ($Repr$). Cela amène une conséquence importante. En effet, lorsque des nouveaux représentants sont ajoutés à la liste $Repr$, nous devons effectuer un *retour en arrière* pour vérifier que les règles déjà produites sont toujours cohérentes en utilisant ces nouveaux représentants comme contextes d'évaluation. Par exemple, soit l'ensemble $Repr_{(0...k-2)}$ contenant tous les représentants

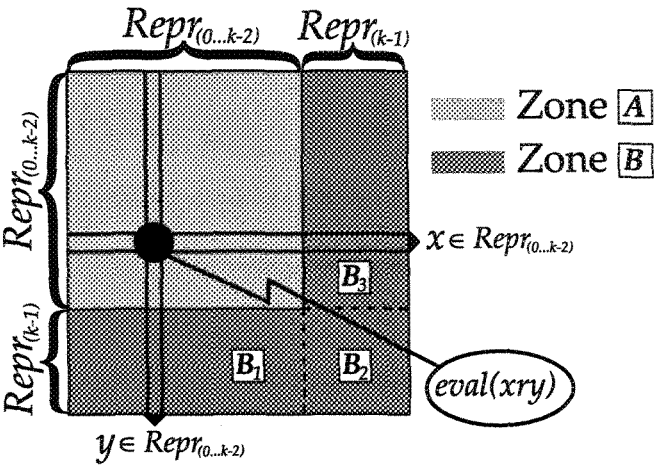

Figure 4.2: Table des évaluations d'un mot r selon l'itération k. La Zone $\boxed{A}$ représente les évaluations déjà testées et la Zone $\boxed{B}$ celles à tester.

qui ont déjà été utilisés comme contextes d'évaluations et soit l'ensemble $Repr_{(k-1)}$ contenant ceux qui ne l'ont pas encore été. Nous savons donc que, utilisant la même notation que pour la Figure 4.1, $\forall(u,v) \in Regles$, les évaluations contenues dans la Zone $\boxed{A}$ de la Figure 4.2 sont identiques pour u et pour v ($\forall x, y \in Repr_{(0...k-2)}$, $eval(xuy) = eval(xvy)$). Le *retour en arrière* consiste alors à vérifier que les évaluations contenues dans la Zone $\boxed{B}$ de cette même figure sont également identiques pour u et v. Si une des évaluations est différente, nous devrons retirer la règle (u,v) de l'ensemble $Regles$ et reclasser u.

La quatrième version de `Calculer_Repr`$(B, |B|)$ (Fonction 4.6) fonctionne de manière itérative et effectue deux tâches principales pour chaque itération k. D'abord, elle appelle la fonction `Verifier_Validite_Regles`$_{Alg.v4}(Regles,\ Repr)$ (Fonction 4.8) qui effectue le *retour en arrière* pour s'assurer que les règles sont cohérentes si les représentants calculés à l'itération $k-1$ sont utilisés comme contextes d'évaluation. Puis, elle appelle la fonction `Classer_Nouveaux_Ra`$_{Alg.v4}(Regles,\ Repr)$ (Fonction 4.10) qui calcule et classe tous les successeurs de représentants formés à partir des représentants

trouvés à l'itération $k-1$. La fonction $\texttt{Calculer\_Repr}_{Alg.v4}$ arrête la recherche de nouveaux représentants lorsqu'il y a eu une itération complète sans trouver de nouveaux représentants (voir Corollaire 3.2.7). Elle vérifie alors, comme pour la troisième version, l'associativité de $(Repr, \bullet)$, où $u \bullet v = \texttt{mult}_{Alg.v4}(u, v)$, et elle retourne `NULL` si ce n'est pas associatif.

Pour que la gestion des représentants soit plus facile, nous avons divisé l'ensemble $Repr$ en trois sous-ensembles distincts, séparant les représentants selon le numéro de l'itération à laquelle ils ont été calculés. Ainsi, $Repr_{(0...k-2)}$ contiendra tous les représentants qui ont été calculés avant ou à l'itération $k-2$, $Repr_{(k-1)}$ tous ceux calculés à l'itération précédente $(k-1)$ et $Repr_{(k)}$ tous ceux calculés à l'itération en cours. Mentionnons que $Repr = (Repr_{(0...k-2)} \cup Repr_{(k-1)} \cup Repr_{(k)})$. Puisque tous les représentants sont utilisés comme contextes d'évaluation, l'utilisation de l'ensemble Ctx est devenu inutile. Les ensembles $Repr_{(0...k-2)}$, $Repr_{(k-1)}$ et $Repr_{(k)}$ seront donc également utilisés comme ensembles de contextes d'évaluation.

Nous avons vu que lors du *retour en arrière*, si, pour une règle (u, v), nous trouvons deux contextes d'évaluation x, y tel que $eval(xuy) \neq eval(xvy)$, alors nous savons que v n'est pas le représentant de u et nous devons retirer la règle (u, v) et reclasser le mot u. C'est la fonction $\texttt{Corriger\_Regle}_{Alg.v4}((u, v),\ Liste\_retires,\ Regles,\ Repr)$ (Fonction 4.9) qui effectue ce travail. Le nouveau représentant v' de u (il est possible que v' soit égal à u si u est un nouveau représentant) ne peut être qu'un élément dont la règle (v', v) existait à l'itération précédente, puisque nous savons que $\forall x, y \in Repr_{(0...k-2)}$, $eval(xv'y) = eval(xuy) = eval(xvy)$ (Zone $\boxed{A}$, Figure 4.2). Pour tout $v \in Repr$, la liste $Liste\_retires \rightarrow liste[v]$ contient l'ensemble des représentants qui ont été dissociés de v à l'itération présente (k). Ainsi, lorsqu'une règle (v', v) est retirée de l'ensemble $Regles$ et conduit à l'ajout du représentant v', v' est ajouté à la liste

$Liste\_retires \rightarrow liste[v]$. L'utilisation de cette liste permet à la fonction `Corriger_Regle`$_{Alg.v4}$ d'éviter de faire des calculs inutiles lorsqu'une règle (u, v) est retirée de l'ensemble *Regles* puisque nous savons alors que le nouveau représentant de u fait partie de l'ensemble $\big(Liste\_retires \rightarrow liste[v] \cup \{u\}\big)$.

La fonction `Trouver_Repr`$_{Alg.v4}(u)$ (Fonction 4.11) effectue la même tâche qu'à la version précédente, c'est-à-dire donner le représentant de u, mais utilise maintenant $(Repr_{(0...k-2)} \cup Repr_{(k-1)})$ comme ensemble de contextes d'évaluation. Les éléments de l'ensemble $Repr_{(k)}$ ne sont pas utilisés comme contextes d'évaluation car nous ajoutons plusieurs éléments à cet ensemble au cours de l'itération k. Cela simplifie donc grandement l'exécution du *retour en arrière* en évitant de répéter des calculs inutilement.

La fonction `Changer_Iteration`$_{Alg.v4}(Repr,\ Ctx,\ k)$ (Fonction 4.7) est une nouvelle fonction servant à transférer les représentants dans les bons ensembles lors du changement d'itération. Notons finalement que les fonctions `Verifier_Asso`$_{Alg.v4}(B, Repr)$ et `Mult`$_{Alg.v4}(u, v)$ (respectivement Fonctions 4.13 et 4.12) sont demeurées identiques à leur version dans l'algorithme $Alg.v3$.

4.3.1 Propriétés du monoïde obtenu

Puisque l'algorithme $Alg.v4$ n'est pas une modification des versions antérieures mais bien un algorithme totalement nouveau, nous ne pourrons pas utiliser directement les preuves effectuées sur les versions précédentes. Notons toutefois que les deux invariants de la boucles "Tant que" de `Calculer_Repr` sont toujours effectifs, ce qui est démontré par les Lemmes 4.3.3 et 4.3.4. L'utilisation de ces deux lemmes permet de démontrer, d'une façon similaire à la preuve du Lemme 4.2.4 de $Alg.v3$, que suite à l'exécution de la fonction `Calculer_Repr`$_{Alg.v4}(B, |B|)$, $|Repr| \leq |D(B)|$. Un lemme similaire au Lemme 4.2.5 (si $(Repr, \bullet)$ est associatif et si $\forall w \in B^*$, $eval(w) = eval([\![w]\!])$, alors $(Repr, \bullet)$ est un monoïde isomorphe au monoïde dérivé) pourrait également être

Fonction 4.6 : $\texttt{Calculer\_Repr}_{Alg.v4}(B, |B|)$

1: $k \leftarrow 1$
2: $Repr_{(0...k-2)} \leftarrow <\ >$
3: $Repr_{(k-1)} \leftarrow < a_1, a_2, \ldots, a_n >$
4: $Repr_{(k)} \leftarrow <\ >$
5: $Regles \leftarrow < (a_1, a_1), (a_2, a_2), \ldots, (a_n, a_n) >$
6:
7: **Tant que** $|Repr_{(k-1)}| > 0$ **faire**
8: **Si** $k > 1$ **alors**
9: $\texttt{Verifier\_Validite\_Regles}_{Alg.v4}(Regles,\ Repr)$
10: $\texttt{Classer\_Nouveaux\_Ra}_{Alg.v4}(B,\ Regles,\ Repr)$
11: $\texttt{Changer\_Iteration}_{Alg.v4}(Repr, k)$
12:
13: **Si** $\texttt{Verifier\_Asso}_{Alg.v4}(B, Repr) = \texttt{VRAI}$ **alors**
14: **retourner** $Repr$
15: **Sinon**
16: **retourner** NULL

Fonction 4.7 : $\texttt{Changer\_Iteration}_{Alg.v4}(Repr,\ k)$

1: $Repr_{(0...k-2)} \rightarrow ajouter(Repr_{(k-1)})$
2: $Repr_{(k-1)} \leftarrow Repr_{(k)}$
3: $Repr_{(k)} \leftarrow <\ >$
4:
5: $k++$

Fonction 4.8 : $\texttt{Verifier\_Validite\_Regles}_{Alg.v4}(Regles,\ Repr)$

1: $Liste\_retires = <\ >$
2: **Pour tout** $(u, v) \in Regles$ **faire**
3: **Si** $u \neq v$ **alors**
4: **Pour tout** $x \in Repr_{(k-1)}$ **faire**
5: **Pour tout** $z \in Repr_{(k-1)}$ **faire**
6: **Si** $eval(xuz) \neq eval(xvz)$ **alors**
7: $\texttt{Corriger\_Regle}_{Alg.v4}\big((u, v), Liste\_retires,\ Regles,\ Repr\big)$
8: **Pour tout** $y \in Repr_{(0...k-2)}$ **faire**
9: **Si** $eval(xuy) \neq eval(xvy)$ OU **si** $eval(yux) \neq eval(yvx)$ **alors**
10: $\texttt{Corriger\_Regle}_{Alg.v4}\big((u, v), Liste\_retires,\ Regles,\ Repr\big)$

Fonction 4.9 : `Corriger_Regle`$_{Alg.v4}((u,v),\ Liste\_retires,\ Regles,\ Repr)$

1: $Regles \rightarrow retirer((u,v))$
2: $Liste\_Repr\_Possibles \leftarrow Liste\_retires \rightarrow liste[v]$
3: $r \leftarrow Liste\_Repr\_Possibles \rightarrow Premier\_element$
4:
5: **Tant que** $|Liste\_Repr\_Possibles| > 0$ et que $r \neq$ `NULL` **faire**
6: **Pour tout** $x \in Repr_{(k-1)}$ **faire**
7: **Pour tout** $z \in Repr_{(k-1)}$ **faire**
8: **Si** $eval(xuz) \neq eval(xrz)$ **alors**
9: $Liste\_Repr\_Possibles \rightarrow retirer(r)$
10: **Pour tout** $y \in Repr_{(0...k-2)}$ **faire**
11: **Si** $eval(xuy) \neq eval(xry)$ OU **si** $eval(yux) \neq eval(yrx)$ **alors**
12: $Liste\_Repr\_Possibles \rightarrow retirer(r)$
13: $r++$
14: **Si** $|Liste\_Repr\_Possibles| == 1$ **alors**
15: Soit $r \in Liste\_Repr\_Possibles$
16: $Regles \rightarrow ajouter((u,r))$
17: **Sinon**
18: $Repr_{(k)} \rightarrow ajouter(u)$
19: $Regles \rightarrow ajouter((u,u))$
20: $Liste\_retires \rightarrow liste[v] \rightarrow ajouter(u)$

Fonction 4.10 : `Classer_Nouveaux_Ra`$_{Alg.v4}(B, Regles,\ Repr)$

1: **Pour tout** $u \in Repr_{(k-1)}$ **faire**
2: **Pour** $i \leftarrow 1$ à $|B|$ **faire**
3: $v \leftarrow$ `Trouver_Repr`$_{Alg.v4}(ua_i,\ Repr)$
4: **Si** $v = ua_i$ **alors**
5: $Repr_{(k)} \rightarrow ajouter(ua_i)$
6: $Regles \rightarrow ajouter((ua_i, ua_i))$
7: **Sinon**
8: $Regles \rightarrow ajouter((ua_i, v))$

Fonction 4.11 : `Trouver_Repr`$_{Alg.v4}(u,\ Repr)$

1: **Si** $\exists v \in Repr$ **tel que** $\forall x, y \in (Repr_{(0...k-2)} \cup Repr_{(k-1)})$, $eval(xuy) = eval(xvy)$ **alors**
2: **retourner** v *//car v est le représentant actuel de u*
3: **Sinon**
4: **retourner** u *//car u est son propre représentant*

Fonction 4.12 : $\texttt{Mult}_{Alg.v4}(u, v)$

1: *//La fonction* $\texttt{Mult}_{Alg.v4}$ *est identique à la fonction* $\texttt{Mult}_{Alg.v3}$
2: $X \leftarrow u$
3:
4: **Pour** $i \leftarrow 1$ à $|v|$ **faire**
5: **Soit** $Y \in Repr$ tel que $(Xv_i, Y) \in Regles$
6: $X \leftarrow Y$ *//*Y *est le représentant de* Xv_i *(multiplication de* X *par* v_i*)*
7: *//*$Y = Xv_i$ *si* Xv_i *est un représentant*
8: **retourner** X

Fonction 4.13 : $\texttt{Verifier\_Asso}_{Alg.v4}(B, Repr)$

1: *//La fonction* $\texttt{Verifier\_Asso}_{Alg.v4}$ *est identique à la fonction* $\texttt{Verifier\_Asso}_{Alg.v3}$
2: **Si** $\exists x, y \in Repr$ **ET** $\exists a \in B$ **tel que**
$\left(\texttt{Mult}_{Alg.v4}(\texttt{Mult}_{Alg.v4}(x, a),\ y)\right) \neq \left(\texttt{Mult}_{Alg.v4}(x,\ \texttt{Mult}_{Alg.v4}(a, y))\right)$ **alors**
3: **retourner** FAUX
4: **Sinon**
5: **retourner** VRAI

démontré.

Pour tous les tests pratiques effectués jusqu'à maintenant (voir Chapitre 5), nous avons remarqué que cet algorithme donne toujours le même monoïde que l'algorithme $Alg.v3$ et ce, même pour les cas où il est prouvé que le monoïde obtenu est le monoïde dérivé de la boucle. Il est également important de noter qu'en pratique, bien qu'il nous soit impossible de le prouver au niveau théorique, l'algorithme a toujours fourni un monoïde à la fin de son exécution. Ainsi, parmi les quelques tests effectués sur les boucles d'ordre 5,6,7 et 8, il n'est jamais arrivé que le résultat soit non associatif.

Lorsque $|Ctx_{(\leq LMC)}|$ est plus petit que $|Repr|$, l'algorithme $Alg.v3$ sera plus rapide que l'algorithme $Alg.v4$. En effet, il faut environ 4 heures en moyenne pour calculer un monoïde à partir d'une boucle d'ordre 5 en utilisant $Alg.v4$ (le monoïde obtenu aura alors environ 450 éléments), alors qu'il ne fallait que quelques minutes à $Alg.v3$ pour calculer le même monoïde en utilisant tous les mots de longueur 2 ou moins. Si $|Ctx_{(\leq LMC)}|$ est beaucoup plus grand que $|Repr|$, le gain de temps de $Alg.v4$ par

rapport à $Alg.v3$ est par contre très important.

Puisqu'un monoïde dérivé d'une boucle d'ordre 6 peut contenir plus de 17 000 éléments (voir Chapitre 5) et que la quantité de contextes d'évaluation utilisés dans cette version équivaut au nombre de représentants, le calcul d'un monoïde avec cette version de l'algorithme est encore très long.

Lemme 4.3.3 (Invariant). *Pendant l'exécution de la fonction* $\texttt{Calculer\_Repr}_{Alg.v4}$, *après chaque exécution de la fonction* $\texttt{Classer\_Nouveaux\_Ra}_{Alg.v4}$, *les règles sont cohérentes pour l'ensemble* $(Repr_{(0...k-2)} \cup Repr_{(k-1)})$, *i.e. :*

$$\forall (u,v) \in Regles,$$
$$\forall x,y \in (Repr_{(0...k-2)} \cup Repr_{(k-1)}),$$
$$eval(xuy) = eval(xvy)$$

Preuve. La preuve est par induction sur le nombre d'itérations (k). Au départ, $k = 1$ et il y a $|B|$ éléments dans $Repr_{(k-1)}$ et $|B|$ éléments dans l'ensemble $Regles$, i.e. $\forall a \in B,\ a \in Repr_{(k-1)}$ et $(a,a) \in Regles$. $Repr_{(0...k-2)}$ et $Repr_{(k)}$ sont vides. Puisque dans ce cas $\forall (u,v) \in Regles,\ u = v$, il est évident que l'invariant est respecté.

Par hypothèse d'induction, nous savons qu'au début de l'itération k, $\forall x,y \in Repr_{(0...k-2)}$, $\forall (u,v) \in Regles$, $eval(xuy) = eval(xvy)$. Il nous faut donc démontrer deux choses. Premièrement, que les règles présentes au début de l'itération sont cohérentes si les représentants contenus dans l'ensemble $Repr_{(k-1)}$ sont également utilisés comme contextes d'évaluation. Deuxièmement, nous devons démontrer que les règles ajoutées pendant cette itération respectent également l'invariant. Notons que nous regarderons uniquement les ajouts de règles non triviales (règles (u,v) pour lesquelles $u \neq v$).

La première partie consiste en réalité à démontrer que le *retour en arrière* effectué par la fonction $\texttt{Verifier\_Validite\_Regles}_{Alg.v4}$ fonctionne correctement. En fait, cette fonction vérifie que pour toute règle (u, v), les évaluations contenues dans les parties $\boxed{B_1}$, $\boxed{B_2}$ et $\boxed{B_3}$ de la Zone $\boxed{B}$ de la Figure 4.2 sont identiques pour u et v, ce qui correspond exactement au *retour en arrière*. Comme nous l'avons vu précédemment, si l'un des tests échoue, la règle (u, v) sera retirée et une nouvelle règle (u, v') sera ajoutée par la fonction $\texttt{Corriger\_Regle}_{Alg.v4}$. Puisque cette fonction effectue les mêmes vérifications que $\texttt{Verifier\_Validite\_Regles}_{Alg.v4}$, la règle ajoutée sera conforme à l'invariant. Nous pouvons en déduire qu'après l'exécution de $\texttt{Verifier\_Validite\_Regles}_{Alg.v4}$, l'ensemble des $(u, v) \in \mathit{Regles}$ respecte l'invariant.

Deuxièmement, nous devons regarder la fonction $\texttt{Classer\_Nouveaux\_Ra}_{Alg.v4}$, puisque celle-ci ajoute une règle non triviale lorsqu'un nouveau successeur de représentant n'est pas un représentant. Cela arrive lorsque v, le représentant de ua_i calculé par la fonction $\texttt{Trouver\_Repr}_{Alg.v4}(ua_i, \mathit{Repr})$, est différent de ua_i (ligne 7 de $\texttt{Classer\_Nouveaux\_Ra}_{Alg.v4}$). Puisque la fonction $\texttt{Trouver\_Repr}_{Alg.v4}$ retourne un élément v tel que $\forall x, y \in (\mathit{Repr}_{(0\ldots k-2)} \cup \mathit{Repr}_{(k-1)})$, $\mathit{eval}(xua_iy) = \mathit{eval}(xvy)$, la règle (ua_i, v) ajoutée à l'ensemble *Regles* respecte l'invariant, ce qui conclut la preuve. □

Voici maintenant le deuxième invariant de la boucle "Tant que" de la fonction $\texttt{Calculer\_Repr}_{Alg.v4}$.

Lemme 4.3.4 (Invariant). *Pendant l'exécution de la fonction* $\texttt{Calculer\_Repr}_{Alg.v4}$, *après chaque exécution de la fonction* $\texttt{Classer\_Nouveaux\_Ra}_{Alg.v4}$, *les représentants forment des classes distinctes, i.e.*

$$\forall r_1, r_2 \in Repr,\ r_1 \neq r_2,$$
$$\exists x, y \in (Repr_{(0...k-2)} \cup Repr_{(k-1)}), \text{ tel que}$$
$$eval(xr_1y) \neq eval(xr_2y)$$

Preuve. La preuve est par induction mathématique sur le nombre d'éléments de l'ensemble *Repr*. À la base, il y a $|B|$ éléments dans $Repr_{(k-1)}$ $(\forall a \in B,\ a \in Repr_{(k-1)})$. $Repr_{(0...k-2)}$ et $Repr_{(k)}$ sont vides. Soit $e \in B$, l'élément identité de la boucle. Si $x = y = e$, $\forall a, b \in Repr$, $a \neq b$, $eval(xay) = eval(a) = a \neq b = eval(b) = eval(xby)$. Donc chacune des classes de bases sont distinctes.

Il y a deux fonctions desquelles peuvent être ajoutés des nouveaux représentants : $\texttt{Corriger\_Regle}_{Alg.v4}$ et $\texttt{Classer\_Nouveaux\_Ra}_{Alg.v4}$. Premièrement, observons en détails la fonction $\texttt{Corriger\_Regle}_{Alg.v4}$. Un représentant sera ajouté pendant l'exécution de cette fonction si, suite à l'exécution de la boucle "Tant que" de la fonction, la liste *Liste_Repr_Possibles* est vide. Cette liste est initialisée au début de la fonction avec tous les représentants r dont la règle (r, v) existait à l'itération précédente, c'est-à-dire tel que $\forall x, z \in Repr_{(0...k-2)}$, $eval(xrz) = eval(xvz) = eval(xuz)$ (Invariant 4.3.3). S'il ne reste aucun élément dans la liste, c'est donc que $\forall r \in Repr$, $\exists x, y \in (Repr_{(0...k-2)} \cup Repr_{(k-1)})$ tel que $eval(xuy) \neq eval(xry)$ et par hypothèse d'induction l'élément u ajouté respecte l'invariant.

Voyons maintenant $\texttt{Classer\_Nouveaux\_Ra}_{Alg.v4}$ en détails. Dans cette fonction, si l'on ajoute un nouveau représentant, c'est que la fonction $\texttt{Trouver\_Repr}_{Alg.v4}$ a retournée $v = ua_i$, ce qui signifie $\nexists v \in Repr$, tel que $\forall x, y \in (Repr_{(0...k-2)} \cup Repr_{(k-1)})$,

$eval(xua_iy) = eval(xvy)$, donc l'invariant est respecté pour ce représentant également, ce qui conclut la preuve. □

Le lemme suivant démontre que suite au calcul effectué par la version $Alg.v4$ de l'algorithme, il existe une règle associant chaque successeur de représentant à son représentant, démontrant par le fait même que la fonction $\texttt{Mult}_{Alg.v4}$ peut effectuer sa tâche correctement.

Lemme 4.3.5. *À la fin de l'exécution de la fonction* $\texttt{Calculer\_Repr}_{Alg.v4}(B, |B|)$, $\forall r \in Repr_{(0...k-2)}$, $\forall a \in B$, $\exists v \in Repr_{(0...k-2)}$ *tel que* $(ra, v) \in Regles$.

Preuve. À chacune des itérations, la fonction $\texttt{Calculer\_Nouveaux\_Ra}_{Alg.v4}$ prend chacun des éléments de l'ensemble $Repr_{(k-1)}$, lui ajoute tour à tour chacune des lettres de la boucle et elle produit une règle associant ce nouveau mot à son représentant (lequel peut être lui-même). Nous savons qu'avec les opérations effectuées à la fin de chacune des itérations sur les sous-ensembles de $Repr$ par la fonction $\texttt{Changer\_Iteration}_{Alg.v4}$, tous les éléments ajoutés à $Repr_{(k)}$ se retrouverons dans $Repr_{(k-1)}$ avant de se retrouver dans l'ensemble $Repr_{(0...k-2)}$. Considérant qu'au début et à la fin de la boucle "Tant que" de la fonction $\texttt{Calculer\_Repr}_{Alg.v4}$ les ensembles $Repr_{(0...k-2)}$ et $Repr_{(k)}$ sont vides, il existe une règle associant chaque successeur de représentant ra à son propre représentant. □

4.4 Utilisation d'un sous-ensemble variable des représentants pour l'ensemble Ctx ($Alg.v5$)

À la Section 4.2, nous avons vu qu'il est possible de calculer un monoïde à partir d'une boucle finie en utilisant seulement des mots de petites tailles comme contextes

d'évaluation ($Alg.v3$). De plus, nous avons vu que le monoïde obtenu est intimement lié au monoïde dérivé. Puis, nous avons étudié à la Section 4.3 la possibilité d'utiliser seulement les représentants comme contextes d'évaluation ($Alg.v4$). Avec ces deux méthodes, le temps d'exécution des deux versions peut parfois être encore trop long. L'idée dans cette section est de combiner ces deux méthodes servant à réduire le nombre de contextes d'évaluation utilisés, en une nouvelle version beaucoup plus rapide ($Alg.v5$).

Ainsi, à l'intérieur de la fonction $\texttt{Calculer\_Repr}_{Alg.v5}(B, |B|)$ (Fonction 4.14), nous tentons d'abord de calculer un monoïde à l'aide des représentants de longueur 0 (nous considérons l'identité comme le représentant de longueur 0). Si le résultat obtenu $\big((Repr, \bullet)$, où $u \bullet v = \texttt{mult}_{Alg.v5}(u, v)\big)$ n'est pas associatif, nous ajoutons les représentants de longueur 1 comme contextes d'évaluations. Nous procédons ainsi de suite en augmentant la longueur des contextes d'évaluation jusqu'à l'obtention d'un monoïde ou jusqu'à ce que tous les représentants soient utilisés comme contextes d'évaluation (ce qui revient alors à l'algorithme $Alg.v4$) (voir Figure 4.3). C'est la variable $Longueur\_Ctx$ qui contiendra la longueur maximale des contextes d'évaluation utilisés à l'itération en cours.

À priori, l'obtention d'un monoïde n'est pas garantie, mais en pratique nous n'avons aucun exemple où il a été impossible de calculer un monoïde avec cette technique. Notons également que le monoïde obtenu en pratique avec l'algorithme $Alg.v5$ a toujours été le même que celui obtenu avec les algorithmes $Alg.v3$ et $Alg.v4$ et ce même dans les cas où il est prouvé que le monoïde obtenu est le monoïde dérivé.

Cette nouvelle version est une modification de l'algorithme $Alg.v4$, mais contrairement à $Alg.v4$, la nouvelle version n'utilise pas nécessairement tous les représentants comme contextes d'évaluations. Ainsi, pour que la gestion des contextes d'évaluations soit plus simple et plus efficace, nous avons ajouté un ensemble (Ctx), séparé de $Repr$

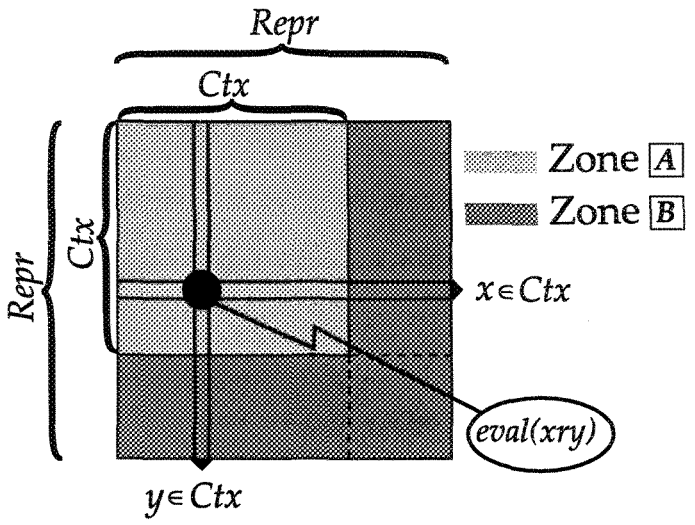

Figure 4.3: Table des évaluations d'un mot r selon l'algorithme $Alg.v5$. La Zone $\boxed{A}$ représente les évaluations déjà testées et la Zone $\boxed{B}$ celles qui pourraient être testées à une itération subséquente.

mais non distinct, les contenants. Notons que cet ensemble est également séparé en trois sous-ensembles distincts $\left(Ctx = (Ctx_{(0...k-2)} \cup Ctx_{(k-1)} \cup Ctx_{(k)})\right)$, séparant les contextes d'évaluations selon le numéro de l'itération à laquelle ils ont été ajoutés à l'ensemble Ctx. En tout temps, nous avons $Ctx \subseteq Repr$. La version $Alg.v5$ de l'algorithme utilise sensiblement les mêmes fonctions que la version $Alg.v4$. En effet, les fonctions $\texttt{Mult}_{Alg.v5}(u, v)$ et $\texttt{Verifier\_Asso}_{Alg.v5}(Repr_{(0...k-2)}, Regles)$ (respectivement Fonctions 4.21 et 4.22) sont identiques à leur version précédente, alors que les fonctions $\texttt{Verifier\_Validite\_Regles}_{Alg.v5}(Regles,\ Repr_{(k)},\ Ctx)$ et $\texttt{Trouver\_Repr}_{Alg.v5}(u)$ (respectivement Fonctions 4.17 et 4.20) effectuent les mêmes tâches que leur dernière version, mais en utilisant les éléments de l'ensemble Ctx comme contextes d'évaluation plutôt que ceux de l'ensemble $Repr$.

Dans le cas des fonctions $\texttt{Corriger\_Regle}_{Alg.v5}((u, v), Liste\_retires, Regles, Ctx, Repr_{(k)}, Longueur\_Ctx)$ (Fonction 4.18) et $\texttt{Classer\_Nouveaux\_Ra}_{Alg.v5}(Regles, Repr, Ctx, Longueur\_Ctx)$ (Fonction 4.19), elles utilisent également l'ensemble Ctx comme

ensemble de contextes d'évaluation, mais elles effectuent en plus une tâche supplémentaire. En effet, lorsqu'elles ajoutent un représentant u, elles doivent vérifier si $|u| \leq Longueur\_Ctx$ et si c'est le cas, l'ajouter à l'ensemble $Ctx_{(k)}$.

En plus de transférer les représentants dans les bons ensembles lors du changement d'itération, la fonction $\texttt{Changer\_Iteration}_{Alg.v5}(Repr,\ Ctx,\ k)$ (Fonction 4.15) fera maintenant de même pour les contextes d'évaluations.

Finalement, une nouvelle fonction servant à ajouter les représentants de longueur $Longueur\_Ctx$ à l'ensemble Ctx est maintenant nécessaire. C'est la fonction $\texttt{Ajouter\_Ctx}_{Alg.v5}(Repr_{(0...k-2)}, Ctx_{(k-1)}, Longueur\_Ctx)$ (Fonction 4.16) qui effectue cette tâche. Suite à l'exécution de cette fonction, nous aurons donc $Ctx = \{r \in Repr \mid |r| \leq Longueur\_Ctx\}$.

4.4.1 Propriétés du monoïde obtenu

Notons d'abord que, sans perte de généralité, l'ensemble des énoncés et des preuves concernant l'algorithme $Alg.v4$ faites à la Section 4.3.1 sont toujours vrais pour cette nouvelle version, mais en utilisant l'ensemble Ctx pour les contextes d'évaluations au lieu de l'ensemble $Repr$. Plus particulièrement, les énoncés des deux invariants de la boucle "Tant que" (ligne 18) de la fonction $\texttt{Calculer\_Repr}_{Alg.v5}$ s'énoncent maintenant comme suit. Premièrement, les représentants forment des classes distinctes ($\forall r_1, r_2 \in Repr\ (r_1 \neq r_2)$, $\exists x, y \in (Ctx_{(0...k-2)} \cup Ctx_{(k-1)})$ tel que $eval(xr_1y) \neq eval(xr_2y)$. Deuxièmement, les règles sont cohérentes avec l'ensemble des contextes d'évaluation ajoutés avant l'étape en cours ($\forall (u, v) \in Regles,\ \forall x, y \in (Ctx_{(0...k-2)} \cup Ctx_{(k-1)})$, $eval(xuy) = eval(xvy)$).

Bien qu'il n'a pas été possible de démontrer que l'algorithme $Alg.v5$ donne toujours le même résultat que les algorithmes $Alg.v3$ et $Alg.v4$, les résultats obtenus en pratique ont toujours été les mêmes. De plus, il suffit souvent d'utiliser des représentants très

Fonction 4.14 : Calculer_Repr$_{Alg.v5}(B, |B|)$

1: $k \leftarrow 0$
2: $Longueur\_Ctx \leftarrow 0$
3:
4: $Repr_{(0...k-2)} \leftarrow <>$
5: $Repr_{(k-1)} \leftarrow < a_1, a_2, \ldots, a_n >$
6: $Repr_{(k)} \leftarrow <>$
7: $Ctx_{(0...k-2)} \leftarrow <>$
8: $Ctx_{(k-1)} \leftarrow < a_1 >$ *//où a_1 est l'identité de B*
9: $Ctx_{(k)} \leftarrow <>$
10: $Regles \leftarrow < (a_1, a_1), (a_2, a_2), \ldots, (a_n, a_n) >$
11:
12: Classer_Nouveaux_Ra$_{Alg.v5}(Regles,\ Repr,\ Ctx,\ Longueur_Ctx)$
13: Changer_Iteration$_{Alg.v5}(Repr,\ Ctx,\ k)$
14:
15: **Tant que** $(\neg$Verifier_Asso$_{Alg.v5}(Repr_{(0...k-2)},\ Regles)$ **ET** $(|Repr| \neq |Ctx_{(0...k-2)}|))$ **faire**
16: $Longueur\_Ctx++$
17: Ajouter_Ctx$_{Alg.v5}(Repr_{(0...k-2)},\ Ctx_{(k-1)},\ Longueur_Ctx)$
18: **Tant que** $(|Repr_{(k-1)}| > 0)$ **OU** $(|Ctx_{(k-1)}| > 0)$ **faire**
19: Verifier_Validite_Regles$_{Alg.v5}(Regles,\ Repr_{(k)},\ Ctx,\ Longueur_Ctx)$
20: Classer_Nouveaux_Ra$_{Alg.v5}(B,\ Regles,\ Repr,\ Ctx,\ Longueur_Ctx)$
21: Changer_Iteration$_{Alg.v5}(Repr,\ Ctx,\ k)$
22:
23: **Si** Verifier_Asso$_{Alg.v5}(Repr_{(0...k-2)}, Regles)$ **alors**
24: **retourner** $Repr$
25: **Sinon**
26: **retourner** NULL

Fonction 4.15 : Changer_Iteration$_{Alg.v5}(Repr,\ Ctx,\ k)$

1: $Repr_{(0...k-2)} \rightarrow ajouter(Repr_{(k-1)})$
2: $Repr_{(k-1)} \leftarrow Repr_{(k)}$
3: $Repr_{(k)} \leftarrow <>$
4:
5: $Ctx_{(0...k-2)} \rightarrow ajouter(Ctx_{(k-1)})$
6: $Ctx_{(k-1)} \leftarrow Ctx_{(k)}$
7: $Ctx_{(k)} \leftarrow <>$
8:
9: $k++$

Fonction 4.16 : `Ajouter_Ctx`$_{Alg.v5}(Repr_{(0...k-2)},\ Ctx_{(k-1)},\ Longueur\_Ctx)$

1: **Pour tout** $u \in Repr_{(0...k-2)}$ **tel que** $|u| = Longueur\_Ctx$ **faire**
2: $\quad Ctx_{(k-1)} \rightarrow ajouter(u)$

Fonction 4.17 : `Verifier_Validite_Regles`$_{Alg.v5}(Regles,\ Repr_{(k)},\ Ctx,\ Longueur\_Ctx)$

1: $Liste\_retires = <>$
2: **Pour tout** $(u, v) \in Regles$ **faire**
3: $\quad$ **Si** $u \neq v$ **alors**
4: $\quad\quad$ **Pour tout** $x \in Ctx_{(k-1)}$ **faire**
5: $\quad\quad\quad$ **Pour tout** $z \in Ctx_{(k-1)}$ **faire**
6: $\quad\quad\quad\quad$ **Si** $eval(xuz) \neq eval(xvz)$ **alors**
7: $\quad\quad\quad\quad\quad$ `Corriger_Regle`$_{Alg.v5}\big((u,v),\ Liste\_retires,\ Regles,\ Repr_{(k)},\ Ctx,\ Longueur\_Ctx\big)$
8: $\quad\quad\quad$ **Pour tout** $y \in Ctx_{(0...k-2)}$ **faire**
9: $\quad\quad\quad\quad$ **Si** $eval(xuy) \neq eval(xvy)$ OU **si** $eval(yux) \neq eval(yvx)$ **alors**
10: $\quad\quad\quad\quad\quad$ `Corriger_Regle`$_{Alg.v5}\big((u,v),\ Liste\_retires,\ Regles,\ Repr_{(k)},\ Ctx,\ Longueur\_Ctx\big)$

Fonction 4.18 : `Corriger_Regle`$_{Alg.v5}((u, v),\ Liste\_retires,\ Regles,\ Ctx,\ Repr_{(k)},\ Longueur\_Ctx)$

1: $Regles \rightarrow retirer((u, v))$
2: $Liste\_Repr\_Possibles \leftarrow Liste\_retires \rightarrow liste[v]$
3: $r \leftarrow Liste\_Repr\_Possibles \rightarrow Premier\_element$
4:
5: **Tant que** $|Liste\_Repr\_Possibles| > 0$ et que $r \neq$ `NULL` **faire**
6: $\quad$ **Pour tout** $x \in Ctx_{(k-1)}$ **faire**
7: $\quad\quad$ **Pour tout** $z \in Ctx_{(k-1)}$ **faire**
8: $\quad\quad\quad$ **Si** $eval(xuz) \neq eval(xrz)$ **alors**
9: $\quad\quad\quad\quad Liste\_Repr\_Possibles \rightarrow retirer(r)$
10: $\quad\quad$ **Pour tout** $y \in Ctx_{(0...k-2)}$ **faire**
11: $\quad\quad\quad$ **Si** $eval(xuy) \neq eval(xry)$ OU **si** $eval(yux) \neq eval(yrx)$ **alors**
12: $\quad\quad\quad\quad Liste\_Repr\_Possibles \rightarrow retirer(r)$
13: $\quad r++$
14: **Si** $|Liste\_Repr\_Possibles| == 1$ **alors**
15: $\quad$ Soit $r \in Liste\_Repr\_Possibles$
16: $\quad Regles \rightarrow ajouter((u, r))$
17: **Sinon**
18: $\quad Repr_{(k)} \rightarrow ajouter(u)$
19: $\quad$ **Si** $|u| \leq Longueur\_Ctx$ **alors**
20: $\quad\quad Ctx_{(k)} \rightarrow ajouter(u)$
21: $\quad Regles \rightarrow ajouter((u, u))$
22: $\quad Liste\_retires \rightarrow liste[v] \rightarrow ajouter(u)$

Fonction 4.19 : $\texttt{Classer\_Nouveaux\_Ra}_{Alg.v5}(B,\ Regles,\ Repr,\ Ctx,\ Longueur\_Ctx)$

1: **Pour tout** $u \in Repr_{(k-1)}$ **faire**
2: **Pour** $i \leftarrow 1$ à $|B|$ **faire**
3: $v \leftarrow \texttt{Trouver\_Repr}_{Alg.v5}(ua_i,\ Repr,\ Ctx_{(0...k-2)},\ Ctx_{(k-1)})$
4: **Si** $v = ua_i$ **alors**
5: $Repr_{(k)} \rightarrow ajouter(ua_i)$
6: $Regles \rightarrow ajouter((ua_i, ua_i))$
7: **Si** $|ua_i| \leq Longueur\_Ctx$ **alors**
8: $Ctx_{(k)} \leftarrow ajouter(ua_i)$
9: **Sinon**
10: $Regles \rightarrow ajouter((ua_i, v))$

Fonction 4.20 : $\texttt{Trouver\_Repr}_{Alg.v5}(u,\ Repr,\ Ctx)$

1: **Si** $\exists v \in Repr$ **tel que** $\forall x, y \in (Ctx_{(0...k-2)} \cup Ctx_{(k-1)})$, $eval(xuy) = eval(xvy)$ **alors**
2: **retourner** v *//car v est le représentant actuel de u*
3: **Sinon**
4: **retourner** u *//car u est son propre représentant*

Fonction 4.21 : $\texttt{Mult}_{Alg.v5}(u, v)$

1: *//La fonction* $\texttt{Mult}_{Alg.v5}$ *est identique à la fonction* $\texttt{Mult}_{Alg.v4}$
2: $X \leftarrow u$
3:
4: **Pour** $i \leftarrow 1$ à $|v|$ **faire**
5: **Soit** $Y \in Repr$ tel que $(Xv_i, Y) \in Regles$
6: $X \leftarrow Y$ *//Y est le représentant de Xv_i (multiplication de X par v_i)*
7: *//$Y = Xv_i$ si Xv_i est un représentant*
8: **retourner** X

Fonction 4.22 : $\texttt{Verifier\_Asso}_{Alg.v5}(B, Repr)$

1: *//La fonction* $\texttt{Verifier\_Asso}_{Alg.v5}$ *est identique à la fonction* $\texttt{Verifier\_Asso}_{Alg.v4}$
2: **Si** $\exists x, y \in Repr$ **ET** $\exists a \in B$ **tel que** $\left(\texttt{Mult}_{Alg.v5}(\texttt{Mult}_{Alg.v5}(x, a),\ y)\right) \neq \left(\texttt{Mult}_{Alg.v5}(x,\ \texttt{Mult}_{Alg.v5}(a, y))\right)$ **alors**
3: **retourner** FAUX
4: **Sinon**
5: **retourner** VRAI

petits pour calculer un monoïde. Ainsi, les représentants de longueur 3 et moins sont suffisants pour calculer le monoïde d'une boucle particulière d'ordre 6. Pour effectuer une comparaison pratique, il faut environ 11 minutes à l'algorithme $Alg.v5$ pour calculer un monoïde de 342 éléments à partir de cette boucle sur un ordinateur utilisant un processeur de 3.2 GHz. Il en faut environ 19 minutes à l'algorithme $Alg.v3$ (en utilisant tous les mots plus petits que 3) et environ 60 minutes à l'algorithme $Alg.v4$. Souvenons-nous que l'algorithme $Alg.v4$ est plus rapide que l'algorithme $Alg.v3$ si et seulement si $|Ctx_{(\leq LMC)}| > Repr$. Nous étudierons plus en détails ces résultats dans le chapitre suivant.

CHAPITRE 5

ANALYSE DES RÉSULTATS ET DISCUSSION

5.1 Introduction

Lors des Chapitres 3 et 4, nous avons vu cinq versions d'un algorithme dont l'objectif est de calculer le monoïde dérivé d'une boucle finie. Nous avons démontré que les deux premières versions ($Alg.v1$ et $Alg.v2$) calculent toujours le monoïde dérivé (voir Sections 3.6.1 et 3.7.1). Par contre, ces deux versions sont en pratique inutilisables car elles utilisent tous les mots de longueur plus petite que deux fois l'ordre de la boucle comme contextes d'évaluation.

Ainsi, nous avons introduit trois nouvelles versions ($Alg.v3$, $Alg.v4$ et $Alg.v5$) utilisant trois solutions différentes pour réduire la quantité de contextes d'évaluation utilisés. Bien qu'il ne soit pas prouvé que ces nouvelles versions calculent toujours le monoïde dérivé, elles calculent en pratique un unique monoïde que nous avons appelé le *monoïde pseudo-dérivé* d'un boucle, dénoté $Repr$. Nous avons vu que ce monoïde possède des liens étroits avec le monoïde dérivé. Notamment, nous savons que $|Repr| \leq |D(B)|$ et que deux représentants dans $Repr$ font partie de deux classes d'équivalences différentes dans $D(B)$.

La première solution ($Alg.v3$) consiste à utiliser tous les mots dont la longueur est plus petite qu'une certaine valeur (LMC) fournie par l'usager. Nous avons démontré à la Section 4.2.1 que cette version permet de calculer le monoïde dérivé de certaines boucles

possédant des caractéristiques particulières. La quatrième et la cinquième version utilisent les représentants trouvés comme contextes d'évaluation, la différence principale étant que $Alg.v4$ utilise nécessairement tous les représentants, alors que $Alg.v5$ les ajoute par longueur et n'en utilise généralement qu'une partie. Notons également que ces deux versions ont un ensemble de contextes d'évaluation variant pendant l'exécution de l'algorithme.

L'objectif de ce chapitre est de présenter les résultats obtenus sur les boucles d'ordre 8 et moins à l'aide des trois dernières versions de l'algorithme. L'étude de ces résultats nous permettra de mieux comprendre les liens existants entre une boucle et son monoïde dérivé. À la Section 5.2, nous présentons le Tableau 5.1 contenant les boucles pour lesquelles nous avons réussi à calculer leur monoïde dérivé à l'aide de la version $Alg.v3$ de l'algorithme. Les boucles pour lesquelles nous avons calculé un monoïde pseudo-dérivé sont présentés dans le Tableau 5.2 à la Section 5.3. Notons que les monoïdes pseudo-dérivés ont été calculés à l'aide de la version $Alg.v5$ de l'algorithme puisque c'est la plus rapide des trois. Comme nous le verrons, certaines vérifications ont tout de même été effectuées à l'aide des versions $Alg.v3$ et $Alg.v4$.

Pour chacune des boucles présentées dans le Tableau 5.1 (resp. Tableau 5.2), nous donnerons plusieurs propriétés[1] s'appliquant à elle-même ou a son monoïde dérivé (resp. monoïde pseudo-dérivé). La liste de ces propriétés est présentée à la Figure 5.1. Voici maintenant leur définition. Soit $(B, \cdot)$, une boucle finie.

[1]La plupart des propriétés ont été calculées par des étudiants travaillant pour François Lemieux (professeur à l'UQAC) à l'été 2003 (Daniel Bouchard et Claude Sasseville) et à l'été 2004 (Gabriel Lavoie et Sébastien Noël).

	Liste des propriétés
0	Associative
1	Di-Associative
2	Power-Associative
3	Nilpotente
4	Résoluble
5	Apériodique
6	Commutative
7	M-Nilpotente
8	M-Résoluble
9	$D(B) \in ECOM$
10	*Retour en arrière* requis

Figure 5.1: Liste des propriétés considérées sur la boucle et son monoïde dérivé

0. Associative

Au Chapitre 2, nous avons vu qu'une boucle associative est une boucle telle que $\forall x, y, z \in B$, $(x \cdot (y \cdot z)) = ((x \cdot y) \cdot z)$. On pourrait également définir une boucle associative comme une boucle telle que $\forall x, y, z \in B$, la sous-boucle générée par (x, y, z) est associative. Cette définition met en évidence la relation entre une boucle associative, di-associative et power-associative.

1. Di-Associative

$\forall x, y \in B$, la sous-boucle générée par (x, y) est associative.

2. Power-Associative

$\forall x \in B$, la sous-boucle générée par (x) est associative.

3. Nilpotente

On définit le *centre* Z d'une boucle B comme l'ensemble $\{a \in B \mid \forall x, y \in B, (a \cdot x) = (x \cdot a), ((a \cdot x) \cdot y) = (a \cdot (x \cdot y)), ((x \cdot y) \cdot a) = (x \cdot (y \cdot a))\}$. Autrement dit, Z est formé de tous les éléments de B qui sont commutatifs et associatifs avec tous les éléments de

B. Il est facile de vérifier que l'ensemble Z forme une sous-boucle normale dans B.

On définit alors récursivement la classe de nilpotence d'une boucle comme suit : si B est associative, alors B est nilpotente de classe 1. Si B possède un centre non trivial Z et que la boucle B/Z est nilpotente de classe m, alors B est nilpotente de classe $m+1$.

4. Résoluble

Soit N, la plus petite sous-boucle normale de B telle que B/N est une boucle associative et commutative. Alors, nous définissons récursivement la classe de résolubilité d'une boucle comme suit : si $N = 1$, alors B est résoluble de classe 1. Si N est résoluble de classe m, alors B est résoluble de classe $m+1$.

5. Apériodique

On dit que la boucle B est apériodique si elle ne possède pas de sous-boucle associative non triviale. De façon équivalente, $\forall x \in B$, $\exists n \in \mathbb{N}$ tel que $x^n = x^{n+1}$.

6. Commutative

$\forall x, y \in B$, $(x \cdot y) = (y \cdot x)$.

7. M-Nilpotence et 8. M-Résoluble

Soit $b \in B$, on définit les applications $R_b, L_b : B \to B$ par $R_b(a) = a \cdot b$ et $L_b(a) = b \cdot a$ pour tout $a \in B$. On nomme *groupe de multiplication* de B, dénoté $\mathcal{M}(B)$, le groupe généré par l'ensemble $\{R_a, L_a \mid a \in B\}$ à l'aide de l'opération de composition des applications. Les classes de M-Nilpotence et de M-Résolubilité de la boucle B correspondent respectivement à la classe de nilpotence et à la classe de résolubilité de $\mathcal{M}(B)$. Puisque qu'un groupe est en fait une boucle associative, les définitions de nilpotence et de résolubilité de $\mathcal{M}(B)$ sont celles des boucles.

9. D(B) $\in$ ECOM

On dit qu'un élément $x \in D(B)$ est idempotent si $(x \cdot x) = x$. Soit E, l'ensemble de tous les éléments idempotents de $D(B)$. Alors $D(B) \in ECOM$ si et seulement si $\forall x, y \in E$, $(x \cdot y) = (y \cdot x)$.

10. *Retour en arrière* requis

Puisque les versions $Alg.v4$ et $Alg.v5$ utilisent les représentants trouvés comme ensemble de contextes d'évaluation, certains calculs doivent être vérifiés lorsque des nouveaux représentants sont ajoutés. Nous avons nommé *retour en arrière* l'opération consistant à s'assurer que les règles contenues dans l'ensemble *Regles* sont cohérentes en utilisant ces nouveaux représentants comme contextes d'évaluation (voir Section 4.3). Si une règle (u, v) n'est pas cohérente ($\exists x, y \in Repr$ tel que $eval(xuy) \neq eval(xvy)$), elle est détruite et une nouvelle règle cohérente (u, v') est insérée.

En effectuant quelques calculs pratiques, nous avons découvert que pour la majorité des boucles le retour en arrière ne corrige aucune règle. Si une boucle ne nécessite pas le retour en arrière et si u et v sont deux représentants tel que $u \neq v$ et $|u| \geq |v|$, alors cela signifie qu'il existe $x, y \in Repr$ tel que $|x| < |u|$ et $|y| < |u|$ et tel que $eval(xuy) \neq eval(xvy)$. Autrement dit, il n'est jamais nécessaire d'utiliser des contextes d'évaluation plus grand ou égal à $|u|$ pour déterminer si u et v appartiennent à la même classe. Dans la version $Alg.v5$ de l'algorithme, le retour en arrière devient une opération très coûteuse lorsque la longueur des représentants utilisés est grande (4 ou plus). Il serait alors très intéressant de connaître les conditions algébriques nécessaires à une boucle pour que ce calcul ne soit pas requis.

5.2 Monoïdes dérivés

Il est facile de vérifier que deux boucles isomorphiques reconnaissent les mêmes langages et que leur monoïde dérivé sont isomorphes. Ainsi, il n'est nécessaire que de calculer le monoïde dérivé d'une seule boucle par classe d'isomorphisme. Pour ce faire, nous utilisons les fichiers générés par l'algorithme présenté dans [18][2] contenant, pour un ordre donné, un élément de chacune des classes d'isomorphisme existantes. Nous identifions alors une boucle par bx_y, où x est l'ordre de la boucle et où y est son numéro d'apparition dans le fichier.

Puisque les deux premières versions de l'algorithme sont inutilisables en pratique, les seuls monoïdes dérivés que nous avons calculés l'ont été à l'aide de la version $Alg.v3$ de l'algorithme et à l'aide du Lemme 4.2.2 et du Théorème 4.2.3. Le lemme indique que le monoïde dérivé d'une boucle associative est égal à la boucle elle-même, alors que le théorème mentionne qu'il est plus facile de calculer le monoïde dérivé des boucles de petit ordre (8 et moins) possédant une sous-boucle normale.

Bien que les monoïdes dérivés présentés ont été calculés à partir de la version $Alg.v3$, il est important de noter que nous avons obtenu dans tout les cas le même monoïde en utilisant la version $Alg.v5$ et ce beaucoup plus efficacement. Bien sûr, il n'est pas encore prouvé que ce dernier obtient toujours le monoïde dérivé. Quelques tests ont également été effectués à l'aide de la version $Alg.v4$ et dans tous les cas, un monoïde isomorphe au monoïde dérivé à été obtenu.

Le Tableau 5.1 présente les résultats obtenus jusqu'à présent. Pour chacune des boucles, nous y retrouvons dans l'ordre les éléments suivants :

[2]Une version de ce programme ainsi que les fichiers générés par celui-ci sont disponibles à l'adresse suivante : `http ://wwwdim.uqac.ca/~flemieux/`

Description des colonnes du Tableau 5.1

No	Identification de la boucle de la forme bx_y
|D(B)|	Nombre d'éléments du monoïde dérivé
*Alg.v*5	La longueur des contextes nécessaires pour calculer le monoïde dérivé de la boucle à l'aide de la version *Alg.v*5 de l'algorithme
$\|N\|$	L'ordre de sa plus petite sous-boucle normale non triviale (s'il en existe une, sinon la case est laissée vide)
0...10	L'ensemble des propriétés mentionnées à la Figure 5.1

Tableau 5.1 : Monoïdes dérivés parmi les boucles d'ordre 5 à 8

No	$\|D(B)\|$	*Alg.v*5	$\|N\|$	0	1	2	3	4	5	6	7	8	9	10
b5_5	5	0		✓	✓	✓	1	1		✓	1	1	✓	
b6_0	6	0	2	✓	✓	✓	1	1		✓	1	1	✓	
b6_1	23	1	2				2	2		✓		2	✓	
b6_2	21	1	2				2	2				2	✓	
b6_9	6	0	3	✓	✓	✓		2				2	✓	
b6_77	161	1	3					2				3	✓	
b6_78	161	1	3					2				3	✓	
b6_81	17	1	3					2				2	✓	
b7_23714	7	0		✓	✓	✓	1	1		✓	1	1	✓	
b8_0	8	0	2	✓	✓	✓	1	1	✓		1	1	✓	
b8_1	44	1	2			✓	2	2	✓		3	2		
b8_2	62	1	2			✓	2	2			3	2		
b8_3	46	1	2			✓	2	2			3	2		
b8_4	46	1	2			✓	2	2			3	2		
b8_5	50	1	2			✓	2	2			3	2	✓	
b8_19	46	1	2			✓	2	2			3	2		
b8_20	46	2	2			✓	2	2	✓		3	2	✓	✓
b8_21	44	1	2			✓	2	2	✓		3	2		
b8_22	54	2	2			✓	2	2			3	2		
b8_23	62	1	2			✓	2	2			3	2		
b8_24	50	1	2			✓	2	2			3	2	✓	
b8_25	54	2	2			✓	2	2			3	2		
b8_94	46	2	2			✓	2	2	✓		3	2	✓	✓
b8_95	62	1	2			✓	2	2			3	2		
b8_96	54	2	2			✓	2	2			3	2		
b8_97	50	1	2			✓	2	2			3	2	✓	
b8_98	62	1	2			✓	2	2			3	2		
b8_99	54	2	2			✓	2	2			3	2		

Suite à la page suivante

Suite, boucles d'ordre 5 à 8

No	$\|D(B)\|$	*Alg.v5*	$\|N\|$	0	1	2	3	4	5	6	7	8	9	10
b8_115	8	0	2	✓	✓	✓	1	1	✓		1	1	✓	
b8_116	50	1	2			✓	2	2			3	2	✓	
b8_117	46	1	2			✓	2	2			3	2		
b8_118	46	1	2			✓	2	2			3	2		
b8_119	62	1	2			✓	2	2			3	2		
b8_120	60	1	2				2	2	✓		4	2		
b8_123	76	1	2				2	2			4	2	✓	
b8_126	38	1	2				2	2			4	2	✓	
b8_155	38	1	2				2	2			4	2	✓	
b8_160	46	2	2				2	2	✓		3	2	✓	✓
b8_211	72	1	2				2	2			4	2		
b8_212	54	1	2				2	2			4	2		
b8_213	56	1	2				2	2			4	2	✓	
b8_214	46	1	2				2	2			4	2	✓	
b8_215	46	1	2				2	2			4	2	✓	
b8_216	48	1	2				2	2			4	2	✓	
b8_217	50	1	2				2	2			4	2	✓	
b8_218	56	1	2				2	2			4	2		
b8_219	46	1	2			✓	2	2			3	2		
b8_220	62	1	2			✓	2	2			3	2		
b8_221	44	1	2			✓	2	2			3	2		
b8_222	46	2	2			✓	2	2			3	2	✓	✓
b8_233	8	0	2	✓	✓	✓	2	2			2	2	✓	
b8_279	46	1	2			✓	2	2			3	2		
b8_280	44	1	2			✓	2	2			3	2		
b8_286	46	1	2			✓	2	2			3	2		
b8_325	46	1	2				2	2			4	2		
b8_326	52	1	2				2	2			4	2		
b8_327	48	1	2				2	2			4	2	✓	
b8_328	46	2	2				2	2			3	2	✓	✓
b8_329	16	1	2				2	2			3	2	✓	
b8_330	46	1	2				2	2			4	2		
b8_2594	50	1	2			✓	2	2			3	2	✓	
b8_2595	50	1	2			✓	2	2			3	2	✓	
b8_2596	54	2	2			✓	2	2			3	2		
b8_2597	44	1	2			✓	2	2	✓		3	2	✓	
b8_2688	50	1	2			✓	2	2			3	2	✓	
b8_2689	50	1	2			✓	2	2			3	2	✓	
b8_2690	46	1	2			✓	2	2			3	2		

Suite à la page suivante

Suite, boucles d'ordre 5 à 8

No	$\|D(B)\|$	*Alg.v5*	$\|N\|$	0	1	2	3	4	5	6	7	8	9	10
b8_2691	46	2	2			✓	2	2	✓		3	2	✓	
b8_2692	50	1	2			✓	2	2			3	2	✓	
b8_2707	54	2	2			✓	2	2			3	2		
b8_2708	46	1	2			✓	2	2			3	2		
b8_2709	62	1	2			✓	2	2			3	2		
b8_2710	50	1	2			✓	2	2			3	2	✓	
b8_2818	72	1	2				2	2			4	2		
b8_2819	54	1	2				2	2			4	2		
b8_2820	46	1	2				2	2			4	2	✓	
b8_2821	48	1	2				2	2			4	2	✓	
b8_2822	66	1	2				2	2	✓		4	2		
b8_2823	44	1	2				2	2	✓		4	2	✓	
b8_2824	46	1	2				2	2			4	2	✓	
b8_2825	48	1	2				2	2			4	2	✓	
b8_2826	56	1	2				2	2			4	2	✓	
b8_2827	46	1	2				2	2			4	2	✓	
b8_2828	50	1	2				2	2			4	2	✓	
b8_2829	56	1	2				2	2			4	2		
b8_2830	46	1	2				2	2			4	2	✓	
b8_2831	44	1	2				2	2	✓		4	2		
b8_2832	44	1	2			✓	2	2			3	2	✓	
b8_2833	46	2	2			✓	2	2			3	2	✓	
b8_2874	54	2	2			✓	2	2			3	2		
b8_2875	44	1	2			✓	2	2			3	2		
b8_2881	44	1	2			✓	2	2			3	2	✓	
b8_2913	48	1	2				2	2			4	2	✓	
b8_2914	50	1	2				2	2			4	2	✓	
b8_2915	32	1	2				2	2			4	2	✓	
b8_2916	46	1	2				2	2			4	2	✓	
b8_2917	40	1	2				2	2			4	2	✓	
b8_2918	50	1	2				2	2			4	2	✓	
b8_13377	44	1	2			✓	2	2	✓		3	2	✓	
b8_13378	54	2	2			✓	2	2			3	2		
b8_13379	50	1	2			✓	2	2			3	2	✓	
b8_13390	44	1	2			✓	2	2	✓		3	2		
b8_13391	62	1	2			✓	2	2			3	2		
b8_13392	46	1	2			✓	2	2			3	2		
b8_13485	44	1	2				2	2	✓		4	2		
b8_13486	66	1	2				2	2	✓		4	2		

Suite à la page suivante

Suite, boucles d'ordre 5 à 8

No	$\|D(B)\|$	*Alg.v5*	$\|N\|$	0	1	2	3	4	5	6	7	8	9	10
b8_13487	48	1	2				2	2			4	2	✓	
b8_13488	46	1	2				2	2			4	2	✓	
b8_13489	56	1	2				2	2			4	2		
b8_13490	50	1	2				2	2			4	2	✓	
b8_13491	46	1	2				2	2			4	2	✓	
b8_13492	56	1	2				2	2			4	2	✓	
b8_13493	46	1	2				2	2			4	2	✓	
b8_13494	44	1	2				2	2	✓		4	2	✓	
b8_13495	48	1	2				2	2			4	2	✓	
b8_13496	46	1	2				2	2			4	2	✓	
b8_13497	54	1	2				2	2			4	2		
b8_13498	72	1	2				2	2			4	2		
b8_13499	50	1	2			✓	2	2			3	2	✓	
b8_13500	46	2	2			✓	2	2			3	2	✓	
b8_13544	50	1	2				2	2			4	2	✓	
b8_13545	48	1	2				2	2			4	2	✓	
b8_13546	46	1	2				2	2			4	2	✓	
b8_13547	40	1	2				2	2			4	2	✓	
b8_13548	32	1	2				2	2			4	2	✓	
b8_13549	50	1	2				2	2			4	2	✓	
b8_14533	46	1	2			✓	2	2			3	2		
b8_14617	56	1	2				2	2			4	2		
b8_14618	50	1	2				2	2			4	2	✓	
b8_14619	48	1	2				2	2			4	2	✓	
b8_14620	46	1	2				2	2			4	2	✓	
b8_14621	60	1	2				2	2	✓		4	2		
b8_14622	46	2	2				2	2	✓		3	2	✓	✓
b8_14623	38	1	2				2	2			4	2	✓	
b8_14624	76	1	2				2	2			4	2	✓	
b8_14625	46	1	2				2	2			4	2	✓	
b8_14626	56	1	2				2	2			4	2	✓	
b8_14627	54	1	2				2	2			4	2		
b8_14628	72	1	2				2	2			4	2		
b8_14629	38	1	2				2	2			4	2	✓	
b8_14630	8	0	2	✓	✓	✓	1	1	✓		1	1	✓	
b8_14631	8	0	2	✓	✓	✓	2	2			2	2	✓	
b8_14663	52	1	2				2	2			4	2		
b8_14664	46	1	2				2	2			4	2		

Suite à la page suivante

Suite, boucles d'ordre 5 à 8

No	$\|D(B)\|$	*Alg.v5*	$\|N\|$	0	1	2	3	4	5	6	7	8	9	10
b8_14665	16	1	2				2	2			3	2	✓	
b8_14666	48	1	2				2	2			4	2	✓	
b8_14667	46	2	2				2	2			3	2	✓	✓
b8_14668	46	1	2				2	2			4	2		

Tableau 5.1: Monoïdes dérivés calculés parmi les boucles d'ordre 5 à 8

5.3 Monoïdes pseudo-dérivés

Certaines des boucles présentées dans cette section possèdent une sous-boucle normale d'ordre 3, il serait donc possible de vérifier si leur monoïde pseudo-dérivé correspond au monoïde dérivé (*Alg.v3*, $LMC = 5$). Toutefois, calculer le monoïde dérivé d'une boucle à l'aide de tous les mots de longueur plus petite ou égale à 5 prend généralement plus de 20 jours. Il nous a donc été impossible de les calculer tous.

Le Tableau 5.2 présente les monoïdes pseudo-dérivés obtenus à l'aide de la version *Alg.v5* de l'algorithme. La description des colonnes est la même que pour le Tableau 5.1, la seule différence étant que les résultats concernant le monoïde dérivé concerne maintenant le monoïde pseudo-dérivé.

Tableau 5.2 : Monoïdes pseudo-dérivés calculés parmi les boucles d'ordre 5 à 8

No	$\|Repr\|$	*Alg.v5*	$\|N\|$	0	1	2	3	4	5	6	7	8	9	10
b5_0	454	2											✓	
b5_1	454	2											✓	
b5_2	519	2				✓							✓	
b5_3	345	2											✓	
b5_4	456	2							✓				✓	
b6_3	342	3	3			✓		2						✓
b6_4	3589	3											✓	
b6_5	3314	3												
b6_7	342	3	3					2						✓
b6_19	96	1											✓	
b6_38	342	3	3			✓		2				3		✓

Suite à la page suivante

Suite, boucles d'ordre 5 à 8

No	$\|Repr\|$	$Alg.v5$	$\|N\|$	0	1	2	3	4	5	6	7	8	9	10
b6_43	342	3	3					2				3		✓
b6_44	1518	2				✓				✓			✓	
b6_48	441	2	3					2		✓		2	✓	
b6_52	1810	2											✓	
b6_71	441	2	3			✓		2				2	✓	
b6_74	1268	2												
b6_75	1517	2											✓	
b6_76	1325	2											✓	
b6_82	429	2	3					2				2	✓	✓
b6_84	2350	2							✓				✓	
b6_85	2383	2							✓				✓	
b6_89	1955	2							✓				✓	
b6_90	1955	2							✓				✓	
b6_97	3300	2							✓				✓	
b6_100	2350	2							✓				✓	
b6_101	2383	2							✓				✓	
b6_105	2408	2							✓				✓	
b6_107	2408	2							✓				✓	
b6_108	3300	2							✓				✓	
b7_17574	7163	2							✓				✓	
b7_20885	10692	2							✓				✓	
b8_260534	108	1	4					2				4	✓	
b8_260541	108	1	4					2				4	✓	
b8_295667	188	1	4					2				4	✓	
b8_296057	140	1	4					2				4	✓	
b8_296078	140	1	4					2				4	✓	

Tableau 5.2: Monoïdes pseudo-dérivés calculés parmi les boucles d'ordre 5 à 8

Il est important de noter que le tableau précédent ne contient pas toutes les boucles d'ordre 8 possédant une sous-boucle normale de 4 éléments, puisqu'il en existe exactement 10 434[3].

Comme nous l'avons déjà mentionné, les résultats présentés dans cette section ont tous été obtenus à l'aide de la version $Alg.v5$ de l'algorithme car c'est la plus rapide.

[3]Une liste complète de ces boucles peut être obtenue à l'adresse suivante : `http ://wwwdim.uqac.ca/~flemieux/`

Par contre, plusieurs tests ont été effectués à l'aide des versions $Alg.v3$ et $Alg.v4$ et nous avons toujours obtenu le même monoïde. Notamment, toutes les boucles d'ordre 5 ont été testées en utilisant tous les mots de longueur 5 et moins ($Alg.v3$, $LMC = 5$) et en utilisant tous les représentants ($Alg.v4$). Voici finalement quelques faits concernant les calculs effectués dans ce chapitre.

- Parmi les boucles d'ordre 6, tous les monoïdes pseudo-dérivés nécessitant des contextes d'évaluation de longueur 0, 1 ou 2 et moins ont été calculés.
- Le monoïde pseudo-dérivé des boucles d'ordre 5 (b5_2) contient au maximum 519 éléments ; celui des boucles d'ordre 6 peut en contenir plus de 17 000 (b6_11) ; celui des boucles d'ordre 7 plus de 17 000 (b7_1) et celui des boucles d'ordre 8 plus de 454 000 (b8_336).
- Parmi les quelques boucles d'ordre 7 testées, aucun monoïde pseudo-dérivé n'a pu être calculé à l'aide des contextes d'évaluations de longueur 1 et moins.
- Tous les monoïdes dérivés des boucles d'ordre 8 possédant une sous-boucle normale d'ordre 2 ont été calculés et présentés à la Section 5.2.

Chapitre 6

Conclusion

Ce travail a porté sur l'étude du monoïde dérivé des boucles finies. Nous avons vu que l'introduction des monoïdes finis en théorie des automates par M.P. Schützenberger a permis aux chercheurs de prendre une nouvelle direction dans l'étude des langages réguliers. L'utilisation de cette structure algébrique associative a notamment permis de caractériser algébriquement les langages réguliers sans étoile qui, comme nous le savons maintenant, correspondent aux langages dont le monoïde syntactique est apériodique. L'étude des structures algébriques non associatives, plus particulièrement des boucles finies, est alors une extension naturelle de ces élégants travaux.

Après avoir défini la notion de monoïde dérivé d'une boucle finie, nous avons vu que celui-ci reconnaît tous les langages qui sont reconnus par la boucle à partir de laquelle il a été calculé. Ainsi, le monoïde dérivé devient un outil permettant d'utiliser la richesse de la théorie des monoïdes dans l'étude des boucles finies.

Cet ouvrage avait deux objectifs principaux. Le premier objectif était de démontrer que le monoïde dérivé d'une boucle finie est calculable. Le deuxième était de trouver un algorithme efficace permettant de calculer le monoïde dérivé des boucles de petit ordre (8 et moins). On se souvient que seules les boucles contenant peu d'éléments sont considérées car il existe plus de 9 365 milliards de classes d'isomorphismes parmi les boucles d'ordre 9 et la taille de leur monoïde dérivé peut être doublement exponentielle.

L'atteinte du premier objectif a nécessité la démonstration de plusieurs propositions. En premier lieu, nous avons prouvé que le monoïde dérivé d'une boucle finie est lui aussi fini. Par la suite, nous avons démontré que la longueur des contextes d'évaluation qu'il est nécessaire d'utiliser pour calculer le monoïde dérivé est bornée supérieurement par deux fois l'ordre de la boucle moins un. Cela a nécessité l'introduction de plusieurs concepts. Ainsi, nous avons notamment définit et étudié les principales propriétés des arbres binaires et des arbres pointés. Ces deux propositions réunies nous ont permis de démontrer que la quantité d'opérations nécessaires pour effectuer le calcul du monoïde dérivé d'une boucle finie est finie, donc que celui-ci est calculable. Cela nous a également permis de donner une borne supérieure sur le nombre d'éléments du monoïde dérivé, mais cette borne pourrait être raffinée.

Par la suite, nous avons présenté la première version d'un algorithme ayant pour objectif de calculer le monoïde dérivé d'une boucle finie ($Alg.v1$). La seconde version de cet algorithme ($Alg.v2$) présente une amélioration de la fonction de multiplication de deux représentants. Nous avons démontré que ces deux versions calculent toujours le monoïde dérivé. Par contre, nos analyses temporelles ont démontré qu'elles sont en pratique inutilisables car la quantité de contextes d'évaluation à utiliser est exponentielle par rapport à l'ordre de la boucle.

Ainsi, nous avons présenté trois solutions pour diminuer la quantité de contextes d'évaluation utilisés. Toutefois, nous n'avons pas démontré que le monoïde calculé à l'aide de ces solutions correspond à chaque fois au monoïde dérivé. La première solution, intégrée à la version $Alg.v3$ de l'algorithme, consiste à utiliser comme contextes d'évaluation tous les mots dont la longueur est plus petite qu'une certaine valeur fournie par l'usager (LMC). En faisant varier la valeur de LMC selon la boucle utilisée, cela nous permet d'obtenir un monoïde. En particulier, nous avons démontré que cette

solution permet de calculer le monoïde dérivé des boucles d'ordre 8 et moins possédant une sous-boucle normale N, en utilisant $LMC = (2|N| - 1)$. Lorsque la valeur de LMC est beaucoup plus petite que deux fois l'ordre de la boucle, cette version est beaucoup plus rapide que $Alg.v1$ et $Alg.v2$. Naturellement, plus la valeur de LMC se rapproche de deux fois l'ordre de la boucle et moins la différence de temps entre $Alg.v3$ et $Alg.v2$ est importante, jusqu'à être nulle si les mêmes contextes d'évaluation sont utilisés.

La deuxième solution consiste à utiliser comme contextes d'évaluations les éléments de l'ensemble *Repr*, contenant les représentants calculés par l'algorithme. Cette solution a été intégrée à la version $Alg.v4$ de l'algorithme. Nous avons vu que cette version est plus rapide que $Alg.v3$ si $|D(B)| < |B|^{LMC}$, où B est la boucle concernée. Finalement, la troisième solution est une combinaison des deux premières. En effet, elle utilise un sous-ensemble variable de *Repr* comme ensemble de contextes d'évaluations. On débute en utilisant seulement l'identité de la boucle comme contexte d'évaluation. Si le résultat n'est pas associatif, on ajoute alors tous les représentants de longueur un. Si ce n'est toujours pas associatif, on ajoute les représentants de longueur deux et ainsi de suite jusqu'à l'obtention d'un monoïde, ou jusqu'à ce que tous les représentants soient utilisés. Nous retrouvons cette solution dans la dernière version ($Alg.v5$) de l'algorithme, qui est la plus rapide des trois.

Nous avons vu lors de l'analyse des résultats que, pour une boucle donnée, les trois solutions obtiennent en pratique un unique monoïde que nous avons nommé le monoïde pseudo-dérivé de la boucle, dénoté $(Repr, \bullet)$. Bien que nous n'ayons pas démontré que ce monoïde est toujours isomorphe au monoïde dérivé, nous avons démontré qu'il contient au plus le même nombre d'éléments que celui-ci. Une question importante est alors de savoir si $\forall w \in B^*$, $eval(w) \supseteq eval(w_1 \bullet w_2 \cdots \bullet w_k)$, où les w_i sont des lettres de la boucle. La finalisation de cette preuve représenterait un apport important à nos travaux

puisqu'elle permettrait de démontrer que le monoïde dérivé et le monoïde pseudo-dérivé sont toujours isomorphes.

Nous avons vu que la version *Alg.v5* de l'algorithme procède de façon itérative pour calculer le monoïde pseudo-dérivé d'une boucle. À chaque itération, elle obtient un groupoïde qui a été calculé avec tous les représentants de *Repr* dont la longueur est plus petite ou égale à *Longueur_Ctx*. Elle termine lorsque le groupoïde obtenu est associatif, i.e. lorsqu'elle a calculé le monoïde pseudo-dérivé. Nous pouvons voir ce processus itératif comme une séquence de groupoïdes où nous retrouvons la boucle au départ, le monoïde pseudo-dérivé à l'arrivée et la succession de groupoïdes entre les deux. Chacun des groupoïdes est alors plus "raffiné" que le précédent. Il serait très intéressant de mieux comprendre cette séquence. Par exemple, nous ne savons pas si les groupoïdes intermédiaires reconnaissent tous les langages reconnus par la boucle. Si tel est le cas, cela démontrerait également que le monoïde pseudo-dérivé et le monoïde dérivé sont toujours isomorphes.

En conclusion, nous pensons que les informations fournies dans cet ouvrage peuvent aider à mieux comprendre le fonctionnement et la structure du monoïde dérivé. Les différentes versions de l'algorithme que nous avons présentées ont répondu à plusieurs questions concernant le monoïde dérivé. Mais principalement, elles ont démontré, en apportant de nombreuses questions, que l'étude du monoïde dérivé est un sujet de recherche intéressant et très complexe.

Annexe A

Exemples de monoïdes dérivés

Puisque la taille d'un monoïde dérivé $D(B)$ est généralement très grande par rapport à l'ordre de la boucle B, il n'est pas possible de présenter un grand nombre d'exemples à l'intérieur de ce mémoire. Aussi, nous présentons dans cet annexe cinq exemples de monoïde dérivés non triviaux de moins de 23 éléments (b6_1, b6_2, b6_81, b8_329 et b8_14665). Voici les principales caractéristiques de ces boucles, telles que présentés au Chapitre 5.

b6_1 (Figures A.1 et A.2) - Nilpotente de classe 2, résoluble de classe 2, commutative et M-Résoluble de classe 2. Son monoïde dérivé fait parti de la classe *ECOM*.

b6_2 (Figures A.3 et A.4) - Nilpotente de classe 2, résoluble de classe 2 et M-Résoluble de classe 2. Son monoïde dérivé fait parti de la classe *ECOM*.

b6_81 (Figures A.5 et A.6) - Résoluble de classe 2 et M-Résoluble de classe 2. Son monoïde dérivé fait parti de la classe *ECOM*.

b8_329 (Figures A.7 et A.8) - Nilpotente de classe 2, résoluble de classe 2, commutative, M-Nilpotente de classe 3 et M-Résoluble de classe 2. Son monoïde dérivé fait parti de la classe *ECOM*.

b8_14665 (Figures A.9 et A.10) - Nilpotente de classe 2, résoluble de classe 2, commutative, M-Nilpotente de classe 3 et M-Résoluble de classe 2. Son monoïde dérivé fait parti de la classe *ECOM*.

0	1	2	3	4	5
1	0	3	2	5	4
2	3	4	5	0	1
3	2	5	4	1	0
4	5	0	1	3	2
5	4	1	0	2	3

Figure A.1: Table de Cayley de la boucle b6_1

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22
1	0	3	2	5	4	7	6	9	8	11	10	13	12	15	14	16	18	17	19	21	20	22
2	3	6	7	8	9	14	15	16	16	17	18	19	19	16	16	19	19	19	22	8	9	16
3	2	7	6	9	8	15	14	16	16	18	17	19	19	16	16	19	19	19	22	9	8	16
4	5	10	11	12	13	16	16	20	21	19	19	15	14	19	19	22	10	11	16	16	16	19
5	4	11	10	13	12	16	16	21	20	19	19	14	15	19	19	22	11	10	16	16	16	19
6	7	14	15	16	16	16	16	19	19	19	19	22	22	19	19	22	22	22	16	16	16	19
7	6	15	14	16	16	16	16	19	19	19	19	22	22	19	19	22	22	22	16	16	16	19
8	9	17	18	19	19	19	19	8	9	22	22	16	16	22	22	16	17	18	19	19	19	22
9	8	18	17	19	19	19	19	9	8	22	22	16	16	22	22	16	18	17	19	19	19	22
10	11	16	16	20	21	19	19	22	22	10	11	16	16	22	22	16	16	16	19	20	21	22
11	10	16	16	21	20	19	19	22	22	11	10	16	16	22	22	16	16	16	19	21	20	22
12	13	19	19	15	14	22	22	16	16	16	16	19	19	16	16	19	19	19	22	22	22	16
13	12	19	19	14	15	22	22	16	16	16	16	19	19	16	16	19	19	19	22	22	22	16
14	15	16	16	19	19	19	19	22	22	22	22	16	16	22	22	16	16	16	19	19	19	22
15	14	16	16	19	19	19	19	22	22	22	22	16	16	22	22	16	16	16	19	19	19	22
16	16	19	19	22	22	22	22	16	16	16	16	19	19	16	16	19	19	19	22	22	22	16
17	18	19	19	8	9	22	22	16	16	17	18	19	19	16	16	19	19	19	22	8	9	16
18	17	19	19	9	8	22	22	16	16	18	17	19	19	16	16	19	19	19	22	9	8	16
19	19	22	22	16	16	16	16	19	19	19	19	22	22	19	19	22	22	22	16	16	16	19
20	21	10	11	16	16	16	16	20	21	19	19	22	22	19	19	22	10	11	16	16	16	19
21	20	11	10	16	16	16	16	21	20	19	19	22	22	19	19	22	11	10	16	16	16	19
22	22	16	16	19	19	19	19	22	22	22	22	16	16	22	22	16	16	16	19	19	19	22

Figure A.2: Table de Cayley du monoïde dérivé D(b6_1)

0	1	2	3	4	5
1	0	3	2	5	4
2	3	4	5	0	1
3	2	5	4	1	0
4	5	1	0	2	3
5	4	0	1	3	2

Figure A.3: Table de Cayley de la boucle b6_2

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20
1	0	3	2	5	4	7	6	9	8	11	10	13	12	14	16	15	17	19	18	20
2	3	6	7	8	9	14	14	15	16	17	17	18	19	17	20	20	20	14	14	14
3	2	7	6	9	8	14	14	16	15	17	17	19	18	17	20	20	20	14	14	14
4	5	10	11	12	13	17	17	20	20	20	20	14	14	20	14	14	14	17	17	17
5	4	11	10	13	12	17	17	20	20	20	20	14	14	20	14	14	14	17	17	17
6	7	14	14	15	16	17	17	20	20	20	20	14	14	20	14	14	14	17	17	17
7	6	14	14	16	15	17	17	20	20	20	20	14	14	20	14	14	14	17	17	17
8	9	17	17	18	19	20	20	14	14	14	14	17	17	14	17	17	17	20	20	20
9	8	17	17	19	18	20	20	14	14	14	14	17	17	14	17	17	17	20	20	20
10	11	17	17	20	20	20	20	14	14	14	14	17	17	14	17	17	17	20	20	20
11	10	17	17	20	20	20	20	14	14	14	14	17	17	14	17	17	17	20	20	20
12	13	20	20	14	14	14	14	17	17	17	17	20	20	17	20	20	20	14	14	14
13	12	20	20	14	14	14	14	17	17	17	17	20	20	17	20	20	20	14	14	14
14	14	17	17	20	20	20	20	14	14	14	14	17	17	14	17	17	17	20	20	20
15	16	20	20	14	14	14	14	17	17	17	17	20	20	17	20	20	20	14	14	14
16	15	20	20	14	14	14	14	17	17	17	17	20	20	17	20	20	20	14	14	14
17	17	20	20	14	14	14	14	17	17	17	17	20	20	17	20	20	20	14	14	14
18	19	14	14	17	17	17	17	20	20	20	20	14	14	20	14	14	14	17	17	17
19	18	14	14	17	17	17	17	20	20	20	20	14	14	20	14	14	14	17	17	17
20	20	14	14	17	17	17	17	20	20	20	20	14	14	20	14	14	14	17	17	17

Figure A.4: Table de Cayley du monoïde dérivé D(b6_2)

0	1	2	3	4	5
1	2	0	4	5	3
2	0	1	5	3	4
3	5	4	1	0	2
4	3	5	2	1	0
5	4	3	0	2	1

Figure A.5: Table de Cayley de la boucle b6_81

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
1	2	0	4	5	3	8	6	7	10	11	9	14	12	13	15	16
2	0	1	5	3	4	7	8	6	11	9	10	13	14	12	15	16
3	5	4	6	7	8	9	10	11	12	13	14	15	15	15	16	15
4	3	5	8	6	7	10	11	9	14	12	13	15	15	15	16	15
5	4	3	7	8	6	11	9	10	13	14	12	15	15	15	16	15
6	8	7	9	10	11	12	13	14	15	15	15	16	16	16	15	16
7	6	8	11	9	10	13	14	12	15	15	15	16	16	16	15	16
8	7	6	10	11	9	14	12	13	15	15	15	16	16	16	15	16
9	11	10	12	13	14	15	15	15	16	16	16	15	15	15	16	15
10	9	11	14	12	13	15	15	15	16	16	16	15	15	15	16	15
11	10	9	13	14	12	15	15	15	16	16	16	15	15	15	16	15
12	14	13	15	15	15	16	16	16	15	15	15	16	16	16	15	16
13	12	14	15	15	15	16	16	16	15	15	15	16	16	16	15	16
14	13	12	15	15	15	16	16	16	15	15	15	16	16	16	15	16
15	15	15	16	16	16	15	15	15	16	16	16	15	15	15	16	15
16	16	16	15	15	15	16	16	16	15	15	15	16	16	16	15	16

Figure A.6: Table de Cayley du monoïde dérivé D(b6_81)

0	1	2	3	4	5	6	7
1	0	3	2	5	4	7	6
2	3	0	1	6	7	4	5
3	2	1	0	7	6	5	4
4	5	7	6	2	3	1	0
5	4	6	7	3	2	0	1
6	7	5	4	0	1	3	2
7	6	4	5	1	0	2	3

Figure A.7: Table de Cayley de la boucle b8_329

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
1	0	3	2	5	4	7	6	9	8	11	10	12	13	14	15
2	3	0	1	6	7	4	5	11	10	9	8	13	12	15	14
3	2	1	0	7	6	5	4	10	11	8	9	13	12	15	14
4	5	7	6	8	9	10	11	12	12	13	13	14	15	13	12
5	4	6	7	9	8	11	10	12	12	13	13	14	15	13	12
6	7	5	4	11	10	9	8	13	13	12	12	15	14	12	13
7	6	4	5	10	11	8	9	13	13	12	12	15	14	12	13
8	9	11	10	12	12	13	13	14	14	15	15	13	12	15	14
9	8	10	11	12	12	13	13	14	14	15	15	13	12	15	14
10	11	9	8	13	13	12	12	15	15	14	14	12	13	14	15
11	10	8	9	13	13	12	12	15	15	14	14	12	13	14	15
12	12	13	13	14	14	15	15	13	13	12	12	15	14	12	13
13	13	12	12	15	15	14	14	12	12	13	13	14	15	13	12
14	14	15	15	13	13	12	12	15	15	14	14	12	13	14	15
15	15	14	14	12	12	13	13	14	14	15	15	13	12	15	14

Figure A.8: Table de Cayley du monoïde dérivé D(b8_329)

0	1	2	3	4	5	6	7
1	0	3	2	5	4	7	6
2	3	1	0	6	7	5	4
3	2	0	1	7	6	4	5
4	5	7	6	2	3	0	1
5	4	6	7	3	2	1	0
6	7	4	5	1	0	2	3
7	6	5	4	0	1	3	2

Figure A.9: Table de Cayley de la boucle b8_14665

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
1	0	3	2	5	4	7	6	9	8	11	10	12	13	14	15
2	3	1	0	6	7	5	4	11	10	8	9	13	12	15	14
3	2	0	1	7	6	4	5	10	11	9	8	13	12	15	14
4	5	7	6	8	9	10	11	12	12	13	13	14	15	13	12
5	4	6	7	9	8	11	10	12	12	13	13	14	15	13	12
6	7	4	5	11	10	8	9	13	13	12	12	15	14	12	13
7	6	5	4	10	11	9	8	13	13	12	12	15	14	12	13
8	9	11	10	12	12	13	13	14	14	15	15	13	12	15	14
9	8	10	11	12	12	13	13	14	14	15	15	13	12	15	14
10	11	8	9	13	13	12	12	15	15	14	14	12	13	14	15
11	10	9	8	13	13	12	12	15	15	14	14	12	13	14	15
12	12	13	13	14	14	15	15	13	13	12	12	15	14	12	13
13	13	12	12	15	15	14	14	12	12	13	13	14	15	13	12
14	14	15	15	13	13	12	12	15	15	14	14	12	13	14	15
15	15	14	14	12	12	13	13	14	14	15	15	13	12	15	14

Figure A.10: Table de Cayley du monoïde dérivé D(b8_14665)

Bibliographie

[1] Albert, A. A. Quasigroups. I. *Transactions of the American Mathematical Society 54* (1943), 507–519 .

[2] Albert, A. A. Quasigroups. II. *Transactions of the American Mathematical Society 55* (1944), 401–419 .

[3] Barrington, D. A. M. Bounded-width polynomial size branching programs recognize exactly those languages in NC^1. *Journal of Computer and System Sciences 38*, 1 (1989), 150–164 .

[4] Barrington, D. A. M., and Thérien, D. Finite monoids and the fine structure of $NC1$. *J. ACM 35*, 4 (1988), 941–952 .

[5] Bédard, F., Lemieux, F., and McKenzie, P. Extensions to Barrington's M-program model. *Theoretical Computer Science 107*, 1 (1993), 31–61 .

[6] Beaudry, M., Lemieux, F., and Thérien, D. Groupoids that recognize only regular languages. In *ICALP* (2005), L. Caires, G. F. Italiano, L. Monteiro, C. Palamidessi, and M. Yung, Eds., vol. 3580 of *Lecture Notes in Computer Science*, Springer, pp. 421–433 .

[7] Beaudry, M., Lemieux, F., and Thérien, D. Finite loops recognize exactly the regular open languages. *IEEE Conference on Computational Complexity* (1997), 110–120 .

[8] Beaudry, M., Lemieux, F., and Thérien, D. Star-free open languages and aperiodic loops. In *STACS '01 : Proceedings of the 18th Annual Symposium on Theoretical Aspects of Computer Science* (London, UK, 2001), Springer-Verlag, pp. 87–98 .

[9] BERSTEL, J. *Tranductions and context-free languages.* Teubner, Stuttgart, 1979 .

[10] BRUCK, R. H. Contributions to the theory of loops. *Transactions of the American Mathematical Society 60* (1946), 245–354 .

[11] BRUCK, R. H. *A survey of binary systems.* Springer-Verlag, 1971 .

[12] CAUSSINUS, H., AND LEMIEUX, F. The complexity of computing over quasigroups. In *Proceedings of the 14th Conference on Foundations of Software Technology and Theoretical Computer Science* (London, UK, 1994), Springer-Verlag, pp. 36–47 .

[13] CLIFFORD, A. H., AND PRESTON, G. B. *The algebraic theory of semigroups. Vol. I.* Mathematical Surveys, No. 7. American Mathematical Society, Providence, R.I., 1961 .

[14] COOK, S. A. Characterizations of pushdown machines in terms of time-bounded computers. *J. ACM 18*, 1 (1971), 4–18 .

[15] DÉNES, J., AND KEEDWELL, A. *Latin Squares and their Applications.* English university Press, 1974 .

[16] EILENBERG, S. *Automata, Languages and Machines.* Academic Press, 1974 .

[17] FROIDURE, V., AND PIN, J.-E. Algorithms for computing finite semigroups. In *FoCM '97 : Selected papers of a conference on Foundations of computational mathematics* (New York,, 1997), Springer-Verlag New York, Inc., pp. 112–126 .

[18] GUÉRIN, P. *Génération des classes d'isomorphisme des boucles d'ordre 8.* Thèse de maîtrise, Université du Québec à Chicoutimi, 2003 .

[19] HOPCROFT, J. E., AND ULLMAN, J. D. *Introduction to Automata Theory, Languages, and Computation.* Series in Computer Science. Addison-Wesley, Reading, MA, 1979 .

[20] KLEENE, S. C. Representation of events in nerve nets and finite automata. In *Automata studies.* Princeton University Press, Princeton, N. J., 1956, pp. 3–41. Annals of mathematics studies, no. 34 .

[21] LEMIEUX, F. *Finite Groupoids and their Applications to Computational Complexity.* PhD thesis, School of computer Science, McGill University, 1996 .

[22] LIPTON, R. J., AND ZALCSTEIN, Y. Word problems solvable in logspace. *J. ACM 24*, 3 (1977), 522–526 .

[23] MACIEL, A. *Threshold Circuits of Small Majority-Depth.* PhD thesis, School of computer Science, McGill University, 1995 .

[24] McKay, B. D., and Wanless, I. M. On the number of latin squares. *Annals of combinatorics 9* (2005), 335–344 .

[25] Mezei, J., and Wright, J. B. Algebraic automata and context-free sets. *Information and Control 11*, 2-3 (1967), 3–29 .

[26] Pflugfelder, H. O. Historical notes on loop theory. *Comment. Math. Univ. Carolinae 41,2* (2000), 359–370 .

[27] Pin, J.-E. *Variétés de Langages Formels.* Masson, Paris, 1984 .

[28] Pin, J.-E. Polynomial closure of group languages and open sets of the Hall topology. *Theoretical Computer Science 169*, 2 (1996), 185–200 .

[29] Schützenberger, M.-P. Une théorie algébrique du codage. *Séminaire Dubreil-Pisot. Algèbre et théorie des nombres* (1955-1956), exposé n^o 15 .

[30] Schützenberger, M.-P. On finite monoids having only trivial subgroups. *Information and Control 8* (1965), 190–194 .

[31] Shannon, C. A symbolic analysis of relay and switching circuits. *Transactions AIEE 57* (1938), 59–98 .

[32] Sipser, M. *Introduction to the Theory of Computation.* International Thomson Publishing, 1996 .

[33] Straubing, H. *Finite Automata, Formal Logic, and Circuit Complexity.* Birkhäuser, Boston, MA, 1994 .