RÉPUBLIQUE ALGÉRIENNE DÉMOCRATIQUE ET POPULAIRE

MINISTÈRE D'ENSEIGNEMENT SUPÉRIEUR ET DE LA RECHERCHE SCIENTIFIQUE

UNIVERSITÉ AMAR TELIDJI LAGHOUAT

FACULTÉ DES SCIENCES
Département : Mathématiques et informatique
École doctorale STIC

Filière : Informatique
Option : Informatique Répartie et Mobile

Mémoire présenté en vue d'obtention du diplôme de Magister en informatique

Thème :

Recherche de motifs d'arbre

Présenté par Par Mlle BELABBACI Ahlem

Devant le jury composé de :

M Y. Ouinten	MC	Université de Laghouat	Président
M M. Yagoubi	Professeur	Université de Laghouat	Examinateur
M N. Lagraa	MC	Université de Laghouat	Examinateur
Mme H. Cherroun	MC	Université de Laghouat	Rapporteur
M Y.Guellouma	MA	Université de Laghouat	Co-encadreur

Année universitaire 2013/2014

Remerciements.

Tout d'abord, je tiens à exprimer ma profonde gratitude à Mr D. Ziadi, Mme H. Bouzouad et Mr Y. Guellouma de m'avoir proposé ce sujet, dirigé et conseillé tout au long de ce travail.

J'exprime mes sincères remerciements à Mr Y. Ouinten de m'avoir fait l'honneur de présider le jury de soutenance, ainsi qu'à Mr M. Yagoubi et Mr N. Lagraa pour avoir accepté d'examiner et d'évaluer ce travail.

Je remercie également toutes les personnes qui m'ont aidé et encouragé.

Enfin, que tous mes enseignants aussi bien de mon cursus scolaire qu'universitaire soient remerciés.

A ma famille, mes amis.

Résumé

Les travaux de la présente recherche s'inscrivent dans le domaine des langages réguliers d'arbre. Plus spécialement, nous nous sommes intéressés au problème de recherche de motifs d'arbre. Ces langages avec leurs différentes représentations forment de puissants outils pour des applications informatiques et scientifiques.

La recherche de motif d'arbre permet d'identifier les occurrences d'un motif d'arbre dans un arbre objet. Ce problème est une extension de celui des mots.

L'étude bibliographique révèle que très peu de travaux ont été guidés dans le sens de la recherche de motifs d'arbres à partir d'une expression régulière d'arbre. Nous proposons une généralisation de l'approche de Thompson pour la recherche de motifs de mots aux arbres. En effet, notre généralisation est accomplie en deux étapes : premièrement, nous avons construit un automate d'arbre pour l'expression régulière du motif. Ensuite, nous avons conçu un algorithme de recherche de motifs d'arbre utilisant cet automate.

Mots clé : expression régulière d'arbre, automate d'arbre, recherche de motif, automate de Thompson.

Abstract

In this work we deal with regular tree languages. More specifically we are interested in the problem of tree pattern matching. These languages with their different representations are powerful tools for computing and scientific applications.

The tree pattern matching problem identifies occurrences of a tree pattern in an object tree. This problem is an extension of the pattern matching on strings.

The literature review reveals that a very little work has been done in the field of the tree pattern matching from a regular tree expression. We propose a generalization of the approach of Thompson for words pattern matching to trees. Indeed, our generalization is accomplished in two steps : first, we build a tree automaton from the regular tree expression of the pattern. Then we design an algorithm of pattern matching using this tree automaton.

Keywords : regular tree expression, tree automata, pattern matching, Thompson's automaton.

Table des matières

Remerciements **i**

Résumé **iii**

Abstract **iii**

Table des matières **v**

Table des figures **vii**

Introduction générale **1**

1 Langages réguliers d'arbre **7**
1.1 Notations et préliminaires 8
1.2 Les langages d'arbre 13
1.2.1 Propriété de fermeture des langages réguliers d'arbre. 13
1.3 Représentation des langages réguliers d'arbre 13
1.3.1 Les automates d'arbre finis 14
1.3.2 Les expressions régulières d'arbre 18
1.3.3 Les grammaires régulières d'arbre 19
1.4 Problèmes des langages réguliers d'arbre résolus par les automates 22
1.5 Applications des langages réguliers d'arbre 23
1.6 Conclusion 23

2 Recherche de motifs d'arbre **24**
2.1 Définition du problème 25
2.2 Algorithmes de recherche de motifs 26
2.2.1 L'algorithme naïf de recherche de motifs 28
2.2.2 Algorithme de Hoffmann et O'Donnell (1982)et ses variantes 29
2.2.3 Algorithme de Cleophas et al. (2005) 34
2.2.4 Algorithme de Kosaraju (1989) 35
2.2.5 Algorithme de Dubiner et al. (1994) 37

2.2.6 Algorithme de Chase (1987) 38
2.2.7 Algorithme de Ramesh et Ramakrishnan (1992) 42
2.2.8 Algorithme de Cole et al. (1999, 2003) 45
2.2.9 Algorithme de Polách et al. (2011) 46
2.3 Synthèse et Discussion 51
2.4 Conclusion 54

3 Notre approche de recherche de motifs d'arbre 56
3.1 Vue générale de l'approche proposée 57
3.2 Rappel de l'automate de Thompson pour les mots 58
3.2.1 Construction de l'automate de Thompson 58
3.2.2 Exemple d'un automate de Thompson de mots 60
3.2.3 Discussion de l'algorithme de Thompson pour les mots 61
3.3 Notre généralisation de la technique de Thompson pour les arbres 61
3.3.1 Forme générale de l'automate de Thompson d'arbre . 62
3.3.2 Construction de l'automate 62
3.4 Preuves d'équivalences entre les constructions et les ER . . . 67
3.4.1 Preuve du Lemme 1 68
3.4.2 Preuve du Lemme 2 71
3.5 L'algorithme de recherche de motifs proposé 73
3.5.1 Construction de l'automate descendant de RMA . . . 73
3.5.2 Construction de l'automate ascendant de RM 74
3.5.3 Algorithme ascendant de reconnaissance de motifs . . 75
3.6 Complexité de l'algorithme proposé 79
3.7 Conclusion 79

Conclusion et perspectives 81

Bibliographie 87

Table des figures

1.1 Un arbre étiqueté ordonné. 9
1.2 Représentation graphique d'un arbre et son domaine. 10
1.3 Première forme de substitution. 12
1.4 Deuxième forme de substitution. 12
1.5 Troisième forme de substitution. 12
1.6 Quatrième forme de substitution. 12
1.7 Exemples de transitions simples dans un automate d'arbre. . 15
1.8 Exemple de transition multiple dans un automate d'arbre. . . 15
1.9 Exemple d'un automate ascendant. 16
1.10 Exemple d'un automate descendant. 17
1.11 Relations entre les différents types d'automates d'arbre. . . . 18
1.12 Un automate d'arbre avec un état atteignable mais inutile (q_2). 22

2.1 Exemple d'un motif et d'un arbre correspondant à ce motif. . 26
2.2 Classification des algorithmes de RMA présentés selon la méthode utilisée. 28
2.3 Classification des algorithmes de RMA présentés selon l'entrée de l'algorithme. 28
2.4 Attribution des codes pour l'algorithme descendant. 31
2.5 Le motif P (a) et son arborescence associée (b). 32
2.6 L'automate de recherche de motifs construit pour le motif P. 33
2.7 Arborescence pour le motif P. 34
2.8 Automate d'Aho-Corasick pour le motif P 34
2.9 Exemple d'un arbre et son arbre de suffixe. 36
2.10 Exemple d'arbre ordonné. 43
2.11 Conversion de ε 47
2.12 Conversion de $a \in \Sigma$ 47
2.13 Conversion de | 47
2.14 Automate de l'union 48
2.15 Automate t_1 et t_2. 48
2.16 Automate de l'union $t_1 \cup t_2$. 48
2.17 Automate déterministe de l'union $t_1 \cup t_2$. 49
2.18 La concaténation de deux automates 49
2.19 Automate de l'opération d'itération. 50

3.1 Principe de notre approche de RMA. 57
3.2 Automate de Thompson du langage vide. 59
3.3 Automate de Thompson du mot vide. 60
3.4 Automate de Thompson d'un caractère a. 60
3.5 Automate de Thompson de l'union $E + F$. 60
3.6 Automate de Thompson de la concaténation $E.F$. 60
3.7 Automate de Thompson de l'étoile E^*. 60
3.8 Exemple d'un automate de Thompson pour les mots. 61
3.9 Forme générale d'un automate de Thompson pour les arbres. 62
3.10 Automate de Thompson de l'arbre feuille. 63
3.11 Automate de Thompson de la fonction d'arité. 64
3.12 Automate de Thompson de l'union de deux automates. 65
3.13 Automate de Thompson de la concaténation de deux automates. 66
3.14 Automate de Thompson de la fermeture d'un automate. . . . 67
3.15 Automate descendant pour la RMA de l'expression E. 74
3.16 Automate ascendant pour la RMA de l'expression E 75
3.17 Exemple de déroulement de notre algorithme de RMA. 78

Introduction générale

En théorie des langages formels, un langage régulier est un langage formel qui peut être exprimé à l'aide d'une expression régulière, généré par une grammaire régulière ou reconnu par un automate fini. L'histoire de la façon dont les automates finis sont devenus une branche de la science informatique illustre sa large gamme d'applications. Les premiers chercheurs à avoir abordé le concept d'une machine à états finis formaient une équipe de biologistes, de psychologues, de mathématiciens, d'ingénieurs. Ils partageaient tous un intérêt commun : modéliser le processus de la pensée humaine, que ce soit dans le cerveau ou dans un ordinateur. Warren McCulloch et Walter Pitts, deux neurophysiologues, ont été les premiers à présenter une description des automates finis. Leur article [MP88] a fait d'importantes contributions à l'étude de la théorie des réseaux de neurones, théorie des automates, la théorie du calcul et de la cybernétique. Plus tard, deux informaticiens, Mealy et Moore, ont généralisé la théorie à des machines plus puissantes dans des documents distincts [Mea55][E.F56].

En 1956, le mathématicien Stephen Kleene a développé ce modèle. Dans son article [Kle56], il présente aussi une algèbre simple. C'est ainsi que les termes ensembles réguliers et expressions régulières sont nés. En 1968, le pionnier d'Unix, Ken Thompson a publié son article [Tho68] où il décrit un compilateur d'expressions régulières qui sera utilisé pour créer le code objet de la machine IBM 7094. Il a également mis en place la notation de Kleene dans l'éditeur QED. L'objectif était de permettre à un utilisateur de faire des recherches de motifs dans des fichiers texte.

La théorie des automates est apparue pour résoudre des problèmes aussi bien pratiques que théoriques. Désormais, les automates font partie des notions fondamentales de l'informatique, et se retrouvent dans la plupart des logiciels. Étant donné un ensemble fini de symboles qu'on appelle alphabet, un automate fini est un graphe fini orienté et étiqueté par des symboles. La structure de graphe permet de décrire les chemins auxquels on associe des séquences de symboles. Si on définit un mot comme une séquence de symboles et un langage comme un ensemble de mots, alors à tout automate fini on associe un langage. Les automates finis permettent donc de définir

des langages de manière formelle. Plus généralement, l'étude des langages sous cet aspect très formel d'ensembles de mots est appelée la théorie des langages formels.

Les automates sont un outil fondamental tant sur le plan théorique que sur le plan pratique, on peut dire qu'ils sont omniprésents ; on les retrouve notamment dans les domaines suivants : description de circuits, langage de programmation (compilation), vérification (model checking), logique (avec la logique monadique de second ordre), algèbre (algèbres de Kleene), recherche de motif dans des textes et même dans les systèmes d'exploitation. En effet, le système d'exploitation Unix a été développé par des spécialistes de la théorie des automates tel que Ken Thompson. Un autre exemple important d'utilisation des automates finis est illustré par la commande 'grep' qui effectue la recherche de motif dans un texte. Cette fonctionnalité se retrouve implémentée d'une manière ou d'une autre dans de nombreux logiciels. Le succès de cette fonctionnalité a fait que les expressions régulières sont devenues un paradigme pour les programmes de manipulation de textes tels que sed, mais aussi pour les langages de script tels que Perl, Python et Ruby.

Dans ces outils, les automates ne font pas seulement que reconnaître si un mot appartient au langage mais ils traduisent un mot en même temps qu'ils le reconnaissent ; on appelle de tels automates des transducteurs.

L'étude de la classe des langages qui sont reconnaissables par des automates finis a débuté avec l'article de Kleene [Kle56]. Le théorème fondamental de Kleene énonce l'équivalence de la classe des langages reconnaissables (qu'on peut définir par automate fini) et ceux qu'on peut définir par une expression régulière. Ce théorème permet de bien identifier cette classe de langages qu'on appelle les langages réguliers.

Les expressions régulières sont des expressions algébriques définies récursivement à partir de symboles et les trois opérations suivantes : union, concaténation et fermeture de Kleene. Ces opérations sont suffisantes pour définir tous les langages qu'on peut reconnaître par un automate fini et l'avantage des expressions régulières sur les automates est qu'elles sont plus pratiques à manipuler parce qu'elles sont plus algébriques.

Cette omniprésence des expressions régulières a fait que les techniques pour les compiler en automates n'ont cessé de progresser. Parmi les articles historiques on peut mentionner ceux de McNaughton et Yamada [MY60] et de Glushkov [Glu61]. Ce sont les premiers articles détaillant des procédures effectives pour synthétiser des automates à partir d'expressions régulières. Sans oublier l'article de Thompson [Tho68] qui est à l'origine de la commande 'grep'.

Après le développement de la théorie des langages formels et les langages des mots dans les années $1950-1960$, des chercheurs ont commencé à généraliser les principes de ces domaines. Parmi ces généralisations figure celle des arbres. Dans les années $1960 - 1970$ plusieurs travaux ont été publiés notamment sur les langages réguliers d'arbres.

Les automates d'arbres sont apparus il y a longtemps dans le contexte de la vérification de circuit. Beaucoup de chercheurs ont contribué à ce domaine qui a été initié par Church [Chu63] à la fin des années 50 et le début des années 60. Parmi ces publications nous mentionnons Trakhtenbrot [Tra95], Büchi [Büc62], Rabin [Rab69], Doner [Don70] [Don65] et Thatcher [Tha67] [Tha70] [Tha73], etc.

Par là, de nouvelles idées se sont développées, par exemple, les liens entre les automates et logique, d'ailleurs les automates d'arbre sont apparus pour la première fois dans ce cadre. Dans les années 70, de nouveaux résultats ont été établis concernant les automates d'arbre [CDG+08].

Problématique et objectifs

Le domaine du problème traité via cette recherche est les langages réguliers d'arbre, plus précisément les termes, c'est à dire les arbres étiquetés, ordonnés avec arité. Nous voulons dire par 'étiqueté' que chaque nœud dans l'arbre contient un symbole comme étiquette. Quand à l'arité, elle détermine le nombre de fils d'un nœud de l'arbre. Dans un arbre avec arité nous ne pouvons avoir deux nœuds différents étiquetés par le même symbole avec un nombre de fils différent.

De nombreuses structures en informatique peuvent être représentées par des arbres : arbres de dérivation, traductions dirigées par la syntaxe, la recherche dans les fichiers, etc. Il a été ainsi développé, une théorie des langages d'arbres reconnaissables, les grammaires d'arbres et des transducteurs d'arbres. Les arbres sont l'une des structures de données hiérarchiques fondamentales utilisés en informatique. De nombreuses méthodes ont été décrites pour résoudre les divers problèmes liés aux arbres. Cependant, la plupart d'entre eux manquent de références claires à une approche systématique de la théorie de mise en langages, des grammaires et des automates.

La théorie des langages réguliers d'arbre est très riche et trouve un large champ d'applications. Une bonne partie de cette théorie est une généralisation des mots ainsi que la relation entre les deux théories. Cette théorie a depuis été utile dans un large éventail de domaines, par exemple problèmes de décision en logique, la théorie des langages formels et la théorie de programmation...

Parmi les domaines d'application de la théorie des langages réguliers d'arbres nous trouvons la génération du code en compilation et particulière-

ment la sélection et optimisation d'instructions, la réécriture des termes, la génétique et particulièrement l'analyse de la structure du ARN, le traitement de document XML, et la vérification, notamment les protocoles réseaux et de sécurité.

La théorie des langages formels d'arbres a été largement étudiée et développée depuis 1960. Les modèles de calcul de cette théorie existent pour différents types d'arbres d'automate, qui représentent la généralisation des automates sur des chaînes aux automates sur les arbres. Les types les plus recherchés des automates d'arbres sont les automates d'arbres finis, qui reconnaissent langages d'arbres réguliers, et leur mise en œuvre est basée sur les procédures récursives.

La recherche de motif est un problème fondamental en informatique. Il s'agit de détecter dans une structure la présence de sous-structures d'un type donné : les motifs. Les structures considérées sont le plus souvent des mots, des arbres ou des graphes. Tester l'apparition d'un motif, compter les occurrences de ce motif, déterminer l'ensemble des positions de ces occurrences sont autant de questions centrales dans plusieurs branches de l'informatique. Elles ont été largement étudiées selon différentes approches : algorithmique, analytique, probabiliste.

De nombreux algorithmes existent pour les traiter. La sortie d'un algorithme de recherche de motifs peut être de plusieurs formes : l'apparition ou non du motif, le nombre d'occurrences, l'ensemble des positions d'occurrences du motif, etc. Il existe deux types de recherche de motif : combinatoire et spatiale. Parmi les exemples de la recherche de motif combinatoire nous retrouvons la recherche de motif de mots, d'ADN, ... La recherche de motif spatiale consiste à retrouver la correspondance entre deux images d'intensité donnée ou deux modèles géométriques.

Dans ce travail, nous focalisons sur le problème de recherche de motifs d'arbre ordonnés et étiquetés. La recherche de motif d'arbre, notée RMA, est considérée comme une étape essentielle dans plusieurs tâches de programmation telle que la modélisation d'interpréteurs pour les langages de programmation non procéduraux ou bien l'implémentation automatique de types abstraits de données ou encore l'optimisation de codes pour les compilateurs et la preuve automatique de théorème [AG85].

Dans la réécriture de termes, la recherche de motif d'arbre peut être utilisée pour retrouver les occurrences des parties gauches des règles de réécriture. Pour la sélection d'instructions, une grammaire régulière d'arbre est utilisée où chaque règle de production est associée à une instruction. La sélection d'instructions correspond alors à la résolution du problème de la représentation intermédiaire qui est l'arbre grammatical. Ce problème est comparable à celui de la reconnaissance (tree acceptance) ; car il s'agit de

déterminer comment un arbre donné est généré par une grammaire régulière [Cle08].

Plusieurs algorithmes de RMA sont apparus dans la littérature. L'étude de ce problème a été introduite par Hoffmann et O'Donnell en 1982. Dans certains travaux la recherche de motif dans les arbres est considérée comme une extension de celle des mots. L'algorithme le plus connu de recherche de motif dans les mots est celui de Thompson (1968). Il consiste à construire, par induction, un automate à partir de l'expression régulière du motif, cet automate est ensuite utilisé pour retrouver les différentes instances du motif. Depuis l'algorithme de Hoffmann et O'Donnell, de nombreux algorithmes de RMA sont apparus utilisant différentes techniques.

Les algorithmes de RMA qui existent dans la littérature ont une complexité théorique acceptable, mais le problème avec ces algorithmes est leurs implémentations. Elle est souvent difficile et nécessite un prétraitement considérable et des structures de données complexes.

Contribution

Notre contribution dans la RMA est une généralisation de l'algorithme de Thompson pour les mots aux arbres. En effet, nous avons procédé en deux étapes :

1. La création d'un automate de Thompson pour les arbres à partir d'une expression régulière d'arbre.
2. L'automate construit est ensuite parcouru afin de retrouver les différents sous-arbres qui correspondent aux motifs.

Structure du mémoire

Le présent mémoire est structuré comme suit :

Premier chapitre une introduction aux langages réguliers d'arbre est présentée avec ses différentes représentations, à savoir les automates, les expressions régulières et les grammaires régulières. Enfin, une discussion des principaux problèmes dans la littérature scientifique qui sont résolus par les automates d'arbre est présentée ainsi que les domaines d'application de ces derniers.

Deuxième chapitre présentation du problème de RMA et la description de quelques algorithmes de RMA parmi les plus connus, avec les techniques de base utilisées par ces algorithmes et leurs complexités engendrées. La dernière section du chapitre est une synthèse et discussion des différents algorithmes présentés.

Troisième chapitre notre contribution dans la RMA est exhibée. Avant de développer les principes de base de notre construction ainsi que son algorithme, une description sommaire de notre approche de RMA est donnée ensuite un rappel sur l'automate de Thompson pour les mots est présenté. La dernière section de ce chapitre est réservée à la discussion de la complexité de notre algorithme.

Chapitre 1

Langages réguliers d'arbre

Dans ce chapitre nous introduisons les langages réguliers d'arbre où nous présentons les différentes représentations des langages réguliers d'arbre, à savoir les automates, les expressions régulières et les grammaires régulières. Enfin, les principaux problèmes dans la littérature scientifique qui sont résolus avec les automates d'arbre sont discutés ainsi que les domaines d'application de ces derniers.

Avant d'entamer le principe des langages réguliers, nous fixons les notations utilisées, les concepts et les définitions nécessaires. Ces définitions et concepts sont rappelés tant pour les mots que pour les arbres. Tous les concepts sur les arbres présentés dans la section suivante sont plus ou moins une généralisation de concepts similaires dans le domaine des langages réguliers des mots. Par exemple le concept de grammaire des arbres est comparable à celui des mots. Cette relation 'arbre–mot' peut être utilisée pour clarifier le concept d'arbre.

1.1 Notations et préliminaires

Nous utilisons les conventions de notation suivantes :

- Σ pour l'alphabet des terminaux et N pour les non-terminaux ;
- $a, ..., e$ pour les symboles terminaux et $A, ..., E, S$ pour les non-terminaux ;
- G pour les grammaires ;
- $\mathcal{L}$ pour les langages ;
- $i, ..., n$ pour les nombres entiers ;
- $\mathcal{A}$ pour les automates finis d'arbre ;
- q pour les états et Q pour l'ensemble d'états ;
- σ pour les fonctions de transitions ;
- v pour les variables pouvant être un arbre.

Un alphabet est un ensemble fini, non vide. Les éléments d'un alphabet sont appelés symboles. Pour un alphabet Σ, nous appelons les éléments Σ^* des mots (words ou strings) sur Σ. Tout sous-ensemble de $\mathcal{P}(\Sigma^*)$ est un langage de mots.

Nous utilisons ε pour dénoter le mot vide. Pour un mot w, $|w|$ représente la longueur d'un mot. $|\varepsilon| = 0$ et $|av| = 1 + |v|$ tel que $a \in \Sigma$ et $v \in \Sigma^*$ [Cle08].

Un arbre est une collection (éventuellement vide) de nœuds et d'arêtes assujettis à certaines conditions : un nœud peut porter un nom et un certain nombre d'informations pertinentes ; une arête est un lien entre deux nœuds. Plus de détails concernant la structure d'arbre, la définition formelle, les fonctions ainsi que des exemples sont présentés ci-dessous.

Un langage est l'ensemble des mots définis par ce langage. Une expression régulière est une notation pour décrire un langage régulier. Pour un alphabet Σ, une expression régulière pour les mots est définie par :

– Les éléments de Σ, ε et $\emptyset$ sont des expressions régulières.
– Si α et β sont deux expressions régulières, alors l'union $(\alpha \cup \beta)$, la concaténation $(\alpha.\beta)$ et la fermeture α^* sont aussi des expressions régulières. ε est l'élément neutre par rapport à la concaténation et $\emptyset$ est l'élément neutre par rapport à l'union.

Dans la littérature, deux définitions des arbres apparaissent. La première définition est basée sur les domaines d'arbre et l'autre sur les termes. Nous présentons les deux définitions, qui sont d'ailleurs équivalentes, ainsi que les trois représentations des arbres. Dans ce travail, et bien que nous utilisons les vocables *arbre* et *terme* indifféremment, le vocable d'arbre est beaucoup plus utilisée.

Un arbre ordonné présente le fait que les nœuds fils d'un nœud quelconque sont ordonnés. Un nœud avec un symbole d'une arité n a n fils, où le nœud fils le plus à gauche est considéré comme le premier fils et celui le plus à droite le n^{ieme} fils. La permutation de l'ordre des fils (sous-arbres) produit un nouvel arbre si les nœuds permutés sont différents [Cle08].

Cette description de l'arbre est informelle. La description formelle est basée sur la définition du domaine d'arbre. Un domaine d'arbre décrit tous les chemins à tous les nœuds de cet arbre.

Définition 1. *Un alphabet ordonné est un couple* (Σ, r)*, où* Σ *est un ensemble fini et* r *est une application de* Σ *à* N*. L'arité d'un symbole* $a \in \Sigma$ *est* $r(a)$*. L'ensemble de symboles d'arité* p *est* Σ_p*.*

La notation $\#(a)$ est équivalente à $r(a)$.

Les éléments d'arité $0, 1, ..., p$ sont appelés respectivement constantes, unaire, binaire, ..., p-airs. Nous supposons que Σ contient au moins une constante. Dans les exemples, nous utilisons les parenthèses et les virgules pour décrire les symboles avec arité. Par exemple, $a(,)$ veut dire que a est un symbole binaire.

Définition 2. *Pour un alphabet* (Σ, r)*, l'arité maximale* R *est définie par* $R = Max\{r(a)/a \in \Sigma\}$*.*

La figure 1.1 montre un exemple d'un arbre étiqueté ordonné.

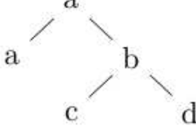


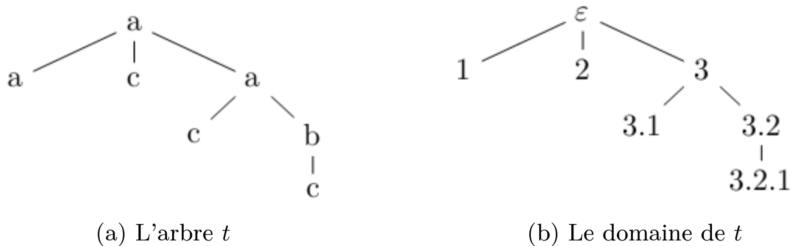
FIGURE 1.1 – Un arbre étiqueté ordonné.

Définition 3. *Étant donné E un ensemble d'étiquettes, un domaine d'arbre est un sous–ensemble D fini, non vide de E^* tel que D est fermé par rapport aux préfixes. Le domaine d'arbre représente intuitivement la structure de l'arbre.*

Par exemple, l'ensemble $\{\varepsilon, 1, 3, 1.2\}$ est un domaine d'arbre, alors que $\{1, 3, 1.2\}$ et $\{\varepsilon, 3, 1.2\}$ ne le sont pas car ils ne sont pas fermés par rapport aux préfixes.

L'ensemble $\{(\varepsilon, a), (1, a), (2, c), (3, a), (3.1, c), (3.2, b), (3.2.1, c)\}$ représente un arbre ordonné avec arité, le symbole a est d'arité 2, b est d'arité 1 et c est d'arité 0.

La figure 1.2 représente l'arbre t et son domaine.



(a) L'arbre t (b) Le domaine de t

FIGURE 1.2 – Représentation graphique d'un arbre et son domaine.

Un arbre peut être aussi représenté sous une forme parenthésée en utilisant les parenthèses pour regrouper les fils d'un même nœud et les virgules pour séparer entre eux. Par exemple, l'arbre de la figure 1.2 peut être représenté par $a(a, c, a(c, b(c)))$.

Remarque : Il existe une relation étroite entre les mots définis sur un alphabet Σ et les arbres ordonnés avec arité sur l'extension de cet alphabet. En effet, tous les symboles de Σ sont d'arité 1 et le symbole de terminaison r est d'arité 0. Le mot $w = w_1, w_2, ..., w_n$ correspond à l'arbre $w_1(w_2(...w_n)...)$.

Fonctions et opérations sur les arbres

Dimension et hauteur d'un arbre

Soit t un arbre. La **dimension** de t représente le nombre de nœuds de t, elle est notée $\|t\|$ et la **hauteur** de t indique sa profondeur, notée $\mathcal{H}(t)$. t_i est le i ième fils de t [CDG+08].

– $\mathcal{H}(t) = 0$, $\|t\| = 0$, si t est une variable ;

- $\mathcal{H}(t) = 1$, $\|t\| = 1$, si t est une feuille ;
- $\mathcal{H}(t) = 1 + max\{\mathcal{H}(t_i) | i \in \{1..n\}\}$, $\|t\| = 1 + \Sigma_{i \in \{1..n\}} \|t_i\|$, sinon.

Exemple

Considérons l'alphabet $\Sigma = \{f(3), g(2), h(1), a, b\}$, les variables $\mathcal{X} = \{x, y\}$ et les termes $t = f(g(a, b), a, h(b))$, $t' = f(g(x, y), a, g(x, y))$.

$\mathcal{H}(t) = 3$; $\mathcal{H}(t') = 2$; $\|t\| = 7$; $\|t'\| = 4$.

Les sous arbres

Définition 4. *La fonction* ***partielle*** *(infixe), notée t/n, est définie pour un arbre $t \in T(\Sigma, r)$ et une position $n \in D_t$ comme étant le sous arbre de t à partir du nœud n. Notons que $t/\varepsilon = t$.*

Le chemin (stringpath)

Un arbre peut être aussi représenté par l'ensemble de ses chemins (stringpath). Cet ensemble contient tous les chemins de la racine jusqu'aux feuilles.

Définition 5. *La fonction $SPath(t)$ pour un arbre $t \in T(\Sigma, r)$ est définie par :*

- $SPath(t) = \{t(\varepsilon)\}$ *si t est une feuille,*
- $SPath(t) = \{t(\varepsilon)\} \bigcup_{i=1...r(t(\varepsilon))} \{i\}.SPath(t/i)$ *sinon.*

Par exemple, pour $t = a(b(c), a(c, c))$, $SPath(t) = \{a1b1c, a2a1c, a2a2c\}$ et $SPath(t/2) = \{a1c, a2c\}$.

Le **rootpath** d'un nœud est le chemin (stringpath) de la racine jusqu'à ce nœud.

La substitution d'arbre

Il existe quatre formes de substitution d'arbre [Cle08] :

1. Substitution d'une seule occurrence d'un sous-arbre indiqué par sa racine par un autre arbre (figure 1.3).
2. Substitution des occurrences d'un sous-arbre par d'autres occurrences de sous-arbres (figure 1.4).
3. Multiples substitutions de plusieurs feuilles, où toutes les occurrences du même symbole sont remplacées par le même sous-arbre, cette opération est aussi appelée dans la littérature 'concaténation'. La restriction à une seule feuille est appelée produit d'arbre (figure 1.5).
4. Substitution des occurrences d'une feuille par des sous-arbres différents (figure 1.6).

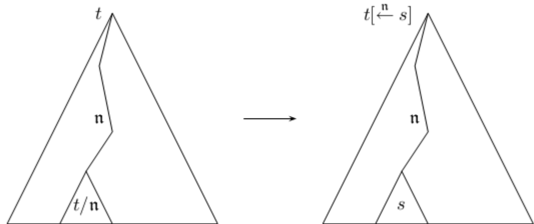


Figure 1.3 – Première forme de substitution.

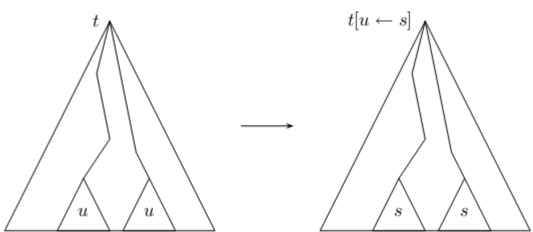


Figure 1.4 – Deuxième forme de substitution.

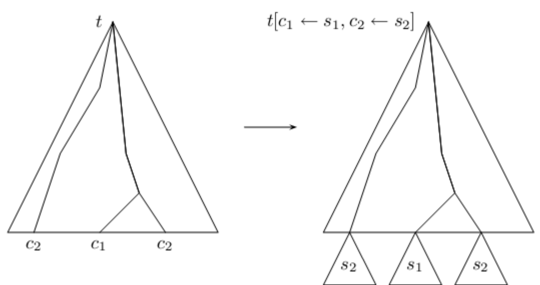


Figure 1.5 – Troisième forme de substitution.

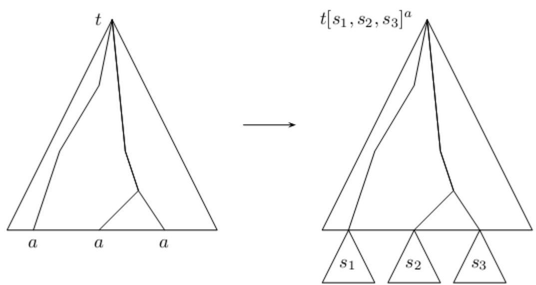


Figure 1.6 – Quatrième forme de substitution.

1.2 Les langages d'arbre

Le concept de langage d'arbre est similaire à celui des mots. Un langage d'arbre est défini comme étant un ensemble d'arbres. Ce concept est important car il permet de décrire les sous-ensembles de tous les arbres possibles.

Ces langages facilitent la description des problèmes tel que : étant donné un arbre t et un langage d'arbre $\mathcal{L}$, déterminer si $t \in \mathcal{L}$

L'ensemble des langages réguliers d'arbre sur un alphabet (Σ, r), noté LRA est défini comme suit [Cle08] :

- Le langage vide $\emptyset$ est un LRA.
- Pour tout feuille $a \in \Sigma_0$, $\{a\} \in LRA$.
- Si $\mathcal{L}_1, ..., \mathcal{L}_n \in LRA$ et $a \in \Sigma \setminus \Sigma_0$, alors $a(\mathcal{L}_1, ..., \mathcal{L}_n) \in LRA$.
- Si $\mathcal{L}_1, \mathcal{L}_2$ sont deux langages réguliers d'arbre, l'union $\mathcal{L}_1 \cup \mathcal{L}_2$ est un langage régulier d'arbre.
- Si $\mathcal{L}_1, \mathcal{L}_2$ sont deux langages réguliers d'arbre et $c \in \Sigma_0$, L'opération consistant à remplacer un arbre appartenant à $\mathcal{L}_1$ dans un arbre appartenant à $\mathcal{L}_2$ à la place de la feuille c, appelée concaténation et notée $\mathcal{L}_{1.c}\mathcal{L}_2$ est un LRA.
- La fermeture ou la closure $\mathcal{L}^{*,a}$ d'un langage régulier d'arbre $\mathcal{L}$ est un LRA, avec $a \in \Sigma_0$. Elle est définie par l'union de tous les ensembles $\mathcal{L}^{n,a}$, tel que :

$$\begin{cases} \mathcal{L}^{0,a} & = a & \text{Si } n = 0 \\ \mathcal{L}^{n+1,a} & = (\mathcal{L}^{n,a}) \cup ((\mathcal{L}) ._a (\mathcal{L}^{n,a})) & \text{Sinon.} \end{cases}$$

1.2.1 Propriété de fermeture des langages réguliers d'arbre.

Une propriété de fermeture d'une classe de langages exprime le fait que cette classe est fermée sous une opération particulière. La classe de langages réguliers d'arbre est fermée par rapport à l'union, l'intersection et la complémentarité. Le livre de référence dans les langages d'arbre [CDG+08] fournit toutes les constructions de ces opérations de fermeture ainsi que leurs preuves.

1.3 Représentation des langages réguliers d'arbre

Les langages d'arbre ne sont pas représentés directement par l'ensemble des arbres, mais définis par une des formes suivantes :

1. un automate d'arbre reconnaissant le langage d'arbre,
2. une expression régulière caractérisant les arbres,
3. une grammaire d'arbre qui peut être dérivée pour obtenir les différentes instances d'arbre.

Nous tenons à noter que les deux représentations qui nous intéressent dans ce mémoire sont les automates d'arbre et les expressions régulières.

1.3.1 Les automates d'arbre finis

Dans la théorie des langages réguliers des mots, les automates finis sont la représentation la plus utilisée. Dans cette théorie, les automates déterministes et non-déterministes sont considérés comme équivalents de point de vue d'acceptation. La direction de traitement des mots est sans importance, et souvent dite de 'gauche à droite' (left-to-right).

Concernant les arbres, plusieurs types d'automates finis d'arbre, qui sont d'ailleurs une généralisation des automates de mots, peuvent être définis. Ici, la direction de traitement de l'arbre est significative. Les automates sont divisés en plusieurs catégories selon deux critères : déterminisme et direction de traitement. Il y a des automates déterministes et non-déterministes, qui peuvent à leurs tours être descendants (root-to-frontier / top-down) ou ascendant (frontier-to-root / bottom-up). Les automates déterministes descendants sont moins puissants que les autres qui sont tous équivalents.

Un automate d'arbre est composé d'un ensemble d'états, un ensemble de transitions, un ou plusieurs états finaux et un alphabet. L'automate a un seul état final s'il est ascendant et plusieurs s'il est descendant.

Contrairement aux automates reconnaissant les mots, où la transition s'exprime par une relation entre deux états, dans les automates d'arbre, une transition est une fonction affectant d'une part un état et d'autre part un ensemble d'états. Ceci s'explique par l'arité des symboles de l'alphabet.

Plusieurs définitions formelles d'automates d'arbre existent dans la littérature. Elles sont similaires et utilisent un alphabet à rang borné.

Définition 6. *Un automate d'arbre $\mathcal{A}$ est un quadruplet (Q, Σ, Δ, Q_T) tel que :*

- *Q est un ensemble fini d'états,*
- *Σ est un alphabet à rang borné,*
- *$\Delta = \{\delta_a, a \in \Sigma\} \cup \{\delta_\varepsilon\}$ est l'ensemble des relations de transition, où $\delta_a \subseteq Q \times Q_n$ pour tout $a \in \Sigma$ et $\delta_\varepsilon \subseteq Q \times Q$ est la relation de transition vide, et*
- *$Q_T \subseteq Q$ est l'ensemble des états finaux. Q_T est aussi notée Q_F.*

Les relations de transition sont de la forme $f(q_1(x_1), q_2(x_2), ..., q_n(x_n)) \longrightarrow q(f(x_1, x_2, ..., x_n))$, où $n \geq 0$, $f \in \Sigma_n$, $q, q_1, ..., q_n \in Q$ et $x_1, x_2, ..., x_n \in \mathcal{X}$.

Si le symbole est une constante, la relation de transition est de la forme $a \longrightarrow q(a)$. Ces règles sont considérées comme des règles initiales (avec lesquelles nous commençons les dérivations).

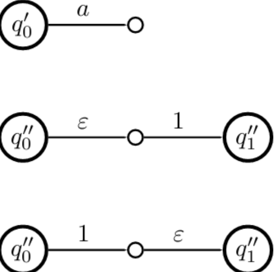


FIGURE 1.7 – Exemples de transitions simples dans un automate d'arbre.

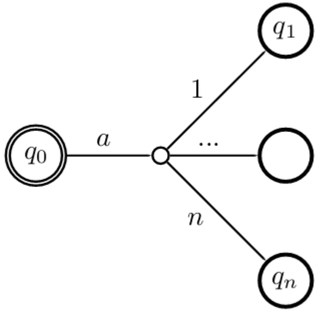


FIGURE 1.8 – Exemple de transition multiple dans un automate d'arbre.

Un arbre (terme) t est reconnu par un automate s'il existe un ensemble de transitions menant à un état final. Cette succession de dérivation est notée $t \overset{*}{\to} q_T(t)$.

Variantes d'automates d'arbre

Lors de son exécution, un automate de mot lit un symbole et se déplace vers un ensemble d'états. Ensuite, il lit un autre symbole et va à un autre ensemble d'états, selon les états précédents. Dans le cas de l'arbre, nous lisons les symboles feuilles et allons à un ensemble de groupes d'états. Dans l'étape suivante de l'automate un autre symbole est lu et en fonction des états des fils cet automate passe à un nouvel ensemble d'états. Les chaînes peuvent être lues en avant ou en arrière. La direction dans laquelle une chaîne est lue n'affecte pas la puissance de l'automate, mais elle peut affecter la taille de ce dernier. Dans le cas de l'arbre cela reste vrai pour le cas non-déterministe, mais pas pour le cas déterministe [ÉF03] [DH03].

Le type d'un automate d'arbre (ascendant ou descendant) est obtenu en imposant une restriction sur l'ensemble des transitions. Ci-dessous, nous donnons une définition formelle de ces types d'automate.

Les automates ascendants (bottom-up ou frontier-to-root)

Définition 7. *Un automate d'arbre $\mathcal{A}$ ascendant est un quadruplet (Q, Σ, Δ, Q_T) tel que :*

- *Q est un ensemble fini d'états,*
- *Σ est un alphabet à rang borné,*
- *Δ est l'ensemble des relations de transition de la forme $f(q_1, q_2, ..., q_n) \to q$, tel que $f \in \Sigma$, $r(f) = n > 0$ et $q_1, q_2, ..., q_n, q \in Q$. Si $r(f) = 0$, nous avons une règle de la forme $f \to q$.*
- *$Q_T \subseteq Q$ est l'ensemble des états finaux.*

Notons qu'il n'y a pas d'états initiaux.

Exemple

Soit $\Sigma = \{a, b, c, d\}$ où $r(a) = 2$, $r(b) = 1$ et $r(c) = r(d) = 0$. Soit un automate $\mathcal{A} = (Q, \Sigma, \Delta, Q_T)$ où : $Q = \{q_0, q_1, q_2, q_3, q_4, q_5, q_6, q_s\}$, $Q_T = \{q_s\}$ et $\Delta = \{d \to q_2, c \to q_6, b(q_6) \to q_5, b(q_1) \to q_0, a(q_5, q_1) \to q_4, a(q_1, q_2) \to q_3\, \varepsilon(q_6) \to q_s, \varepsilon(q_4) \to q_s, \varepsilon(q_s) \to q_1, \varepsilon(q_3) \to q_s, \varepsilon(q_2) \to q_1, \varepsilon(q_0) \to q_1\}$.

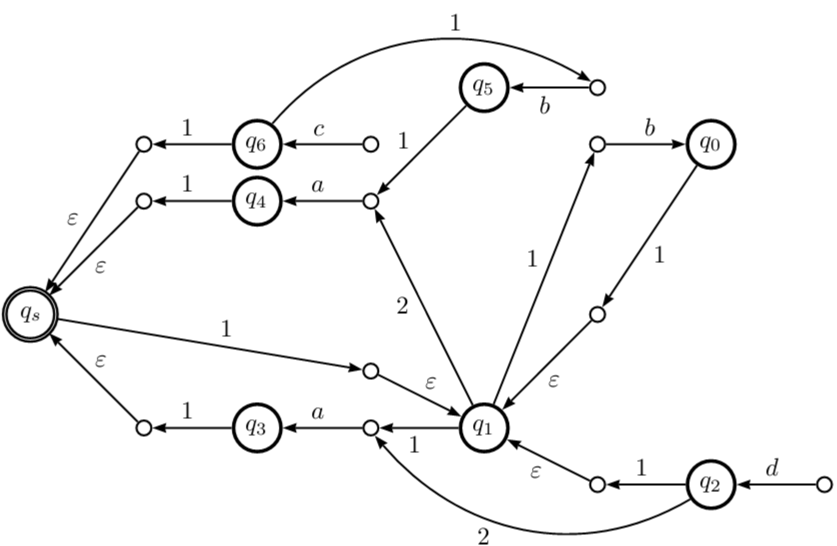


FIGURE 1.9 – Exemple d'un automate ascendant.

Les automates descendants (top-down ou root-to-frontier)

Définition 8. *Un automate d'arbre $\mathcal{A}$ descendant est un quintuplet $(Q, Q_I, \Sigma, \Delta, Q_T)$ tel que :*

- *Q est un ensemble fini d'états,*
- *$Q_I \subseteq Q$ est l'ensemble des états initiaux,*
- *Σ est un alphabet à rang borné,*
- *Δ est l'ensemble des relations de transition de la forme $q(f) \to (q_1, q_2, ..., q_n)$, tel que $f \in \Sigma$, $r(f) = n > 0$ et $q_1, q_2, ..., q_n, q \in Q$. Si $r(f) = 0$, nous n'avons pas de règles dans Δ.*

– $Q_T \subseteq Q$ *est l'ensemble des états finaux.*

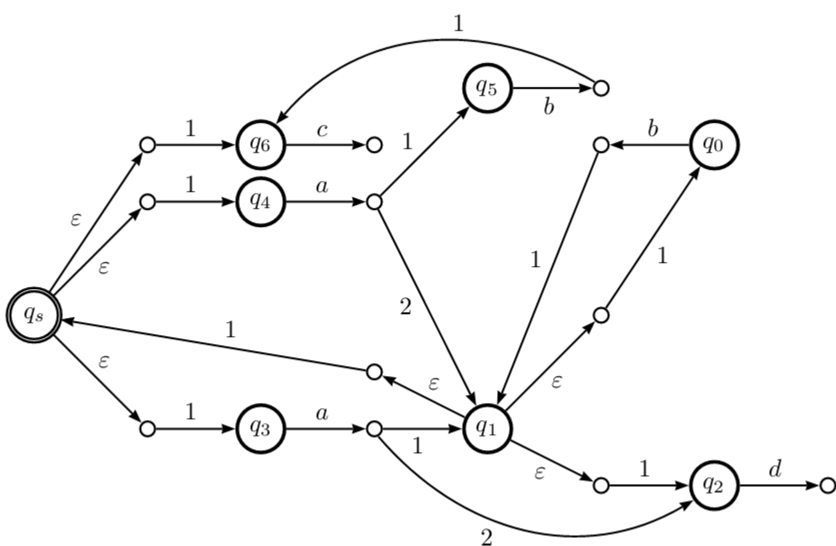


FIGURE 1.10 – Exemple d'un automate descendant.

Bien que la manipulation des automates d'arbres descendants semble plus naturelle, les automates d'arbre ascendants sont plus faciles à implémenter et manipuler [CDG+08].

Automates d'arbre et déterminisme

Le non-déterminisme dans les automates d'arbre peut être défini de la même manière que pour les automates de mots. Un automate d'arbre est non-déterministe si pour un état q et un symbole de l'alphabet a, il existe plus d'une transition ou éventuellement une transition étiquetée par ε.

Autrement dit, pour chaque état dans un automate d'arbre déterministe, il existe au plus une transition pour chaque symbole différent de ε. Un automate d'arbre est dit complet si et seulement s'il existe au moins une règle de transition pour chaque symbole $x \in \Sigma$.

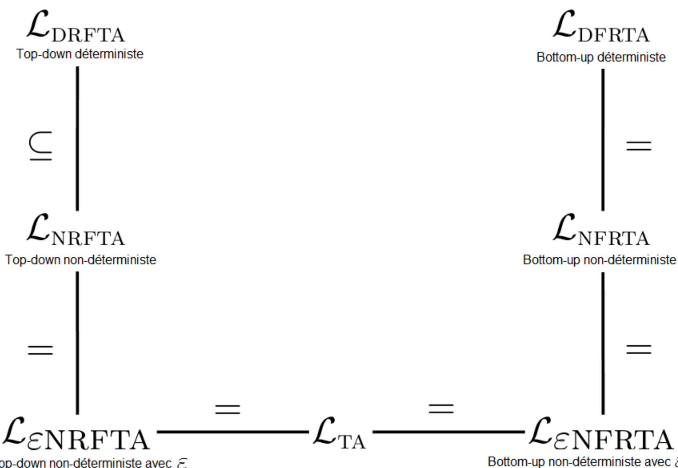


FIGURE 1.11 – Relations entre les différents types d'automates d'arbre.

La figure 1.11 schématise les relations entre ces différents types d'automates. Tous ces automates d'arbre ont la même force d'expression (ils acceptent les mêmes langages). Néanmoins, il a été démontré que les automates descendants déterministes sont moins puissants que les autres [Cle08].

Pour chaque langage régulier, il existe un automate d'arbre fini le reconnaissant. En plus, cet automate peut être déterministe et minimal. Les auteurs du livre TATA [CDG+08] décrivent une panoplie d'algorithmes permettant de rendre les automates d'arbre déterministes et minimales ainsi que les preuves d'existence de tels automates.

Minimisation des automates d'arbre

Le théorème Myhill-Nerode est un résultat classique de la théorie des automates finis [Bor03]. Ce théorème caractérise les ensembles reconnaissables et il a de nombreuses applications. Une conséquence de ce théorème, entre autres, c'est qu'il y a un automate minimum unique pour chaque langage reconnaissable sur un alphabet fini. Le théorème Myhill-Nerode est généralisé de manière directe aux automates d'arbre finis [CDF07]. Les preuves d'existence de cet automate ainsi que l'algorithme de minimisation d'un automate d'arbre sont donnés dans [CDG+08].

1.3.2 Les expressions régulières d'arbre

Les expressions régulières d'arbre sont aussi des représentations permettant de décrire les langages réguliers d'arbre définis sur un même alphabet Σ. Cependant, les arbres peuvent être concaténés en plusieurs endroits, contrairement aux mots qui ne peuvent être connectés qu'à la fin.

Construction d'expressions régulières d'arbre

Nous définissons l'ensemble $RTE(\Sigma)$ des expressions régulières sur l'alphabet Σ.

1. L'ensemble vide $\emptyset$ est une expression régulière,
2. Si $a \in \Sigma_0$, alors $a \in RTE(\Sigma)$,
3. Si $f \in \Sigma_n$ et $E_1, E_2, ..., E_n \in RTE(\Sigma)$, alors $f(E_1, E_2, ..., E_n) \in RTE(\Sigma)$,
4. Si $E_1, E_2 \in RTE(\Sigma)$ alors $(E_1 + E_2) \in RTE(\Sigma)$,
5. Si $E_1, E_2 \in RTE(\Sigma)$ et $c \in \Sigma_0$, alors $(E_1 ._c E_2) \in RTE(\Sigma \cup \{\varepsilon\})$,
6. Si $E \in RTE(\Sigma \cup \{\varepsilon\})$ et $c \in \Sigma_0$, alors $E^{*c} \in RTE(\Sigma \cup \{\varepsilon\})$.

1.3.3 Les grammaires régulières d'arbre

Les grammaires régulières d'arbre jouent un rôle important dans les problèmes algorithmiques tel que la recherche de motifs ou l'acceptance.

Une grammaire régulière d'arbre (regular tree grammar : RTG) est une grammaire utilisée afin de générer des arbres réguliers. Ces arbres sont générés de la même manière que les mots : l'application des règles de productions sur les symboles non-terminaux en commençant par le symbole de départ.

Définition 9. *Une grammaire régulière d'arbre est définie par un quintuplet :* $(N, \Sigma, r, S, Prods)$ *où :*
- N *et* Σ *représentent les ensembles des terminaux et non-terminaux,*
- r *représente la fonction d'arité,*
- S *est le symbole de départ,*
- *et* $Prods$ *est l'ensemble des règles de production de la grammaire.*

Chaque règle de production est de la forme $A \rightarrow \alpha$, où A est un non-terminal et α est un arbre. Cet arbre peut contenir des terminaux et non-terminaux, avec la restriction que les non-terminaux sont présents uniquement comme des feuilles, ils ont toujours l'arité 0.

Exemple Soit la grammaire $G = (N, \Sigma, r, S, Prods)$ tel que :
$N = \{S, B\}$
$\Sigma = \{a, b, c, d\}$
$r = \{(S, 0), (B, 0), (a, 2), (b, 1), (c, 0), (d, 0)\}$
$Prods = \{S \longrightarrow a(B, d), S \longrightarrow a(b(c), B), S \longrightarrow c, B \longrightarrow b(B), B \longrightarrow S, B \longrightarrow d\}$

La construction d'un arbre à partir de cette grammaire se fait par l'application de la règle de production dont le coté gauche commence par le non-terminal initial S en remplaçant ce symbole par le coté droit. Ce processus est répété jusqu'à ce qu'il ne reste plus de symboles non-terminaux [Cle08].

Le langage régulier généré par G est l'ensemble de tous les arbres qui peuvent être produits par une dérivation à partir de l'axiome S. Deux grammaires sont dites équivalentes si elles génèrent le même langage régulier [Cle08].

Définition 10. ***La dérivation***

Pour une grammaire G et pour tout $\beta_1, \beta_2 \in T(N \cup \Sigma, r)$, la dérivation $\beta_1 \overset{A \to \alpha}{\Rightarrow} \beta_2$ veut dire qu'il existe une règle de production $A \to \alpha \in Prods$ et $\beta_2 = \beta_1[A \to \alpha]$. Elle peut être notée simplement $\beta_1 \Rightarrow \beta_2$.

$\Rightarrow^+$ et $(\Rightarrow^*)$ sont utilisées pour indiquer une suite de dérivation. $\Rightarrow^+$ pour une dérivation ou plus et $(\Rightarrow^*)$ pour zéro ou plusieurs dérivations.

Pour une grammaire G et $\alpha \in T(N \cup \Sigma, r)$, $\mathcal{L}(G, \alpha) = \{t, t \in T(\Sigma, r) \wedge \alpha \Rightarrow^* t\}$.

Le langage d'une grammaire G est défini par $\mathcal{L}(G) = \mathcal{L}(G, S)$, où S est le symbole de départ de G.

Caractéristiques des grammaires régulières

Il existe deux caractéristiques importantes pour les grammaires régulières d'arbre. Ces caractéristiques décrivent la présence de règles de production particulières [Cle08].

La première caractéristique vérifie si une grammaire peut contenir $(U+)$ ou non $(U-)$ des règles de production dont la partie droite (right hand side) est un non-terminal. Ces règles sont appelées 'règles unitaires'. La suppression de ces règles produira une autre grammaire $(U-)$, et par conséquent change le langage généré par la grammaire initiale. La forme générale des règles de production d'une grammaire $(U-)$ est $A \longrightarrow a(t_1, ..., t_n)$ (pour $n = 0$, $A \longrightarrow a$).

La deuxième caractéristique se concentre sur le cas contraire, où la partie droite de la grammaire contient des arbres contenant ce qu'on appelle des nœuds Z (Z-nodes). Ces nœuds Z sont des nœuds qui ne sont pas des nœuds racines et qui contiennent des terminaux. Une grammaire qui contient des nœuds Z est appelée une grammaire $Z+$ et $Z-$ pour le cas contraire. Les règles de production d'une grammaire $Z-$ sont de la forme : $A \longrightarrow a(A_1, ..., A_n)$ (pour $n = 0$, $A \longrightarrow a$) ou $A \longrightarrow B$.

A partir de ces caractéristiques, plusieurs types de grammaires régulières peuvent être définis :

- Dans les grammaires Z_+U_+ : les règles de production sont de la forme $A \longrightarrow \alpha$, $A \longrightarrow a(t_1, ..., t_n)$ ou $A \longrightarrow B$. Ce genre de grammaire est appelé 'grammaire régulière d'arbre' ou 'grammaire d'arbre à forme normale' (TNF grammars, tree normal form grammars).

- Les grammaire Z_+U_- sont des grammaires TNF sans règles de production unitaires. Les règles de production prennent la forme $A \longrightarrow a(t_1, ..., t_n)$ (pour $n = 0$, $A \longrightarrow a$).
- Les grammaires Z_-U_- sont appelées 'grammaires d'arbre expansives' (ETG, expansive tree grammars) ou grammaires d'arbre en forme normale ou encore grammaires d'arbre régulières normalisées. Les règles de production sont de la forme $A \longrightarrow a(A_1, ..., A_n)$ (pour $n = 0$, $A \longrightarrow a$).
- Les grammaire Z_-U_+ sont des grammaires ETG avec des règles de production unitaires. Donc, les règles de production prennent la forme $A \longrightarrow a(A_1, ..., A_n)$ (pour $n = 0$, $A \longrightarrow a$) ou $A \longrightarrow B$.

Exemples Considérons la grammaire $G = \{N, \Sigma, r, Prods, S\}$, $N = N_0 = \{S, A, B, C\}$.

- $Prods = \{S \longrightarrow b(a(c, c)), S \longrightarrow A, A \longrightarrow b(A), A \longrightarrow b(a(c, c))\}$. La grammaire est de type Z_+U_+.
- $Prods = S \longrightarrow b(a(c, c)), S \longrightarrow b(S)$. Cette grammaire est de type Z_+U_-.
- $Prods = S \longrightarrow b(A), A \longrightarrow b(A), A \longrightarrow a(B, B), B \longrightarrow c$. La grammaire est de type Z_-U_-.
- $Prods = S \longrightarrow b(A), A \longrightarrow b(A), B \longrightarrow a(c, c) \longrightarrow c$. Cette grammaire est de type Z_-U_+.

Normalisation des grammaires régulières

Après la définition des différents types de grammaires régulières nous allons discuter trois transformations pouvant être effectuées sur les grammaires régulières d'arbre.

La première transformation consiste à supprimer les symboles et les règles de production inutiles. Les autres transformations permettent l'élimination des règles de production qui empêchent la grammaire d'être Z_- ou U_-. Autrement dit, les transformations effectuées concernent les règles de production unitaires ou ayant un nœud Z dans leur partie droite.

Symboles et règles de production inutiles

Une grammaire régulière d'arbre peut avoir des symboles ou des règles de production inutiles exactement comme pourrait l'être une grammaire de mots. La présence de ces symboles et règles de production peut rendre les preuves compliquées ou influencer sur l'efficacité de certains algorithmes opérants sur les grammaires [Str07].

Symboles atteignables et symboles productifs

Dans cette section nous discutons l'utilité des symboles et règles de production d'une grammaire régulière d'arbre. Certaines règles de productions peuvent ne pas être utiles pour la construction d'arbres. Ces items inutiles

sont de deux types : non-atteignables depuis le symbole de départ S ou improductifs.

Un symbole est dit atteignable depuis S s'il peut être atteint depuis le symbole de départ S et une règle de production est dite atteignable depuis S si le symbole de sa partie gauche est atteignable. Un symbole B est dit atteignable par un autre symbole A, si B peut être atteint en appliquant n'importe quelle séquence de règles de production à partir de A [Str07].

Les symboles et les règles de production qui ne sont pas atteignables depuis S sont inutiles.

Les règles et les symboles improductifs sont des problèmes de la même nature. Un non-terminal est appelé productif s'il existe une règle de production de la grammaire productive dont sa partie gauche est ce symbole. Un symbole terminal est toujours considéré comme productif.

Une règle de production est considérée comme productive si elle ne contient pas de symboles non-productifs dans sa partie droite.

L'élimination de ces symboles et ces règles de production ainsi que des preuves et des exemples sont donnés en détail dans la thèse de PhD de L. Cleophas [Cle08].

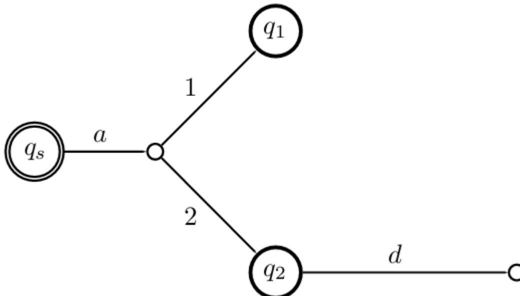


FIGURE 1.12 – Un automate d'arbre avec un état atteignable mais inutile (q_2).

1.4 Problèmes des langages réguliers d'arbre résolus par les automates

Le problème le plus simple et le plus important est celui de l'acceptation. Il consiste à déterminer si un arbre t peut être généré par une grammaire régulière d'arbre G. Ceci se fait souvent à travers la construction de l'automate d'arbre de la grammaire G et ensuite le parcours de cet automate sur t [Str07].

Le deuxième problème est celui de la RMA. Il s'agit de retrouver pour un ensemble fini, non vide de motifs (patterns) et un arbre, toutes les occurrences de ces motifs dans l'arbre. Ce problème est aussi résolu en utilisant les automates d'arbre. Nous construisons les automates d'arbre correspondants aux motifs puis nous déroulons ces derniers sur l'arbre objet.

Le dernier problème est celui de l'analyse grammaticale (tree parsing). C'est une extension du problème d'acceptation, sauf qu'ici nous cherchons seulement à déterminer si un arbre est généré par une grammaire et comment. Autrement dit, quelles règles de production appliquer à chaque nœud de l'arbre pour avoir l'arbre entier. La construction de l'automate d'arbre de la grammaire permet de résoudre ce problème.

1.5 Applications des langages réguliers d'arbre

Le domaine d'application des langages réguliers d'arbre est très large et varié. Nous pouvons citer [Cle08] :

- Génération de code et optimisation des compilateurs [AG85, HC86, AGT89, HK86] ;
- Réécriture de termes et unification [HO82a, HO82b, O'D85] ;
- Génétique, particulièrement la recherche de motifs ADN/ARN [SZ90] ;
- Traitement de documents XML [Mur99, BKMW01, Nev02a, Nev02b, MLMK05, Sch07] ;
- Vérification de protocoles, particulièrement ceux des réseaux et cryptographie [GK00, Gou00, Mon99].

Les concepts de base de ces domaines sont convertis en arbres. Par exemple, en compilation, le code source d'un programme est transformé en un arbre syntaxique en utilisant l'analyse lexicale et grammaticale. L'arbre résultant est traduit en une séquence d'instructions machine. Cette traduction est appelée 'génération/sélection de code' et est considérée comme un problème de tree parsing [Str07].

De la même façon, un fichier XML est représenté par un arbre où les nœuds sont des tags. Extraire n'importe quelle information revient à faire une exploration de l'arbre.

1.6 Conclusion

Ce chapitre est une introduction aux langages réguliers d'arbre. Nous y avons présentés les notions de base de ce type de langages ainsi que ses différentes représentations : les automates, les expressions régulières et les grammaires. Ces notions seront utilisées dans les chapitres suivants pour l'étude des algorithmes de recherche de motifs qui existent dans la littérature ainsi que pour la proposition d'un nouvel algorithme basé sur les automates d'arbre.

Chapitre 2

Recherche de motifs d'arbre

Plusieurs algorithmes de RMA sont apparus dans la littérature. L'étude de ce problème a été introduite par Hoffmann et O'Donnell en 1982 dans leur article [HO82a]. La recherche de motif dans les arbres peut être considérée comme une extension de celle des mots. L'algorithme le plus connu de recherche de motif à partir d'une expression régulière dans les mots est celui de Thompson [Tho68]. Il consiste à construire, par induction, un automate à partir de l'expression régulière du motif puis dans une seconde étape de parcourir le texte objet et de le dérouler sur l'automate pour identifier les différents motifs. Depuis l'algorithme de Hoffmann et O'Donnell, de nombreux algorithmes améliorant ce dernier sont apparus ainsi que d'autres techniques de RMA. Dans ce chapitre nous allons présenter quelques algorithmes parmi les plus intéressants, les techniques de base utilisées et leurs complexités engendrées. Nous commençons par définir le problème de recherche de motifs dans les arbres.

2.1 Définition du problème

Un motif d'arbre (pattern) est un arbre dans lequel des nœuds variables peuvent figurer. Ces nœuds doivent être obligatoirement au niveau des feuilles. Nous appelons ce genre d'arbre motif à cause de ces nœuds particuliers qui représentent tous les sous-arbres possibles. C'est à dire qu'ils peuvent être substitués par n'importe quel arbre du langage.

Pour définir les motifs nous avons besoin d'étendre l'alphabet du langage en ajoutant un nouveau symbole v pour désigner une variable. Pour l'alphabet (Σ, r), nous aurons l'alphabet (Σ', r'), où :

- $\Sigma' = \Sigma \cup \{v\}$,
- $r'(v) = 0$,
- et $r'(a) = r(a)$ pour tout $a \in \Sigma$.

Les arbres de $T' = (\sigma', r')$ sont appelés motifs, motifs d'arbre (tree patterns, pattern trees, patterns). Notons que v peut être uniquement un terme dans Σ_0 [Cle08].

Définition 11. ***La recherche de motifs***

La fonction match $\in T'(\Sigma', r') \times T(\Sigma, r) \times \mathbb{N}_+ \longrightarrow \mathbb{B}$ *est définie pour un motif* $p \in T'(\Sigma', r')$*, un arbre objet* $t \in T(\Sigma, r)$ *et* $n \in \mathbb{N}_+$ *par :*

$$match(p, t, n) = (n \in \mathcal{D}_t) \wedge (\exists s_1, ..., s_k \in T(\Sigma, r) / p[s_1, ..., s_k]^v = t/n)$$

où $p[s_1, ..., s_k]^v$ *est la substitution de* v *par* $s_1, ..., s_k$*.*

Dans l'exemple de la figure 2.1, pour $t = a(b(c), a(b(c), a(c, c)))$ et $p = a(b(c), v)$, la fonction $match(p, t, n)$ est vérifiée uniquement pour $n = 0$ et $n = 2$. $match(p, t, 0)$ est vérifiée car $t/0 = p[a(b(c), a(c, c))]^v$. La même chose pour $match(p, t, 2)$, $t/2 = p[a(c, c)]^v$.

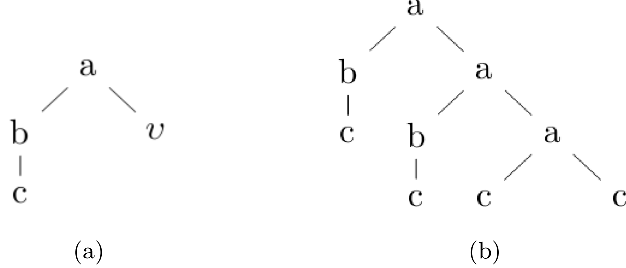


Figure 2.1 – Exemple d'un motif et d'un arbre correspondant à ce motif.

Nous appelons un arbre correspondant à un motif d'arbre une instance de ce motif.

2.2 Algorithmes de recherche de motifs

Les algorithmes de recherche de motifs dans la littérature sont nombreux. Ils sont basés sur des mécanismes différents. Ces différences sont dues à la représentation du motif ainsi que l'arbre objet. En effet, nous retrouvons les algorithmes utilisant les automates d'arbre, ceux qui transforment l'arbre vers des chaînes de caractères et ensuite utilisent les algorithmes de recherche de motifs pour les strings...

Cleophas, dans sa thèse PhD [Cle08] a effectué une étude sur les algorithmes de reconnaissance des arbres ainsi que la recherche des motifs d'arbre. Il a construit une taxonomie des différents algorithmes de la littérature. Ces algorithmes sont de trois catégories :

1. algorithmes utilisant les automates d'arbre,
2. algorithmes basés sur la notion de 'match set',
3. algorithmes basés sur le 'stringpath matching'.

Les concepts de 'match–set' et 'stringpath matching' seront abordés plus loin dans ce chapitre.

Pour l'élaboration de cette taxonomie, L. Cleophas a utilisé TABASCO (TAxonomy-BAsed Software COnstruction) ; une méthode de modélisation des domaines algorithmiques. Cette méthode implique un certain nombre d'étapes :

1. Sélection d'un domaine algorithmique spécifique. Un domaine est choisi en fonction de sa richesse (l'existence de plusieurs algorithmes et structures de données) ; sa maturité (disponibilité d'une théorie riche pouvant être utilisée pour résoudre) et son applicabilité (les algorithmes doivent avoir un large champ d'application dans les systèmes réels).

2. Étude de la littérature du domaine. Après le choix du domaine d'étude, une recherche scientifique et un survey sont effectués, pour rassembler le plus grand nombre possible d'algorithmes et de structures de données.
3. La construction de la taxonomie. C'est une classification des problèmes et leurs solutions selon les détails essentiels des algorithmes classifiés, ceci permet de rendre ce domaine plus accessible et peut conduire à la déduction et la découverte de nouveaux algorithmes.
4. Conception et mise en œuvre d'une boite à outils (toolkit). La disponibilité d'une taxonomie simplifie le processus de conception d'une boîte à outils.

Cette taxonomie est élaborée à partir de certains algorithmes de base. Nous allons étudier de chacune des catégories citées ci-dessus quelques algorithmes en détails en donnant leurs principes de fonctionnement ainsi que leurs complexités.

1. Nous allons commencer par donner une brève description de l'algorithme naïf, ensuite dans la catégorie des algorithmes utilisant les automates d'arbre nous présentons les algorithmes de Hoffmann et O'Donnell (descendant-1982), de Cleophas (2005) et al. et de Chase (1986).
2. Dans les algorithmes se basant sur la notion de 'match set', nous donnons l'algorithme de Hoffmann et O'Donnell (ascendant).
3. Après, nous présentons les algorithmes de Ramesh et Ramakrishnan (1992) et Cole et Ramesh (1997, 1998, 2003) qui rentrent dans la catégorie des algorithmes utilisant la notion de 'stringpath'.
4. Nous présentons aussi deux autres algorithmes utilisant des techniques qui n'ont pas été classifiées dans la taxonomie de Cleophas : L'algorithme de Kosaraju (1989) et son amélioration par Dubiner et al. (1994).
5. Enfin, nous exposons un algorithme lié à notre approche, utilisant un autre type d'automate d'arbre : les automates à pile (push-down automata). Cet algorithme a été proposé par Polách dans son mémoire de Master [Pol11], il consiste à construire un automate d'arbre à pile à partir d'une expression régulière d'arbre par analogie à la construction de Thompson pour les mots [Tho68].

Nous avons rajouté à la taxonomie des algorithmes de RMA la notion de la recherche à partir d'une expression régulière d'arbre. Les figures 2.2 et 2.3 résument les différents algorithmes étudié selon la technique utilisée pour la recherche de motifs et selon l'entrée de l'algorithme de RMA.

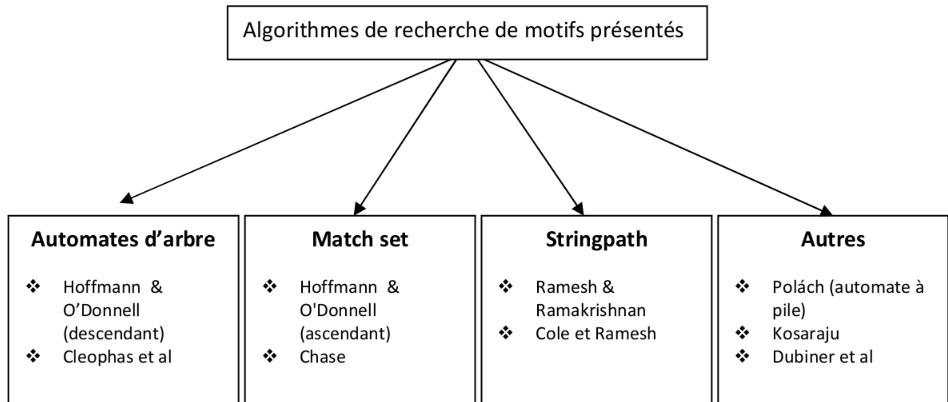


FIGURE 2.2 – Classification des algorithmes de RMA présentés selon la méthode utilisée.

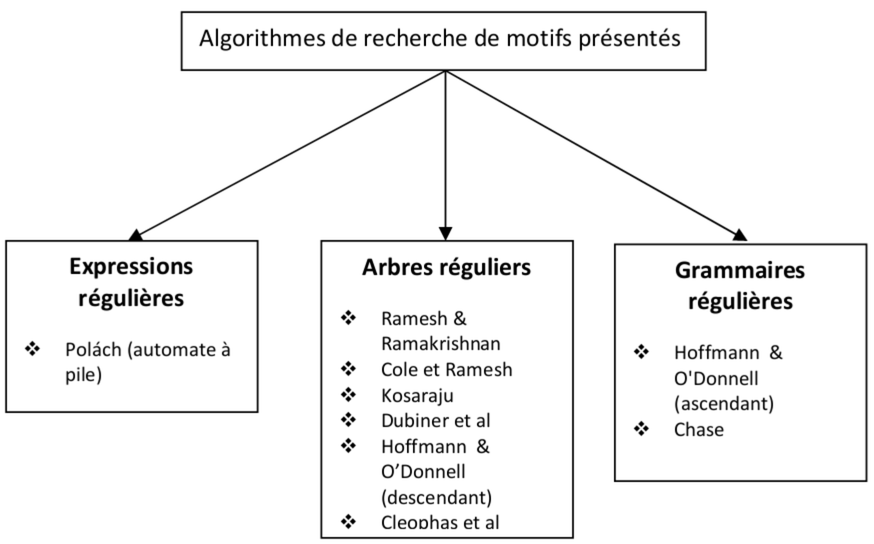


FIGURE 2.3 – Classification des algorithmes de RMA présentés selon l'entrée de l'algorithme.

2.2.1 L'algorithme naïf de recherche de motifs

Par analogie aux mots, l'algorithme de RMA naïf est le plus simple. Il procède de la manière suivante. Pour chaque nœud possible du motif dans l'arbre objet, nous testons si ce nœud est une occurrence du motif ou non. Ce test est effectué en comparant les nœuds du motif avec les nœuds de l'arbre objet de gauche à droite. Si tous les nœuds du motif sont égaux aux nœuds du texte aux positions correspondantes, une occurrence a été trouvée et la position de cette occurrence est retournée. Sinon, la recherche se poursuit en

passant au nœud suivant. La complexité de l'algorithme naïf est de l'ordre de $O(n.m)$ où n est la taille du motif et m est la taille de l'arbre objet.

2.2.2 Algorithme de Hoffmann et O'Donnell (1982)et ses variantes

Hoffmann et O'Donnell proposent [HO82a] deux approches de RMA : la première approche qui est une généralisation de l'algorithme de Knuth-Morris-Pratt des mots [KMP77], est ascendante. La seconde approche est descendante.

Alors que la méthode ascendante généralise le cas des mots, celle descendante réduit le problème de recherche de motifs dans les arbres à un problème équivalent dans les mots.

La méthode ascendante est caractérisée par un nombre important de traitements mais aussi par une rapidité dans la recherche de motifs. Elle est développée à partir de la notion de 'match set'.

Les algorithmes descendants ont un temps de traitement meilleur que celui des algorithmes ascendants mais ils ont une vitesse de recherche de motif inférieure.

Les algorithmes présentés dans cet article ont une complexité de l'ordre de $O(n^2 + n^{r(a)}.ht)$ où : n est la somme des tailles de motifs, ht est la profondeur d'un arbre construit au cours du traitement et $r(a)$ est la plus grande arité de l'alphabet.

L'algorithme ascendant

L'idée de base de cet algorithme est de retrouver, dans chaque nœud dans l'arbre objet, tous les sous-arbres qui correspondent au motif à partir de ce nœud (match). Pour un nœud n dans l'arbre objet étiqueté par un symbole b d'arité q, supposons que nous voulons calculer l'ensemble M de tous les motifs différents de la variable v qui correspondent au motif dans le nœud n, car v correspond toujours à n'importe quel nœud.

Supposons que nous avons déjà calculé ces ensembles pour tous les fils de n, nous appelons ces ensembles de droite à gauche : $M_1, ..., M_q$. Alors M contient v plus exactement ces sous-arbres de motifs $b(t_1, ..., t_q)$ tel que $t_i \in M_i$ pour $i = 1..q$. Donc, nous pouvons calculer M en formant les arbres $b(t_1, ..., t_q)$ pour toutes les combinaisons $(t_1, ..., t_q)$ et ensuite vérifier si chaque membre de M est un sous-motif. Une fois ces ensembles associés à chaque nœud de l'arbre objet, le problème de recherche de motif est résolu car n'importe quel motif est signalé par la présence d'un motif complet dans un ensemble quelconque.

Il faut noter que ces ensembles M sont finis parce que Σ et l'ensemble des motifs P sont finis. Par conséquent nous pouvons calculer ces ensembles

à l'avance, les coder par des énumérations et après construire des tables. Pour un symbole donné b et les différents codes de M_i, ces tables donnent le code pour M. Pour un symbole avec une arité q, nous aurons une matrice de dimension q pour ce symbole.

Avec ces tables le problème de recherche de motifs devient trivial : traverser l'arbre objet en post-fixé et attribuer à chaque nœud n le code c représentant l'ensemble des motifs partiels dans le nœud n.

Les tables sont composées de tableaux, un tableau pour chaque symbole de l'alphabet. Si un nœud n est étiqueté par le symbole b d'arité q, alors un tableau de dimension q est utilisé pour b. Le code c à un nœud n est la valeur indexée par le tuple $(c_1, ..., c_q)$ où c_i est le code associé au $i^{\text{ième}}$ fils de n (à partir de la gauche). Si l'ensemble représenté par c contient le motif p_i, alors le couplet (n, i) est ajouté à la solution.

Le temps de recherche de motif de cet algorithme est clairement de l'ordre de $O(m)$ pour calculer tous les codes plus $O(match)$ pour créer la liste des solutions.

La linéarité constante de la complexité est due à la construction d'un tableau de référence pour calculer les codes, un seul test pour vérifier la présence du motif, plus le overhead pour la traversée post-fixée. L'espace requis pour cet algorithme dépend de la taille des tables.

Exemple

Considérons le problème de recherche de motifs avec les motifs :

$P_1 = a(a(v, v), b)$ et $P_2 = a(b, v)$. L'alphabet $\Sigma = \{a, b, c\}$, a est d'arité binaire et b et c sont des feuilles.

Parmi les trente deux ensembles construits : $set1 = \{v\}$, $set2 = \{b, v\}$, $set3 = \{a(v, v), v\}$, $set4 = \{a(b, v), a(v, v), v\}$, $set5 = \{a(a(v, v), b), a(v, v), v\}$.

Table pour le nœud a

Fils gauche \ Fils droit	1	2	3	4	5
1	3	3	3	3	3
2	4	4	4	4	4
3	3	5	3	3	3
4	3	5	3	3	3
5	3	5	3	3	3

Table pour le nœud b : 2
Table pour le nœud c : 1

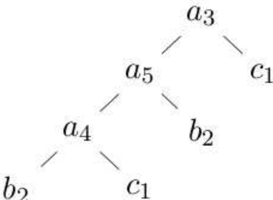


FIGURE 2.4 – Attribution des codes pour l'algorithme descendant.

Par conséquent, l'attribution de 4 à un nœud n indiquera que tous les membres de l'ensemble 4 correspondent au motif dans le nœud n. C'est à dire le motif P_2 est retrouvé. Pour une valeur de 5 le motif P_1 est retrouvé.

L'entrée $(3, 2)$ de la table de a vaut 5 parce que pour le fils gauche, nous avons une correspondance avec les motifs $a(v, v)$ et v, et pour le fils droit nous avons une correspondance avec les motifs b et v.

La figure 2.4 montre l'attribution de tous les codes pour l'algorithme ascendant avec les tables créées précédemment. Notons que le motif P_1 est vérifié pour le nœud avec le code 5 et P_2 pour le nœud avec le code 4.

L'algorithme descendant

Comme l'algorithme ascendant, cet algorithme est basé sur celui de Knuth-Morris-Pratt pour les mots. Cette fois ci au lieu de généraliser la recherche de motifs de mots, l'algorithme réduit le problème de recherche de motifs à un problème équivalent pour les mots.

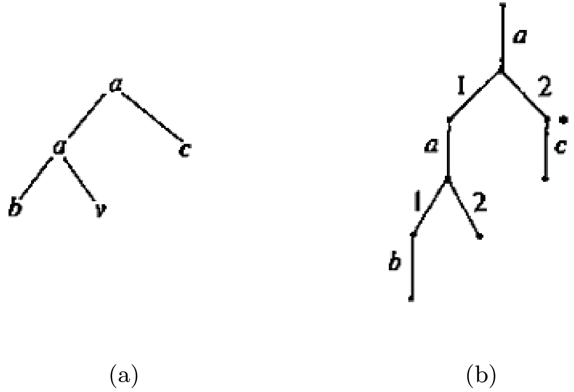
L'idée de base de la réduction du problème est de considérer chaque chemin de la racine vers la feuille comme une chaîne de caractères dans laquelle les symboles de l'alphabet sont intercalés avec des nombres indiquant quelle branche du père a été suivie. Comme les variables v correspondent toujours aux motifs, elles ne vont pas être incluses dans cette chaîne de caractères.

Par exemple, le motif $a(a(b, v), c)$ est associé à l'ensemble de chaînes de caractères $\{a1a1b, a1a2, a2c\}$. Notons que nous avons omis le symbole v de la fin de la deuxième chaîne de caractères.

Pour simplifier la présentation de l'algorithme, les auteurs développent en premier lieu l'algorithme pour le cas d'un ensemble de motif composé d'un seul arbre. Étant donné un motif P, il est facile de générer tous les chemins de chaînes de caractères depuis la racine vers les feuilles. Ensuite, ils utilisent l'algorithme d'Aho et Corasick [AC75] pour produire un automate d'arbre reconnaissant toutes les instances des chemins dans un arbre objet. Comme la combinaison des longueurs de tous les chemins est de l'ordre de $O(n)^2$, ils modifient cette construction pour éviter de générer les chaînes de caractères explicitement. Ceci diminue la complexité de traitement à $O(n)$.

La première étape dans l'algorithme d'Aho et Corasick est de construire une structure arborescente (a trie en anglais) pour les chemins du motif d'arbre P. Cette arborescence est appelée la fonction 'goto'. Cette structure est un arbre dont les nœuds représentent les préfixes distincts des chemins. Si un nœud n représente x et n' représente xa, $a \in \Sigma \cup N$, alors n est le père de n', et l'arête de n à n' est étiquetée par a.

Exemple
L'arborescence associée a l'arbre de motif $a(a(b, v), c)$ est donnée dans la figure 2.5.



(a) (b)

FIGURE 2.5 – Le motif P (a) et son arborescence associée (b).

L'arborescence est construite en partageant chaque nœud du motif différent de v en deux nœuds connectés par une arête étiquetée par le symbole du nœud original. La figure 2.6 montre l'automate associé au motif de l'exemple précédent.

Les étapes de construction de l'automate pour la recherche de motifs est exactement la même que celle d'Aho et Corasick [AC75] car nous traitons ici un problème de chaînes de caractères. Donc, la construction totale requiert $O(n)$ étapes.

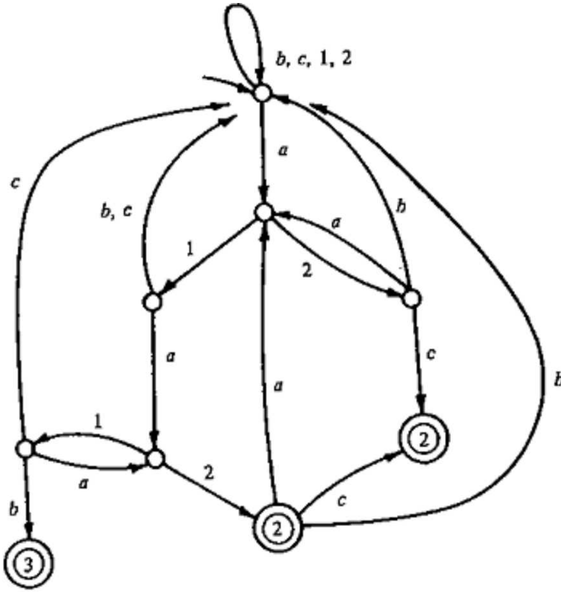


FIGURE 2.6 – L'automate de recherche de motifs construit pour le motif P.

Les auteurs de ce travail ont présenté pour conclure un tableau résumant leurs différents algorithmes de recherche de motifs ainsi que les différentes constructions et les restrictions sur les motifs et l'arbre objet ainsi que leurs complexités spatiales et temporelles correspondantes. Ils estiment que chacun des algorithmes a ses avantages et ses inconvénients. Néanmoins, pour choisir une technique de recherche de motifs, certains détails doivent être pris en considération. Par exemple, pour un temps de recherche de motifs plus rapide, l'utilisation de l'algorithme ascendant avec une table est privilégiée. Ceci est au détriment de l'espace occupé par les tables.

Dans leur article, les auteurs exposent aussi des algorithmes ascendants utilisant le hashage, que nous n'avons pas mentionné dans notre étude. Leur complexité respective est de l'ordre de $O(set.(rank.n + nbr_{pat})) + O(m + match)$ où set est le nombre d'ensembles distincts de nœuds qui correspondent aux motifs générés et $match$ est le nombre total de nœuds qui correspondent aux motifs retrouvés.

Parmi les travaux de Hoffmann et O'Donnell, un algorithme de RMA descendant basé sur l'automate d'Aho et Corasick [AC75] reconnaissant les chemins dans un arbre, a été présenté. Mais aucune relation entre cet algorithme et les algorithmes utilisant les automates d'arbre n'a été mise en évidence. Le travail décrit ci-dessous s'est y intéressé.

2.2.3 Algorithme de Cleophas et al. (2005)

Le travail de Cleophas et al. [CHZ05] montre qu'un automate d'arbre descendant déterministe peut être utilisé pour la recherche de chemins dans les algorithmes descendant de RMA. Les auteurs estiment que l'algorithme est original, et qu'il fournit le lien qui manquait entre les algorithmes de RMA utilisant les automates de chemins et ceux utilisant les automates d'arbre.

Dans cet article, les auteurs prouvent que même si un automate d'arbre descendant déterministe ne peut pas être utilisé pour résoudre le problème d'acceptation ou de RMA, un automate spécifique du même genre est utilisé avec une fonction pour la recherche de motifs de mots en traversant l'arbre descendant. Ils présentent une version de l'algorithme descendant de Hoffmann et O'Donnell qui utilise l'automate d'Aho et Corasick et ensuite une modification de cet algorithme en utilisant un automate d'arbre descendant déterministe associé à une fonction de sortie. Ils jugent que l'algorithme donné n'est pas nécessairement efficace. Sa complexité dans le pire des cas est égale à celle de l'algorithme naïf, c'est à dire elle est de l'ordre de $O(n.m)$ où n est la taille du motif et m est celle du texte objet.

Exemple

La figure 2.7 représente l'arborescence construite pour le motif $P = a(b(c), \nu)$ et la figure 2.8 l'automate d'Aho-Corasick pour le même motif.

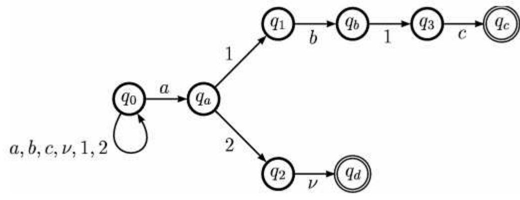


FIGURE 2.7 – Arborescence pour le motif P.

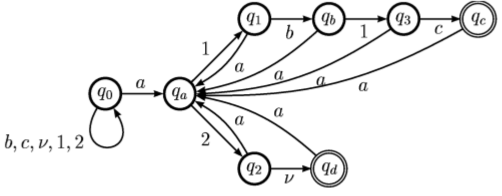


FIGURE 2.8 – Automate d'Aho-Corasick pour le motif P

Le principe de la construction de l'automate descendant et la fonction de sortie est comme suit :

– Construire l'automate d'arbre descendant déterministe reconnaissant le motif.
– Pour chaque état et chaque symbole de l'alphabet indiquant une correspondance avec un chemin (stringpath), attribuer ce chemin à la fonction de sortie, sinon la fonction ne retourne aucune valeur.
– Ajouter les transitions de boucles pour chaque symbole d'arité non nulle à l'état initial.
– Rendre cet automate déterministe en adaptant la fonction de sortie.

Pour chaque nœud visité, l'algorithme doit enregistrer les correspondances avec les motifs indiquées par la fonction de sortie pour l'état et le symbole en cours.

L'algorithme de Cleophas et al. est très similaire à celui de Hoffmann et O'Donnell, la différence entre eux réside dans le type d'automate et la fonction de sortie utilisés. Cleophas et al. estiment que les deux automates sont équivalents et qu'un des automates peut être transformé en l'autre.

Malheureusement, les auteurs de ce travail n'ont donné aucune indication sur la complexité de l'algorithme, mais vu qu'ils utilisent les mêmes structures de données de l'algorithme de Hoffmann et O'Donnell, nous nous permettons de conclure qu'ils ont la même complexité, à savoir $O(n+m.nbr_{pat})$ où nbr_{pat} est le nombre total des motifs à rechercher.

2.2.4 Algorithme de Kosaraju (1989)

Cet algorithme [Kos89] ramène la complexité de l'algorithme naïf de $\mathcal{O}(n.m)$ à une complexité de l'ordre de $O(n.m^{0.75}(\log m))$. Cette amélioration se fait par le biais de l'utilisation des notions de convolution d'arbre, suffixes d'arbre et le partitionnement d'arbres en chaînes et anti-chaînes.

Convolution de mots et convolution d'arbre

La convolution de deux séquences linéaires $a_0, a_1, ..., a_{n-1}$ et $b_0, b_1, ..., b_{m-1}$,$n \geq m$, est la séquence $c_0, c_1, ..., c_{n-1}$ où :

$$c_i = \sum_{0 \leq j \leq m-1} a_{i-j} b_j, \quad pour \quad 0 \leq i \leq n-1.$$

Supposons que l'addition et la multiplication sont des opérateurs arithmétiques et qu'une valeur négative est remplacée par 0.

Pour la convolution d'arbre, nous remplaçons la séquence a_i par un arbre A à n nœuds. Chaque nœud u est doté d'une valeur $a(u)$. Alors la convolution de A et la séquence b_i associe à chaque nœud de A une valeur $c(u)$:

$$c(u) = \sum_{0 \leq j \leq m-1} \sum_{\pi^j(v)=u} a(v) b_j.$$

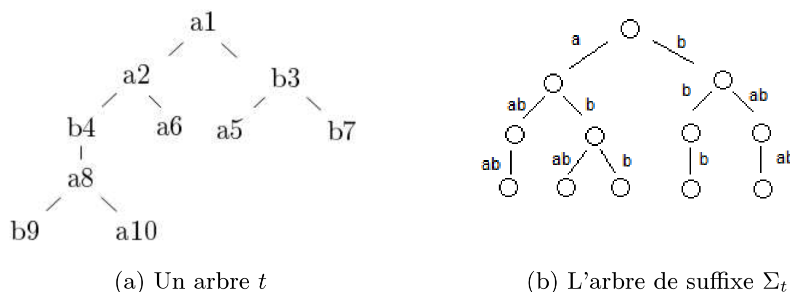
Où pour un nœud w, $\pi(w)$ est le parent de w, $\pi^j(w)$ est le $j^{\text{ième}}$ parent de w, $\pi^0(w) = W$.

L'arbre de suffixe

L'arbre de suffixe pour un mot $a_0, a_1, ..., a_{n-1}$ est la sélection des suffixes de $a_0, a_1, ..., a_{n-1}\$$ tel que $ est un symbole spécial. L'arbre de suffixe compact est obtenu en condensant l'information du mot tel que chaque nœud interne a au moins deux fils. Après cette condensation, chaque arête est étiquetée par une sous-chaîne de $a_0, a_1, ..., a_{n-1}\$$ et elle est déterminée par un intervalle dans lequel cette sous-chaîne apparaît dans le mot.

Nous définissons l'arbre de suffixe pour un arbre A comme étant la sélection de l'ensemble $\{\sigma^R_{A,v} | v \text{ est un nœud de } A\}$ et R indique que le chemin est réversible. La modification permettant d'obtenir l'arbre de suffixe compact pour les arbres est analogue à celle des mots.

La figure 2.9 montre un exemple d'arbre et son arbre de suffixe.



(a) Un arbre t (b) L'arbre de suffixe Σ_t

FIGURE 2.9 – Exemple d'un arbre et son arbre de suffixe.

Partitionnement d'arbre

Un ensemble de nœuds S d'un arbre A a la propriété de suffixe, si $\sigma_{A,S}$ pouvant être ordonné $x_1, x_2, ..., x_k$ tel que chaque x_i est un suffixe de x_{i+1} pour $i = 1..(k-1)$.

L'ensemble S a la propriété dite anti-suffixe, si $\sigma_{A,S}$ vérifie que tous les mots de cet ensemble ne sont pas des suffixes d'un autre mot.

Dans un arbre, un ensemble de nœuds S est une chaîne si S peut être ordonné $u_1, u_2, ..., u_n$ tel que chaque u_i est le père de u_{i+1} pour $i = 1..(k-1)$. L'ensemble S est une anti-chaîne si aucun nœud n'est l'ancêtre d'un autre nœud dans S.

Les étapes de l'algorithme de base

1. Pour le motif P, calculer son arbre de suffixe $\sum_p$ en $O(m \log m)$ étapes.

2. Partitionner $\sum_p$ en chaînes et anti-chaînes avec $m = a^{0.25}$. Donc, $O(m^{0.25})$ de l'ensemble sont des chaînes et $O(m^{0.75})$ sont des anti-chaînes.
3. Tout ensemble de feuilles de P ayant la propriété 'anti-suffixe' peut être traité en $O(n+m)$ étapes. Le résultat est un ensemble de nœuds de T, où pour chacun d'eux, P vérifie le motif dans ces feuilles. Cette procédure est répétée pour chacun des $O(m^{0.75})$ ensembles ayant la propriété 'anti-suffixe'. L'intersection des ensembles de nœuds de T calculés en $O(m^{0.75})$ étapes est l'ensemble des nœuds de T où pour chacun d'eux, P est restreinte à l'ensemble des feuilles de l'union des ensemble ayant la propriété 'anti-suffixe' qui correspondent (match) à T. Le calcul global requiert $O(m^{0.75}(n+m))$ étapes.
4. Tout ensemble de feuilles de P ayant la propriété de suffixe peut être traité en $O(n\sqrt{m}\log m)$ étapes. Ceci est répété $m = a^{0.25}$ toujours pour chaque ensemble de feuilles de P ayant la propriété de suffixe. Le résultat est le même que celui de l'étape 3.
5. calculer l'intersection des deux ensembles obtenus dans les étapes 3 et 4. Tout nœud de T figurant dans l'intersection est un nœud à partir duquel P correspond (match) à T. Ceci est accompli en $O(n)$ étapes.

L'auteur de cet article estime qu'il a présenté une amélioration significative de la complexité de l'algorithme naïf de la recherche de motif $O(n.m)$. Il estiment que la complexité $O(n\sqrt{m}\log m)$ peut être atteinte dans les meilleurs des cas, néanmoins la complexité présentée est de l'ordre de $O(n.m\log m)$.

2.2.5 Algorithme de Dubiner et al. (1994)

Cet algorithme [DGM94] est une amélioration de l'algorithme précédent de Kosaraju [Kos89]. Ce dernier a introduit pour la première fois les techniques de l'arbre de suffixe d'un arbre, la convolution de mot et d'arbre, ainsi que le partitionnement d'arbres en chaînes et anti-chaînes.

Bien que le présent algorithme soit largement inspiré de celui de Kosaraju, les auteurs n'utilisent aucune de ces techniques. Ils travaillent plutôt avec les arbres de suffixe tronqués. L'amélioration est obtenue par la découverte et l'exploit des mots périodiques apparaissant dans le motif P.

Dénotons par $|\alpha|$ la longueur d'un mot α. Un mot α est une période d'un autre mot β si β est un préfixe de α^k pour $k > 0$. Nous déduisons alors :

1. α est une période de β ssi $\beta = \alpha\gamma = \gamma\delta$, γ et δ étant des mots quelconques.
2. β a une période de longueur k ssi $\beta_i = \beta_{i+k}$ pour $1 \leq i \leq (|\beta| - k)$.
3. si β et $\beta\gamma$ a une période de longueur k et $|\beta| \geq k$, alors $\alpha\beta\gamma$ a une période de longueur k.

Pour chaque nœud v dans P, dénotons par $\sigma_{v,k}$ le mot des k derniers symboles de σ_v, s'ils existent bien sûr ; $\sigma_{v,k} = \sigma_v$ dans le cas $|\sigma_v| < k$. L'arbre de suffixe k-tronqué de l'arbre P, $\Sigma_{P,k}$, est défini comme étant une arborescence de l'ensemble $\{\sigma_{v,k}^R\}$ tel que v est nœud de P et R indique que le chemin est réversible. Ce qui veut dire que pour n'importe quel nœud v dans P, il existe un nœud correspondant à v' dans P et $\Sigma_{P,k}$, tel que le chemin de la racine de P, $\Sigma_{P,k}$ jusqu'à v' est l'inverse du chemin $\sigma_{v,k}$. Différents nœuds de P peuvent avoir le même nœud correspondant dans $\Sigma_{P,k}$.

La première étape de l'algorithme est de calculer l'arbre de suffixe $3l$-tronqué de P, l est un entier. Ensuite, les auteurs prouvent que pour chaque feuille v de P, σ_v a une période d'une longueur au plus l. À chaque feuille v de P, une paire unique est attribuée, contenant sa queue et sa période minimale.

L'étape suivante consiste à retrouver les correspondances entre les feuilles et les paires queue-période en utilisant la notion de chemin périodique maximal. Pour chaque chemin périodique maximal calculé dans P, une séquence est attribuée, appelée $\{0,1\}$-sequence, tel que :

$$a_i = \begin{cases} 1 & \text{si après } i \text{ périodes dans le chemin, la queue} \\ & \text{commence et se termine dans une feuille,} \\ 0 & \text{sinon.} \end{cases}$$

Donc, pour un ensemble de feuilles qui correspondent à un nœud v de l'arbre objet T, il doit y avoir un chemin périodique maximal dans T qui passe par v tel que :

1. dans le chemin de T, la position de v dans la période et la position de la racine de P dans le chemin périodique maximal de P est la même,
2. si a_i vaut 1, le b_j correspondant dans le chemin de v doit valoir 1 aussi.

Ceci veut dire que le problème de RMA est réduit à un problème équivalent de mots entre les séquences a_i et n'importe quelle séquence b_j, sur l'alphabet $\{0,1\}$.

La complexité totale de l'algorithme est estimée à $O(n\sqrt{m}\log m)$. Les preuves des calculs de complexité effectués sont données dans l'article précédemment référencé.

2.2.6 Algorithme de Chase (1987)

Cet algorithme [Cha87] est aussi une amélioration de celui de Hoffmann et O'Donnell. Les tables générées (naïvement) à partir de l'ensemble des nœuds qui correspondent aux motifs contiennent souvent des sous-tables dupliquées. Il est possible de compresser ces tables et d'utiliser un index pour convertir les numéros des motifs en indices pour la table compressée.

Si nous arrivons à déterminer ces sous-tables, alors pour chaque symbole et chaque fils de ce symbole un index est créé pour permettre la transformation de ces ensembles des nœuds qui correspondent aux motifs avec les sous-tables en un seul ensemble. Cependant, il est possible de compresser les tables avant même qu'elles ne soient générées, ceci se fait en examinant les ensembles des motifs retrouvés.

Principe de compression des tables

Définissons PA_j comme étant l'ensemble des motifs qui apparaissent dans le $j^{\text{ième}}$ fils des motifs avec l'étiquette A dans PF (PF est le pattern forest qui est l'ensemble des motifs P combiné avec tous leurs sous-motifs). PF est fermé par rapport aux sous-arbres, $PA_j \in PF$.

Avant de donner le principe de compression des tables de recherche de motifs, nous rappelons l'algorithme de Hoffmann pour le calcul des ensembles des nœuds qui correspondent aux motifs.

$M(t) =$

Soit $U = \begin{cases} \{\upsilon\} & \text{si } \upsilon \in PF \\ \{\} & \text{sinon} \end{cases}$

Si $t = A[]$ alors

Si $(t = A[] \in PF)$ alors $U \cup \{A[]\}$

Sinon U

Sinon

soit $A[t_1, ..., t_n] = t$

soit $MSP = M(t_1) \times ... \times M(t_n)$

$$PF \cap \left(U \cup \bigcup_{(p_1,...p_n) \in MSP} \{A[p_1, ..., p_n]\} \right)$$

$M(t)$ retourne tous les motifs dans PF qui correspondent à t. Pour une feuille t, si $t \in PF$ alors $M(t)$ est l'ensemble $\{t, \upsilon\}$ (ou $\{t\}$ si υ n'est pas dans PF). Sinon (t n'est pas dans PF) $M(t)$ est $\{\upsilon\}$ (ou $\{\}$). Dans les deux cas $M(t)$ est l'ensemble des motifs dans PF qui correspondent à la feuille t.

Si t n'est pas une feuille, M est appliquée à chaque fils de t pour produire leurs ensembles de nœuds qui correspondent aux motifs. Le produit de ces ensembles est utilisé pour construire MSP qui est un ensemble de tuples de motifs. Chaque tuple $(p_1, ..., p_n)$ dans MSP est une combinaison indépendante de motifs qui correspondent aux fils de t. Donc l'arbre $t(p_1, ..., p_n)$ forme un nouveau motif qui correspond à t.

Cette opération est effectuée pour tous les éléments de MSP afin de construire un ensemble de motifs qui correspondent à t. Pour enlever les motifs superflus, nous ferons l'intersection de cet ensemble avec PF.

Calculons maintenant l'entrée de la table de recherche de motifs pour un tuple de numéro d'ensembles donné. Pour les ensembles $(R_1, ..., R_n)$ et l'étiquette A, l'ensemble R attribué à cette entrée de la table selon l'algorithme de Hoffmann est le résultat de

$$R = PF \cap \left(U \cup \bigcup_{(p_1,...,p_n) \in MSP} \{A[p_1, ..., p_n]\} \right).$$

Où $MSP = R_1 \times ... \times R_j \times ... \times R_n$.

Remplaçons maintenant l'ensemble R_j par l'ensemble $R_j \cap P_{Aj}$. Ceci ne va pas changer la valeur de R car les motifs qui n'appartiennent pas à l'intersection $R_j \cap R_{Aj}$ sont ceux qui ne sont pas le $j^{\text{ième}}$ fils du nœud A dans PF. Donc, n'importe quel tuple de motifs enlevé de MSP ne va pas produire un motif qui est dans PF. Notons que ceci est valable pour n'importe quelle valeur de j.

S'il existe un autre ensemble R'_j tel que $R'_j \cap P_{Aj} = R_j \cap P_{Aj}$, alors R'_j et R_j produiront les mêmes ensembles des nœuds qui correspondent aux motifs après la substitution définie précédemment. Cette observation est le principe de compression de la table des matchs.

L'intersection avec P_{Aj} produit une relation d'équivalence sur $\mathcal{R}$: deux ensembles de motifs R et R' sont Aj-équivalents si $R \cap P_{Aj} = R' \cap P_{Aj}$. Pour tout symbole A et ses fils j, l'ensemble des ensembles des nœuds qui correspondent aux motifs $\mathcal{R}$ est partitionné en un ensemble des ensembles de nœuds qui correspondent aux motifs équivalents.

Exemple

Pour l'ensemble des motifs $\{a(a(b, v), b), a(a(v, c), c)\}$ et la forêt des motifs $PF = \{a(a(b, v), b), a(a(v, c), c), a(b, v), a(v, c), b, c, v\}$, nous calculons les ensembles des nœuds qui correspondent aux motifs : $M_1 = \{v\}$,

$M_2 = \{b, v\}$,
$M_3 = \{c, v\}$,
$M_4 = \{a(b, v), a(v, c), v\}$,
$M_5 = \{a(v, c), v\}$,
$M_6 = \{a(b, v), v\}$,
$M_7 = \{a(a(v, c), c), a(v, c), v\}$,
$M_8 = \{a(a(b, v), b)\}$.

La table du symbole A est :

Fils gauche \ Fils droit	1	2	3	4	5	6	7	8
1	1	1	5	1	1	1	1	1
2	6	4	6	6	6	6	6	6
3	1	1	5	1	1	1	1	1
4	1	8	7	1	1	1	1	1
5	1	1	7	1	1	1	1	1
6	1	8	5	1	1	1	1	1
7	1	1	7	1	1	1	1	1
8	1	1	5	1	1	1	1	1

Et enfin θ_A la table compressée de A : l'index pour le symbole A avec le $j^{\text{ième}}$ fils est μ_{Aj}.

μ_{A2}

1	2	3	4	5	6	7	8
1	2	3	1	1	1	1	1

μ_{A1}

1	1
2	2
3	1
4	3
5	4
6	5
7	4
8	1

θ_A 1 \ 2	1	2	3
1	1	1	5
2	6	6	4
3	1	8	7
4	1	1	7
5	1	8	5

Chase présente aussi dans cet article d'autres améliorations de l'algorithme de Hoffmann qui concernent les itérations de calcul des ensembles des nœuds qui correspondent aux motifs. Des études expérimentales ont été effectuées sur l'algorithme de base ainsi que sur les améliorations apportées. Ces études ont montré qu'une compression extraordinaire peut être accomplie. L'article présente les résultats obtenus par ces expérimentations sur des benchmarks où il a atteint un taux de compression de 49%.

À la fin de l'article, Chase affirme que son travail apporte une bonne amélioration de l'algorithme de Hoffmann et O'Donnell bien qu'il n'a pas prouvé cela formellement. Il estime que malgré l'efficacité de son algorithme par rapport à celui de Hoffmann et O'Donnell, il est difficile d'en borner le temps et l'espace requis vu que ça dépend de la taille et la nature des motifs.

2.2.7 Algorithme de Ramesh et Ramakrishnan (1992)

Les auteurs de cet article [RR92] présentent un nouvel algorithme séquentiel de RMA dans les arbres qu'ils estiment plus performant que plusieurs algorithmes concernant le temps d'exécution et la facilité d'intégration. La complexité théorique de l'algorithme est linéaire à la somme des tailles des deux arbres (le motif et le texte objet).

Pour calculer la complexité de l'algorithme, les auteurs utilisent la terminologie suivante : $\mathcal{K}(p)$ pour dénoter l'ensemble de tous les chemins de la racine vers les feuilles du motif P dont les feuilles sont étiquetées par des variables. K est le nombre total des occurrences de variables en P. Alors $|\mathcal{K}(p)| = K$.

L'algorithme se déroule en deux phases : une phase de prétraitement et une autre de recherche de motifs. Le temps de prétraitement est de l'ordre de $O(m)$ qui est le même que celui de l'algorithme de Hoffmann.

Cet algorithme n'utilise pas la structure arborescente du motif et de l'arbre objet, mais plutôt la représentation en chaîne de caractères ainsi que la chaîne d'Euler. Plus particulièrement, pour vérifier si un motif d'arbre correspond à un nœud dans l'arbre objet, il vérifie alors si leurs chaînes d'Euler sont identiques après le remplacement des variables dans le motif par les chaînes d'Euler des arbres adéquats. Pour s'assurer de la validité de cette approche, il faut retrouver la relation qui existe entre la chaîne d'Euler et la structure de l'arbre. Ceci est détaillé dans l'article avec les preuves nécessaires.

Cette approche de RMA est basée sur la linéarisation des arbres en utilisant leurs chaînes d'Euler. Ces chaînes sont obtenues comme suit : Pour chaque arrête (v_i, v_j) de l'arbre, créer une nouvelle arrête (v_j, v_i). Ensuite, classer les nœuds de cette structure en suivant un parcours préfixé. Formellement :

- la chaîne d'Euler d'un arbre feuille $t = a$ est l'ordre du nœud a ;
- si t est un arbre et $t_1, t_2, ..., t_n$ sont des sous-arbres dans les racines $r_1, r_2, ..., r_n$, alors la chaîne d'Euler de t est $r_1e_1r_2e_2...r_ne_n$ où e_i est la chaîne d'Euler du sous-arbre t_i dans la racine r_i.

Par exemple, pour l'arbre ordonné de la figure 2.10, sa chaîne d'Euler est 12425213686963731.

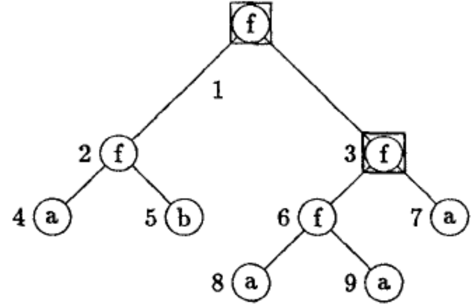


FIGURE 2.10 – Exemple d'arbre ordonné.

D'après cette définition de la chaîne d'Euler, les propriétés suivantes peuvent être déduites :

- Les feuilles d'un arbre apparaissent une seule fois dans la chaîne ;
- La sous-chaîne d'Euler entre la première et la dernière occurrence d'un nœud est la chaîne d'Euler du sous-arbre dont la racine est ce nœud ;
- un nœud v qui a k fils apparait $k+1$ fois dans la chaîne d'Euler, et la chaîne d'Euler du sous-arbre dont la racine est son $i^{\text{ième}}$ fils est la sous-chaîne entre la $i^{\text{ième}}$ et $(i+1)^{\text{ième}}$ occurrence de v.

Le mot d'Euler d'un arbre est obtenu en remplaçant tous les nœuds dans la chaîne d'Euler par leurs étiquettes. Par exemple, le mot d'Euler pour l'arbre précédent (figure 2.10) est *ffafbffffafaffaff*.

Détail de l'algorithme

D'un point de vue conceptuel, la recherche de motifs linéaire peut être accomplie en deux phases. Dans la première phase, une recherche de motifs linéaire est effectuée et le résultat est un ensemble de nœuds de l'arbre objet qui correspondent aux motifs. Pour chaque nœud de l'ensemble, la cohérence de la substitution calculée pour les variables identiques n'est pas vérifiée. Ceci est accompli dans la seconde phase.

Supposons C_s et C_p les chaînes d'Euler de l'arbre objet et le motif respectivement. E_s et E_p leur mots d'Euler et $|E_s| = O(n)$ et $|E_p| = O(m)$ dénotent leurs tailles respectives. Supposons que le motif P contient k variables que nous dénotons $v_1, v_2, ..., v_k$.

Puisque seulement les feuilles peuvent être des variables et chaque feuille apparait exactement une fois dans le mot d'Euler E_p, donc chacune de ces variables apparait exactement une fois dans E_p.

E_s est rangé dans un tableau. Ensuite E_p est fractionné en $k+1$ mots de motif en lui ôtant les k variables. Ces mots sont dénotés $\sigma_1, \sigma_2, ..., \sigma_{k+1}$.

Par exemple, pour le motif $P' = f(f(a, X), Y)$, la suppression des variables X et Y de son mot d'Euler donne les trois motifs de mots suivants : $\sigma_1 = ffaf$, $\sigma_2 = ff$ et $\sigma_3 = f$. Après, nous construisons les tables booléennes $M_1, M_2, ..., M_k$, chacune d'elles contient $|E_s|$ entrées. S'il existe un nœud qui correspond aux motifs pour σ_i dans E_s dans la position j, alors la $j^{\text{ième}}$ entrée de M_i contient 1 sinon elle contient 0.

Étant donné que σ_1 est un préfixe de E_p, il est évident que l'ensemble des nœuds où p correspond à s est un sous-ensemble des nœuds qui correspondent aux motifs pour σ_1 dans E_s. Nous cherchons alors le motif qui correspond uniquement dans les entrées non nulles de M_1 qui correspond à la première occurrence d'un nœud.

Supposons que i est cette entrée. Pour déterminer si un motif correspond à i nous procédons comme suit : $i + |\sigma_1| - 1$ est la position dans E_s où le nœud qui correspond aux motifs σ_1 se termine.

Comme E_p est $\sigma_1 v_1 \sigma_2 v_2 ... \sigma_k v_k \sigma_{k+1}$, la substitution de v_1 doit être un sous-mot de E_s qui commence de la position $i + |\sigma_1|$. Cette position doit correspondre à la première occurrence d'un nœud, disant x, (car seulement les variables peuvent être substituées par un sous-arbre) dans C_s. Sinon, il ne peut pas y avoir de nœud qui correspond aux motifs pour le motif dans la position i.

En plus, si cette condition est vérifiée, alors la substitution de v est un sous-mot de E_s entre les positions qui correspondent à la première et la dernière occurrence de x dans C_s. Supposons la dernière occurrence de x à la $j^{\text{ième}}$ position de C_s. Alors, nous commençons la recherche de motif pour σ_2 à partir de la $(j+1)^{\text{ième}}$ position dans E_s. Évidement, $M_2[j+1]$ est une entrée non nulle. De la même façon nous calculons la substitution pour $v_2, v_3, ..., v_k$. En cas de réussite, nous concluons que E_p correspond à E_s dans la position i.

Dans cet algorithme, $O(k)$ tables doivent être construites, et chacune ayant $|E_s|$ entrées. Donc, une implémentation directe implique une complexité de $O(n.k)$. Pour améliorer ce temps, ces tables ne doivent pas être calculées explicitement.

Pour estimer la complexité de cet algorithme les définitions suivantes sont nécessaires.

Définition 12. *Soit Ψ un ensemble fini de mots, $|\Psi| = \eta$. Le nombre de suffixe de $\lambda \in \Psi$ est le nombre de mots de Ψ qui sont suffixes de λ. L'index de suffixe de Ψ noté η^* est le maximum parmi tous les nombres de suffixes des mots de Ψ. Si $\eta = 0$ alors $\eta^* = 1$.*

Définition 13. *Soit $\mathcal{L}(p)$ l'ensemble des chemins de p et l le nombre de feuilles de p. Il est évident que $|\mathcal{L}(p) = l|$. Soit k le nombre de feuille variables de p et $\mathcal{K}(p)$ l'ensemble des chemins de p se terminant par des variables. Nous avons $|\mathcal{K}(p) = k|$.*

D'autre part, nous avons $k \leq l$, donc $k^* \leq l^*$.

Les auteurs de l'article estime la complexité de l'algorithme à $O(nk^*) \leq O(nl^*)$.

2.2.8 Algorithme de Cole et al. (1999, 2003)

Cole et Ramesh ont publié plusieurs travaux [CHI99][CH03] dans la RMA : dont un en 1999 qui présente un algorithme aléatoire d'un temps de l'ordre de $O(n \log^3 m)$, et un autre en 2003 proposant un algorithme linéaire pour le même problème.

Dans le premier article, les auteurs introduisent la notion de 'subset matching'. Ce problème consiste à retrouver toutes les occurrences d'un motif de mot p de longueur m dans un texte de mots t de longueur n, où chaque emplacement d'un motif ou d'un texte est un ensemble de caractères faisant partie d'un alphabet. Le motif apparait dans le texte à la position i si l'ensemble $p[j]$ est un sous-ensemble de l'ensemble $t[i + j - 1]$, pour tout $1 \leq J \leq m$.

Les auteurs donnent un algorithme de l'ordre de $O((s+n) \log^2 m \log s + n)$, où s dénote la somme des tailles de tous les ensembles. Ensuite, le problème de RMA est réduit en instances de problème de 'subset matching'. Cette réduction est linéaire et la somme des tailles des problèmes de 'subset matching' qui est lui aussi linéaire. Donc, avec la complexité précédente, le résultat est un algorithme de l'ordre $O(n \log^2 m \log n)$.

Le présent travail montre qu'un problème de RMA peut être réduit à un problème dit de 'spine pattern matching'. Dans ce dernier, il existe un chemin spécial dans le motif et un autre dans le texte objet appelés leurs épines (spines). L'épine d'un arbre commence de sa racine et chaque nœud de l'épine a au moins un fils n'appartenant pas à cette épine. La recherche des correspondances est restreinte à ce que l'épine du motif doit correspondre à une portion de l'épine du texte. Autrement dit, l'épine du motif doit être un nœud qui correspond aux motifs de celle de l'arbre.

La stratégie de leur approche se résume en deux concepts :

1. La réduction du problème de RMA à un problème de spine pattern matching : l'idée générale est de rechercher la correspondance du chemin central du motif dans celui du texte et ensuite rechercher les sous-arbres reliés à ce chemin dans les sous-arbres appropriés du texte.
2. La réduction du spine pattern matching à un subset matching : ceci est simple et dû à l'observation que les nœuds des sous-arbres à recher-

cher peuvent être identifiés par des caractères au lieu d'une structure arborescente.

Ces concepts sont utilisés pour l'algorithme de subset matching qui s'effectue en deux étapes :

* la définition du problème pour un caractère a pour devenir le motif à rechercher dans chaque sous ensemble S remplacé par $S \cap \{a\}$,
* les motifs recherchés sont simplement l'intersection des ensembles de motifs obtenus en appliquant la première étape sur chaque caractère.

2.2.9 Algorithme de Polách et al. (2011)

Cet algorithme a été présenté par Radomír Polách dans son mémoire de Master [Pol11]. Il s'agit de construire un automate reconnaissant un langage à partir d'une expression régulière d'arbre. Ce travail est semblable à celui de Thompson pour les mots [Tho68], et aussi à notre approche proposé dans le chapitre suivant. La différence réside dans le type d'automate utilisé. Ici, l'auteur construit un nouveau type d'automate d'arbre : les automates à pile ou pushdown automata.

L'auteur construit son automate à pile d'une façon inductive en construisant les automates pour les opérations effectuée sur les expressions régulières : l'union, la concaténation, la fermeture. Les différentes constructions de l'automate à partir des expressions de base sont données.

L'auteur utilise une notation en barre (bar notation) dont le principe de base est le suivant :

– la racine d'un sous-arbre est toujours derrière la parenthèse gauche dans la notation préfixée ;
– la racine d'un sous-arbre est toujours avant la parenthèse droite dans la notation post-fixée ;

Cette observation démontre qu'il existe une parenthèse gauche (droite) redondante dans la notation préfixée (post-fixée). La notation en barre réduit dans les deux cas le nombre de symboles (parenthèses de droite et de gauche) en n'utilisant qu'un seul symbole pour définir les liens entre les nœuds d'un arbre. Ce symbole est la barre $|$.

La construction de la notation en barre d'une expression régulière d'arbre est comme suit :

– pour une feuille a, $pref_bar(a) = a|$ et $post_bar(a) = |a$;
– pour un arbre $t = a(b_1, b_2, ..., b_n)$ on a :

$$\begin{cases} pref_bar(t) & = & a.pref_bar(b_1).pref_bar(b_2)...pref_bar(b_n)| \\ post_bar(t) & = & |post_bar(b_1).post_bar(b_2)...post_bar(b_n).a \end{cases}$$

Notons que le nombre de barres dans une notation en barre d'un arbre t est toujours égal au nombre de symboles dans t.

Construction de l'automate

La construction d'un automate à pile à partir d'une expression régulière d'arbre est basée sur les règles suivantes :

- le symbole ε ne modifie jamais la pile A ;
- le symbole $|$ toujours empile le symbole lu dans la pile A ;
- les symboles $a_1, a_2, ..., a_n \in \Sigma$ toujours dépilent un symbole de la pile A.

Conversion de la notation en barres en automate à pile : Ces règles produisent uniquement des transitions silencieuses, c'est à dire qu'elles n'affectent pas la pile. Elles sont notées par $\varepsilon|\varepsilon \mapsto \varepsilon$. Aucun symbole n'est lu, empilé ou dépilé.

1. Conversion de ε : $\varepsilon \mid \varepsilon \mapsto \varepsilon$.
2. Conversion de $a \in \Sigma$: $a \mid t \mapsto \varepsilon$.
3. Conversion de $|$: $| \mid \varepsilon \mapsto t$.

Les figures suivantes schématisent ces différentes conversions.

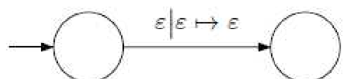


FIGURE 2.11 – Conversion de ε

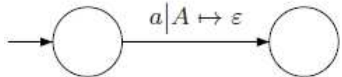


FIGURE 2.12 – Conversion de $a \in \Sigma$

FIGURE 2.13 – Conversion de $|$

L'union

Pour l'automate d'union de deux automates A et B, deux nouveaux états sont créés, un état représentant le nouvel état initial et l'autre le nouvel état final. Des transitions entre le nouvel état initial et tous les autres états initiaux sont ajoutées. La même chose est faite pour les états finaux. Ces transitions sont silencieuses : aucun symbole n'est lu, empilé ou dépilé.

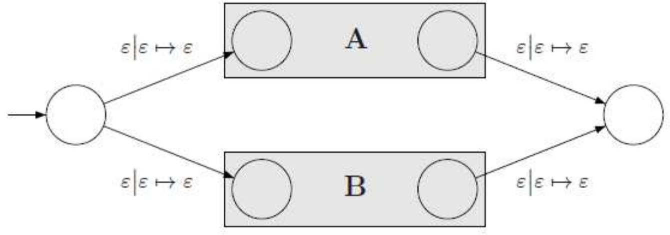


FIGURE 2.14 – Automate de l'union

Exemple : Pour les deux arbres $t_1 = a(b)$ et $t_2 = c$, les figures suivantes montrent la construction de l'automate d'union de t_1 et t_2 ainsi que la déterminisation de l'automate résultant.

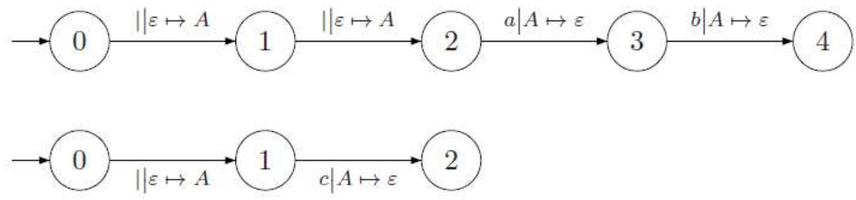


FIGURE 2.15 – Automate t_1 et t_2.

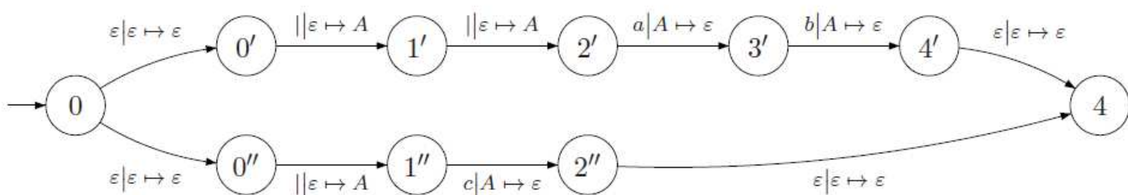


FIGURE 2.16 – Automate de l'union $t_1 \cup t_2$.

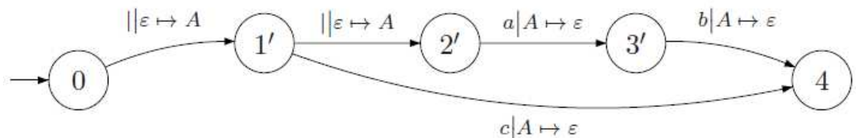


FIGURE 2.17 – Automate déterministe de l'union $t_1 \cup t_2$.

La concaténation

L'automate de concaténation de deux automates A et B dans le symbole de concaténation c, noté $A_{.c}B$ est construit à partir des automates de A et B en fractionnant chaque transition du symbole c en deux transitions. Une transition de l'état initial de la transition à l'état initial de B et une autre transition de l'état final de B à l'état final de la transition.

Les transitions de symbole de concaténation c divisent l'expression régulière d'arbre A en plusieurs parties A, A', A'', ... Toutes les transitions de A^x vers B sont de la forme $\varepsilon|A \mapsto A^x$ et toutes les transitions de B vers A^x sont de la forme $\varepsilon|A^x \mapsto A$.

Les transitions ajoutées sont toujours silencieuses, elles n'affectent pas la taille de la pile : aucun symbole n'est lu, empilé ou dépilé.

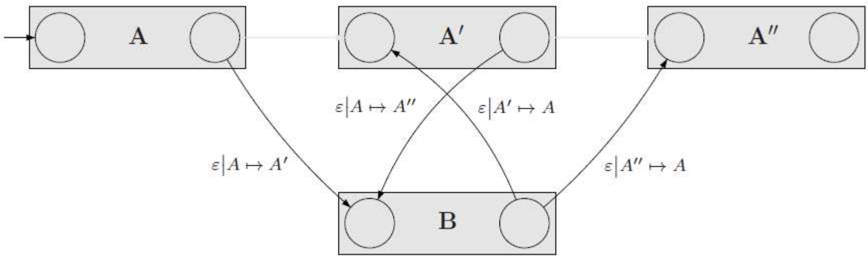


FIGURE 2.18 – La concaténation de deux automates

L'itération ou la closure

Pour un automate A construit à partir d'une expression régulière d'arbre, l'automate qui correspond à l'opération de closure A^{*c} est créé en remplaçant toutes les transitions du symbole de concaténation c par des transitions silencieuses $\varepsilon|\varepsilon \mapsto \varepsilon$ et en ajoutant deux nouvelles transitions. Une transition de l'état initial de la transition à l'état initial de A et une autre transition de l'état final de A à l'état final de la transition.

La même chose avec la concaténation, les transitions de symbole de concaténation c divisent l'expression régulière d'arbre A en plusieurs parties A, A', A'', ...

Toutes les transitions entre les nouvelles partitions A^x sont de la forme $\varepsilon|A^x \mapsto A$ et toutes les transitions entre A et A^x sont de la forme $\varepsilon|A \mapsto A^x$ et elles sont toutes silencieuses.

Et comme l'opération d'itération accepte l'arbre vide, il faut ajouter une autre transition silencieuse $\varepsilon|\varepsilon \mapsto \varepsilon$ de l'état initial de A vers l'état final de A.

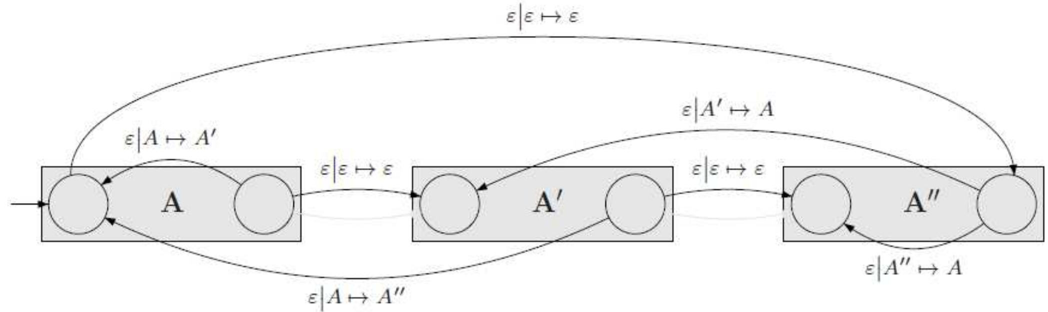


FIGURE 2.19 – Automate de l'opération d'itération.

Principes de base de construction de l'algorithme de l'automate à pile.

1. Vérification syntaxique de l'expression régulière d'arbre.
2. Transformation de l'expression régulière en forme post-fixée.
3. Si le symbole lu est un symbole de l'alphabet Σ, créer un fragment d'automate fini et empiler ce symbole.
4. Si le symbole est un opérateur ($|, \cup, ...$), dépiler les opérandes de la pile, effectuer l'opération indiquée par l'opérateur et empiler le résultat.
5. Répéter ces opérations jusqu'à ce qu'il n'y aura plus de symbole à lire.

Après la construction de l'automate à pile de Thompson de l'expression régulière, la RMA s'effectue de la même manière que celle des mots. C'est à dire que le texte objet est parcouru en utilisant l'automate ainsi construit et à chaque état final atteint, un motif de l'expression régulière est identifié.

La complexité de la recherche de motifs en utilisant l'expression régulière d'arbre inclue la complexité de traitement de l'expression régulière, la construction de l'automate à pile et ensuite le rendre déterministe, et enfin le déroulement de l'automate sur un texte objet.

La complexité de construction de l'automate est comparable à celle des mots, elle est de l'ordre de $O(2^n)$, ceci est expliqué par le fait que le nombre de choix à effectuer est toujours 2. La recherche de motifs en utilisant un automate déterministe est aussi comparable à celle des mots. La complexité qui reste est celle de le déterminisation de l'automate à pile. Il a été démontré dans des travaux antérieurs qu'elle est de l'ordre de $O(2^{n^2}).|\Sigma_c|$.

2.3 Synthèse et Discussion

Dans le tableau suivant nous présentons une synthèse des différents algorithmes de RMA étudiés dans la section précédente. Cette étude comparative porte sur les critères suivants : l'entrée de l'algorithme de recherche, la technique utilisée pour effectuer la recherche, le principe de cette technique et enfin la complexité.

- n est la taille du motif,
- m est la taille de l'arbre objet,
- $match$ est le nombre total de nœuds qui correspondent aux motifs retrouvé,
- nbr_{pat} est le nombre de motifs à rechercher,
- ht est la taille d'un arbre construit pendant le pré-traitement,
- k représente le nombre de feuilles de P et l le nombre de variables de P,
- MS dénote la technique du match set.

Notons que les complexités mentionnées dans le tableau suivant représentent les complexités pire cas. De plus, ces complexité englobent le temps de construction des structures de travail (automates, ensembles match–set, ...) et le temps de la recherche de motifs.

	Algorithme	Entrée	Technique utilisée	Principe	Complexité
	Naïf	Arbre	–	Comparaison nœud par nœud.	$O(n.m)$
Automate	Hoffman & O'Donnell	Arbre	Automate descendant	Transformer l'arbre en chemins puis construire l'automate AC reconnaissant ces chemins.	$O(n) + O(m + match)$
	Cleophas et al.	Arbre	automate d'arbre descendant	Même étapes que celui de H&O mais en construisant un automate d'arbre.	–
MS	Hoffman & O'Donnell	Grammaire	Automate ascendant	Construire pour chaque nœud de l'arbre objet l'ensemble de tous les sous-arbres qui correspondent au motif, ensuite créer une table pour chaque symbole de Σ regroupant les matchs retrouvés.	$O(n.ht^2) + O(m + match)$
	Chase	Grammaire	Compression	Compression des tables de l'ex algorithme en éliminant les entrées dupliquées.	–
Stringpath	Ramesh et Ramakrishnan	Arbre	chaînes et mots d'Euler	Transformer l'arbre objet et le motif en chaînes d'Euler et vérifier la correspondance entre eux.	$O(nk^*) \leq O(nl^*)$
	Cole et al.	Arbre	subset matching, spine pattern matching	Utilisation de la permutation de caractère pour créer un nouvel arbre et nouveau motif.	$O(m \log^2 m \log s + n + O(m \log m))$
Autres	Kosaraju	Arbre	Convolution, suffixes d'arbre, chaînes et anti-chaînes	Calculer l'arbre de suffixe, partitionner le en chaînes et anti-chaînes et puis vérifier la propriété anti-suffixe.	$O(n.m^{0.75}(\log m))$
	Dubiner et al.	Arbre	Arbres de suffixe tronqués	Calculer l'arbre de suffixe tronqué du motif, attribuer les paires période-queue, et ensuite identifier les chemins périodiques maximaux.	$O(n\sqrt{m} \log m)$
	Polach et al. (Thompson à pile)	Expression régulière d'arbre	Automates d'arbre à pile	Construction par induction d'un automate d'arbre reconnaissant l'expression régulière d'entrée.	$O(2^n)$

Les premiers algorithmes parus dans la littérature pour la RMA sont ceux de Hoffmann et O'Donnell (1982), ce qui explique en grande partie le fait que plusieurs travaux de recherche se sont basés sur ces algorithmes, soit en améliorant simplement la complexité comme a fait Chase en compressant les tables générées pour les nœuds de l'arbre objet, ou en utilisant le principe de base qui est la construction d'un automate reconnaissant les motifs comme le montre le travail de Cleophas et al.

Les travaux de Hoffmann et O'Donnell et ceux qui se sont inspirés d'eux sont basés sur le calcul de l'ensemble des correspondances au motif (les match-sets) ou sur la construction d'automate.

Une autre catégorie d'algorithme de recherche de motif utilise la notion de match set. L'idée de base de ces algorithmes est de construire toutes les correspondances possibles pour un nœud donné, ensuite les coder et les classer en tables. À chaque symbole de l'alphabet une table est attribuée, et la dimension de cette dernière n'est autre que l'arité du symbole concerné. C'est la version ascendante de l'algorithme de Hoffmann et O'Donnell qui a initié cette notion. Ensuite, les travaux de Chase [Cha87] ont contribué à son amélioration en procédant à la compression de ces tables.

Une troisième catégorie d'algorithme de recherche de motif est basée sur la décomposition de l'arbre objet et l'arbre motif en chemins (stringpaths). Parmi les travaux dans cette catégorie, nous avons présenté ceux de Ramesh et Ramakrishnan (1992) et Cole et al (1997, 1998, 2003).

Le premier algorithme procède à la linéarisation des arbres objet et motif en les transformant en chaînes de caractères, ayant certaines propriétés, appelées dans la théorie des langages des mots les chaînes d'Euler. Le problème de RMA consistera dans ce cas à vérifier la correspondance entre les deux chaînes d'Euler. Le second travail introduit les techniques dites de 'subset pattern matching' et 'la recherche de motif d'épine' (spine pattern matching).

D'autres algorithmes de recherche de motif ne faisant partie d'aucune des trois catégories présentées précédemment existent. Ces algorithmes utilisent plutôt des méthodes modifiant la structure de l'arbre objet et l'arbre motif. Nous avons présenté l'algorithme de Kosaraju [Kos89] qui a introduit les notions d'arbre de suffixe, chaînes et anti-chaînes pour les arbres. Dubiner et al. ont présenté une amélioration de cet algorithme en utilisant la notion d'arbre de suffixe tronqué.

Le dernier algorithme présenté est une généralisation de celui de Thompson pour les mots. Il consiste à créer un automate d'arbre à pile à partir de l'expression régulière du motif, et ensuite de dérouler l'arbre objet sur cet automate pour reconnaitre les motifs existants. C'est l'algorithme le plus

proche de notre contribution, la différence réside dans le type d'automate construit.

L'algorithme descendant de Hoffmann et O'Donnell que nous avons présenté dans la première catégorie, celle des algorithmes utilisant les automates, a été classé dans la taxonomie de [Cle08] comme un algorithme utilisant les chemins (stringpaths). En effet, cet algorithme décompose l'arbre de motif en chemins avant de construire l'automate d'Aho-Corasick pour la recherche de motif.

Dans la catégorie des algorithmes utilisant les automates, Cleophas a présenté une méthode permettant de construire un automate d'arbre depuis l'arbre motif. Les états de cet automate sont tous les sous-arbres possibles du motif, et les transitions représentent les liens entre ces états selon la relation entre un arbre et ses sous-arbres.

En terme de complexité, nous remarquons que les algorithmes utilisant les techniques de construction de l'automate d'Aho-Corasick et de calcul des ensembles des 'match-set' sont les plus performants, mais leurs inconvénients c'est qu'elles sont difficiles à implémenter, et l'espace occupé pour le traitement des tables et automates est important.

Dans la seconde position arrivent les algorithmes se basant sur le partitionnement des arbres en chaînes et l'exploitation des périodes dans ces chaînes.

Les algorithmes utilisant la notion de chemins sont moins performants que ces derniers, mais ils ont l'avantage d'être les plus faciles à implémenter car la recherche de motif dans ce cas traite des chaînes de caractères.

L'algorithme le moins performant en terme de complexité dans les pires des cas est celui de Thompson. C'est un algorithme connu pour être exponentiel, mais qui peut être borné par une complexité linéaire en majorant les transitions à effectuer pour un état quelconque. Cet algorithme est le plus facile à implémenter bien qu'il est non déterministe.

2.4 Conclusion

Dans ce chapitre nous avons présenté quelques algorithmes parmi les plus connus dans le domaine de la RMA. Nous avons remarqué que la plupart des algorithmes dans la littérature sont plus ou moins une amélioration de l'algorithme de Hoffmann et O'Donnell. Certains algorithmes utilisent les mêmes structures de données alors que d'autres les modifient.

De part l'analyse que nous avons effectué sur la complexité ainsi que le degré de difficulté de l'implémentation, nous ne pouvons pas énoncer qu'un algorithme est meilleur qu'un autre, chacun d'eux possède des avantages et des inconvénients. L'algorithme de Thompson avec un automate d'arbre

à pile est le plus proche de notre contribution dans la RMA, la différence avec notre approche réside dans le type d'automate construit. Nous utilisons plutôt un automate d'arbre des langages réguliers.

La quasi majorité des algorithmes de RMA présentés dans ce chapitre permet d'effectuer la recherche d'un motif ou un ensemble fini de motifs. Cependant, nous visons un problème de RMA plus général qui démarre d'un motif exprimé par une expression régulière. Notre approche est décrite dans le chapitre suivant.

Chapitre 3

Notre approche de recherche de motifs d'arbre

Dans ce chapitre nous allons présenter notre contribution dans la RMA. Avant de développer les principes de base de notre construction ainsi que l'algorithme de recherche, nous commençons par donner une description sommaire de notre approche de recherche de motifs puis nous présentons un rappel sur l'automate de Thompson pour les mots : son principe de fonctionnement, ses différentes constructions et sa complexité. Ensuite, les preuves de correction de notre approche sont établies. La dernière section de ce chapitre est réservée à la discussion de la complexité de notre algorithme.

3.1 Vue générale de l'approche proposée

Soit E une expression régulière d'arbre exprimant un langage de motifs. Soit t un arbre objet. Le problème que l'on se propose de résoudre est de rechercher tous les nœuds de t qui correspondent au motif E. La figure 3.1 schématise le principe de fonctionnement de notre approche de RMA. Cette approche est divisée en deux phases : une phase de construction de l'automate de Thompson d'arbre à partir d'une expression régulière d'arbre représentant le motif et une seconde phase de parcours de l'automate construit sur l'arbre objet.

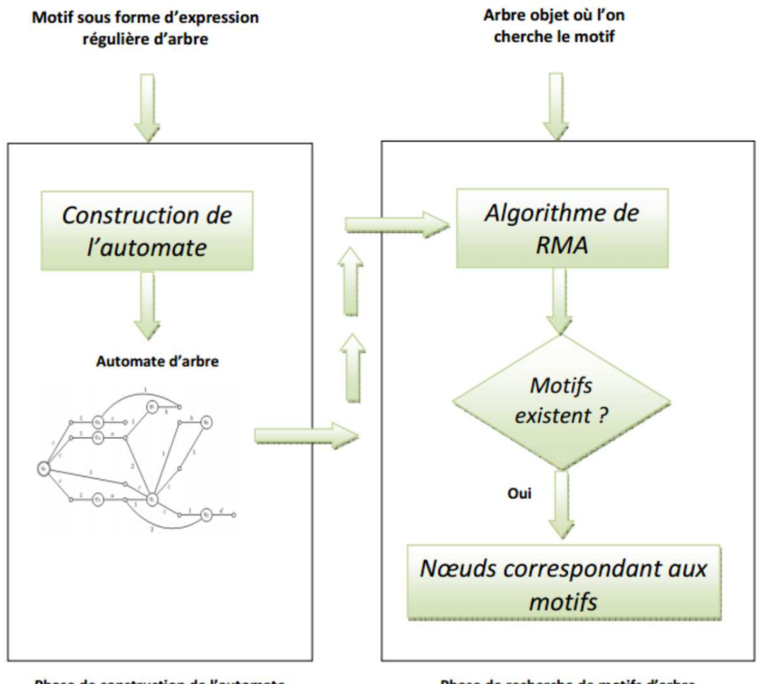


Figure 3.1 – Principe de notre approche de RMA.

3.2 Rappel de l'automate de Thompson pour les mots

Dans son article de 1968, Regular expression search algorithm, Ken Thompson décrit une technique permettant de construire un automate fini non déterministe représentant une expression régulière donnée et d'en simuler l'exécution [Tho68].

Selon la technique décrite par Thompson, l'expression régulière doit d'abord être convertie en notation post-fixée. Une fois cette étape accomplie, l'automate est construit par la composition successive de fragments d'automate, la procédure est décrite dans la section suivante. Pour ce faire, Thompson emploie deux piles, la première contenant l'expression régulière précédemment convertie en notation post-fixée et une seconde contenant les fragments d'automate. Bien que cette méthode soit parfaitement fonctionnelle, il est possible de remplacer la conversion explicite de l'expression régulière en notation post-fixée et l'utilisation des deux piles par une descente récursive. Cette seconde approche est certainement plus naturelle et surtout plus expressive.

Une fois l'automate construit, la tâche consiste simplement à le visiter en conservant, pour chaque étape de calcul, l'ensemble des états atteignables. L'ensemble de départ n'étant composé que d'un seul état : l'état initial de l'automate. L'exécution de l'algorithme se termine dès que la chaîne d'entrée est épuisée ou lorsque l'ensemble des états atteignables est vide. Lorsque l'état final est atteint la chaîne de caractères est acceptée par l'automate et, par conséquent, correspond à l'expression régulière, c.à.d au motif.

L'algorithme de Thompson est un procédé efficace pour obtenir un automate reconnaissant le langage dénoté par une expression rationnelle. Cet algorithme existe avec différentes variantes [KM08], notamment sur la manière de faire la concaténation. En effet, l'article d'origine de Thompson s'exprime non pas en terme d'automate mais en terme de circuit [Ber05]. La version que nous présentons ci-dessous permet de caractériser entièrement l'automate et ainsi, de faire facilement une bijection entre un automate de Thompson et une expression régulière.

3.2.1 Construction de l'automate de Thompson

La construction d'un automate de Thompson pour les mots s'effectue de manière inductive sur la forme de l'expression. Tel que mentionné précédemment, l'automate est composé d'un ensemble de fragments. Ces fragments sont assemblés suivant les opérations de base des expressions régulières. Les

figures 3.2 à 3.7 montrent comment les fragments sont assemblés en fonction des opérations appliquées.

Nous construisons sans difficulté des automates pour les langages $\{\varepsilon\}$ et les singletons. Puis, supposons des automates de Thompson Th_E et Th_B construits pour les expressions régulières E et F, nous construisons de façon inductive les automates de Thompson pour l'union, la concaténation et l'itération comme le montre les figures suivantes.

Soit E et F deux expressions régulières :

1. La figure 3.2 schématise l'automate de Thompson pour le langage vide. L'état i (respectivement t) représente l'état initial (terminal) de l'automate.
2. La figure 3.3 schématise l'automate de Thompson pour le mot vide. L'état initial i est relié à l'état final t par une transition vide.
3. L'automate de Thompson reconnaissant un caractère (figure 3.4) est construit en reliant l'état initial à l'état final par la lecture du caractère a.
4. L'automate de Thompson de l'expression $E+F$ est construit en créant deux nouveaux états : un état initial lié par une ε-transition aux deux états initiaux de E et F, et un état final auquel est lié les états finaux de E et F également par une ε-transition (figure 3.5).
5. La construction de l'automate de Thompson pour la concaténation $E.F$ est obtenu en confondant l'état final de E avec l'état initial de F. L'état initial de E devient l'état initial de l'automate $E.F$ et l'état final de F devient son état final (figure 3.6).
6. La figure 3.7 représente la construction de l'automate de Thompson pour l'opération de l'étoile. Un état initial et un état final sont ajoutés ainsi que quatre ε-transitions :
   - une transition reliant l'état initial de E^* à celui de E,
   - une autre transition reliant l'état final de E à celui de E^*,
   - une transition de l'état initial de E à son état final afin de pouvoir générer le mot vide,
   - et une dernière transition reliant l'état final de E à son état initial.

FIGURE 3.2 – Automate de Thompson du langage vide.

FIGURE 3.3 – Automate de Thompson du mot vide.

FIGURE 3.4 – Automate de Thompson d'un caractère a.

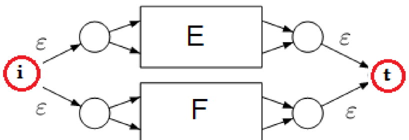


FIGURE 3.5 – Automate de Thompson de l'union $E + F$.

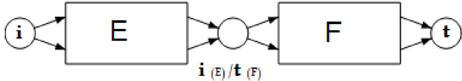


FIGURE 3.6 – Automate de Thompson de la concaténation $E.F$.

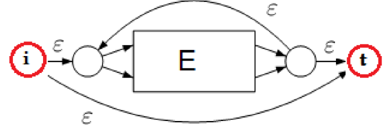


FIGURE 3.7 – Automate de Thompson de l'étoile E^*.

3.2.2 Exemple d'un automate de Thompson de mots

La figure 3.8 montre l'automate de Thompson construit pour l'expression régulière de mots $(a+b)^*b(\varepsilon+a)(a+b)^*$. L'automate a 21 états et 27 flèches. Notons que la plupart des transitions (flèches) sont étiquetées par le mot vide.

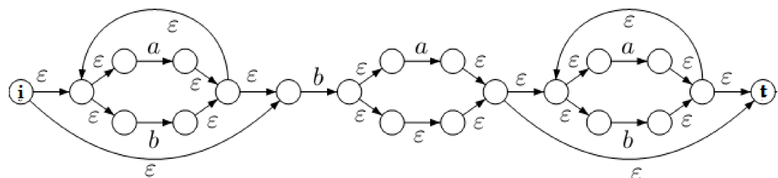


FIGURE 3.8 – Exemple d'un automate de Thompson pour les mots.

3.2.3 Discussion de l'algorithme de Thompson pour les mots

L'automate de Thompson est un automate particulier que l'on dit normalisé, il respecte les propriétés suivantes [Ber05] :

1. il existe un seul état initial, et un seul état final et ces deux états sont distincts ;
2. aucune flèche ne pointe sur l'état initial, aucune flèche ne sort de l'état final ;
3. tout état est soit l'origine d'exactement une flèche étiquetée par une lettre, soit l'origine d'au plus deux flèches étiquetées par le mot vide (ε).

Notons que le nombre de flèches d'un automate normalisée est au plus le double du nombre de ses états. Cette normalisation nous permet de conclure que l'algorithme de Thompson est d'une concision appréciable.

L'implémentation présentée ci-dessus n'est évidemment pas optimisée, mais demeure meilleure, de par la nature même de l'algorithme utilisé, à toutes les implémentations récursives exploitant du retour sur trace. Plus précisément, la complexité de l'algorithme de Thompson est bornée par $O(mn)$, où m est la taille de l'expression régulière et n la taille du texte analysé. En comparaison, les implémentations récursives exploitant du retour sur trace affichent une complexité $O(2^n)$, où n est la taille du texte. Évidemment, de tels comportements ne surviennent que pour le pire des cas.

3.3 Notre généralisation de la technique de Thompson pour les arbres

Par analogie à la construction de Thompson pour les mots, nous allons construire un automate de Thompson ascendant pour les arbres à partir d'une expression régulière d'arbre afin de l'utiliser pour une recherche de motifs d'arbre. Pour ce faire, nous avons besoin de définir une forme générale d'un automate d'arbre ayant des caractéristiques similaires à celles de Thompson pour mots.

3.3.1 Forme générale de l'automate de Thompson d'arbre

L'idée de base consiste à ramener l'automate d'arbre à un automate d'une forme particulière comme l'indique la figure 3.9. En effet, selon cette forme, l'automate de Thompson d'arbre pour une expression d'arbre E possède :

1. un seul état final q_f,
2. un ensemble d'états initiaux représentant les feuilles (le rectangle Σ_0)
3. et un ensemble de transitions quelconques reliant l'état final à Σ_0.

L'ensemble Σ_0 ne doit contenir qu'un et un seul état pour chaque feuille. Cela se fait en rajoutant des ε-transitions entre les états concernés et les états de Σ_0.

Le but essentiel de cette représentation est de faciliter les opérations de concaténation et de fermeture. En effet, ces opérations dans le cas des arbres sont plus compliquées que celles des mots car elles peuvent être effectuées dans plusieurs nœuds feuilles. Pour remédier à ce problème nous avons créé pour chaque feuille un nouvel état initial avec des ε-transitions vers tout les états initiaux de cette même feuille.

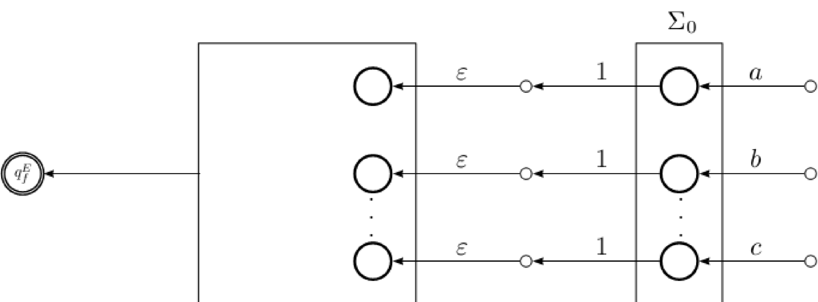


FIGURE 3.9 – Forme générale d'un automate de Thompson pour les arbres.

3.3.2 Construction de l'automate

Notre construction de l'automate de Thompson pour les arbres est exprimée de la même manière que celle des mots. Nous allons présenter les constructions de base concernant l'arbre feuille et la fonction d'arité ainsi que les opérations effectuées sur les arbres : l'union, la concaténation et la fermeture. Nous allons donner pour chaque construction l'ensemble des états et celui des fonctions de transition.

L'automate élémentaire (l'arbre feuille)

L'automate est composé d'un seul état : l'état initial qui est lui même l'état final. Spécialement pour l'arbre feuille, les ε-transitions sont omises.

$Q^E = \{q_f^E\} = \{q_a^E\}$
$\Delta^E = \{a \to q_a^E\}$

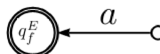


FIGURE 3.10 – Automate de Thompson de l'arbre feuille.

L'automate de la fonction d'arité $E = f(E_1, E_2, ..., E_n)$

Étant donnés les automates de Thompson des expressions régulières d'arbre E_i, $Th_{E_i} = (\Delta^{E_i}, Q^{E_i})$ pour $i = 1$ à n. L'automate de Thompson correspondant à l'expression de la fonction d'arité $E = f(E_1, E_2, ..., E_n)$ est schématisé dans la figure 3.11. En effet, l'automate est construit en rajoutant un nouvel état final q_f^E et en fusionnant les ensembles des états représentants les feuilles Σ_0 afin de garder la structure de base de l'automate de Thompson. Ceci se fait par :

- l'ajout de k états initiaux q_a^E, tel que $k = |\Sigma_0|$, et $a \in \Sigma_0$,
- la suppression des transitions $a \to q_a^{E_i}$, tel que $a \in \Sigma_0$ et $i : 1...n$,
- l'ajout de ε-transitions entre q_a^E et $q_a^{E_i}$,
- l'ajout de la transition $a \to q_a^E$ pour tout $a \in \Sigma_0$.

Cette manière de fusionner les ensembles Σ_0 est aussi utilisée pour les opérations d'union et de concaténation.

Donc :

$$\begin{array}{rcl} Q^E & = & (\bigcup\limits_{i=1...n})Q^{E_i} \cup \{q_f^E\} \cup \{q_a^E \backslash a \in \Sigma_0\} \\ \Delta^E & = & \{\Delta^{E_i}\} \setminus \{a \to q_a^{E_i}, a \in \Sigma_0\} \\ & & \cup\{f(q_f^{E_1}, q_f^{E_2}, ..., q_f^{E_n}) \to q_f^E\} \\ & & \cup\{a \to q_a^E \backslash a \in \Sigma_0\} \\ & & \cup\{\varepsilon(q_a^{E_i}) \to q_a^E \backslash a \in \Sigma_0, i = 1 \ldots n\} \end{array}$$

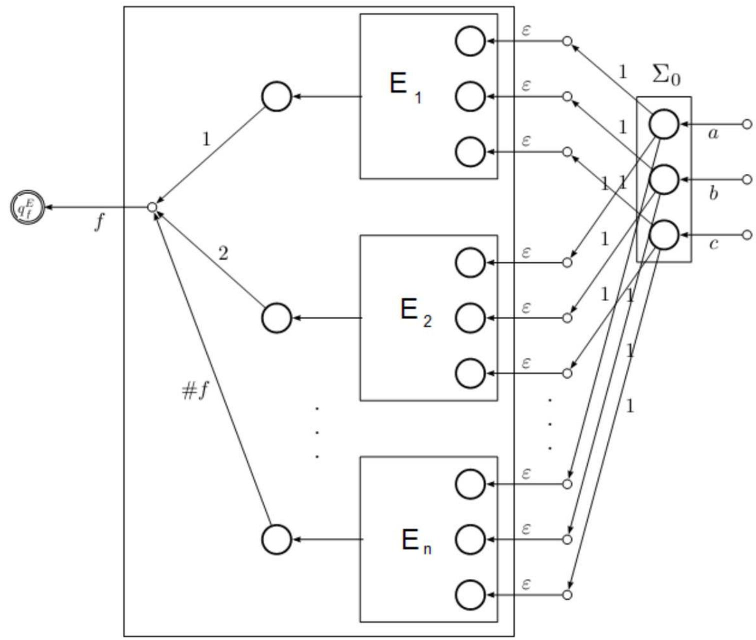


FIGURE 3.11 – Automate de Thompson de la fonction d'arité.

L'automate de l'union $E = F + G$

L'union de deux automates de Thompson Th_F et Th_G correspondants aux expressions F et G est illustré par la figure 3.12. En effet, on rajoute un nouvel état final reliant les états finaux des deux automates q_f^F et q_f^G avec des ε-transitions et on fusionne les ensembles Σ_0.

Formellement :

$$
\begin{array}{lcl}
Q^E & = & Q^F \cup Q^G \cup \{q_f^E\} \cup \{q_a^E \backslash a \in \Sigma_0\} \\
\Delta^E & = & \{\Delta^F\} \setminus \{a \to q_a^F, a \in \Sigma_0\} \cup \{\Delta^G\} \setminus \{a \to q_a^G, a \in \Sigma_0\} \\
 & & \cup\{\varepsilon(q_a^E) \to q_a^G \backslash a \in \Sigma_0\} \\
 & & \cup\{\varepsilon(q_a^E) \to q_a^F \backslash a \in \Sigma_0\} \\
 & & \cup\{\varepsilon(q_f^F) \to q_f^E \backslash\} \\
 & & \cup\{\varepsilon(q_f^G) \to q_f^E \backslash\} \\
 & & \cup\{a \to q_a^E \backslash a \in \Sigma_0\}
\end{array}
$$

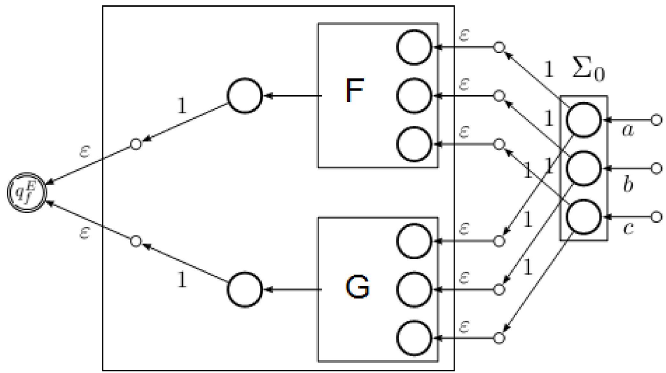


Figure 3.12 – Automate de Thompson de l'union de deux automates.

L'automate de la concaténation $E = \mathbf{F}._c\mathbf{G}$

La construction de l'automate de concaténation est un peu différente vu que l'opération de concaténation dans les arbres n'a pas la même sémantique que pour les mots.

Soient deux expressions régulières d'arbre F et G. Soient Th_F et Th_G leur automate de Thompson correspondants. La figure 3.13 illustre l'opération de concaténation de ces deux automates.

Nous supprimons de l'ensemble Σ_0 l'état représentant le symbole de concaténation c, et nous ajoutons une ε-transition de cet état vers l'état initial de l'automate à concaténer G.

Alors :

$$\begin{array}{lcl} Q^E & = & Q^F \cup Q^G \cup \{q_f^E\} \cup \{q_a^E \backslash a \in \Sigma_0\} \\ \Delta^E & = & \{\Delta^F\} \setminus \{a \to q_a^F, a \in \Sigma_0\} \cup \{\Delta^G\} \setminus \{a \to q_a^G, a \in \Sigma_0\} \\ & & \cup \{a \to q_a^E \backslash a \in \Sigma_0\} \\ & & \cup \{\varepsilon(q_a^E) \to q_a^G \backslash a \in \Sigma_0\} \\ & & \cup \{\varepsilon(q_a^E) \to q_a^F \backslash a \in (\Sigma_0 \setminus \{c\})\} \\ & & \cup \{\varepsilon(q_f^G) \to q_f^c\} \end{array}$$

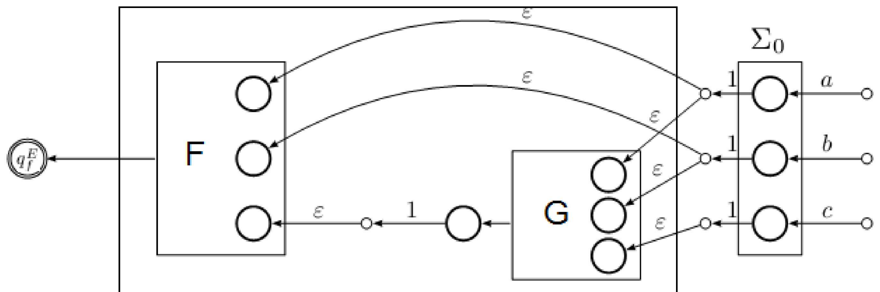


FIGURE 3.13 – Automate de Thompson de la concaténation de deux automates.

L'automate de la fermeture $E = F^{*c}$

Pour la construction de l'automate de Thompson de l'opération de fermeture nous augmentons l'automate de l'expression régulière de base F de trois ε-transitions supplémentaires et d'un nouvel état initial :

- une transition de l'état final de F vers l'état représentant le symbole de concaténation dans F,
- une transition de l'état représentant le symbole de concaténation dans F vers l'état final de E,
- et une autre transition de l'état final de F vers celui de E.

Formellement :

$$
\begin{array}{lcl}
Q^E & = & Q^F \cup \{q_f^E\} \cup \{q_a^E \backslash a \in \Sigma_0\} \\
\Delta^E & = & \{\Delta^F\} \setminus \{a \to q_a^F, a \in \Sigma_0\} \\
& & \cup\{a \to q_a^E \backslash a \in \Sigma_0\} \\
& & \cup\{\varepsilon(q_c^E) \to q_f^E\} \\
& & \cup\{\varepsilon(q_c^F) \to q_f^F\} \\
& & \cup\{\varepsilon(q_f^F) \to q_f^E\}
\end{array}
$$

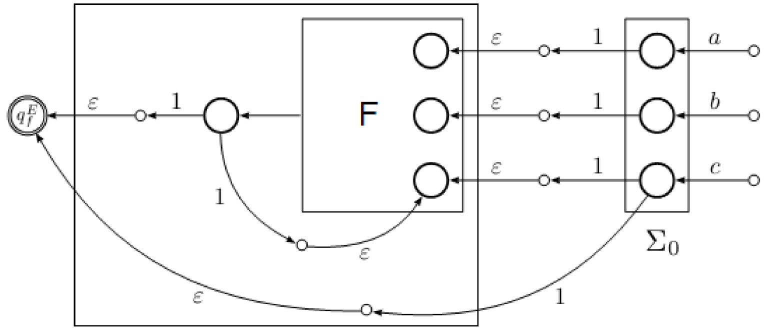


FIGURE 3.14 – Automate de Thompson de la fermeture d'un automate.

3.4 Preuves d'équivalences entre les constructions et les ER

La construction de l'automate de Thompson pour une expression régulière d'arbre E étant faite, il nous reste à prouver qu'il y a bien une équivalence entre le langage régulier correspondant à E et celui reconnu par l'automate construit.

Commençons par définir la fonction ε-closure qui sert à éliminer toutes les transitions de type $\varepsilon(X) \to Y$

Définition 14. *Soit t un arbre, nous définissons la fonction d'élimination des ε-transitions de l'arbre t, ε-closure(t) comme suit :*

$\varepsilon\text{-}closure(a) = a$ *pour tout* $a \in \Sigma_0$
$\varepsilon\text{-}closure(\varepsilon(t)) = t$
$\varepsilon\text{-}closure(f(t_1, t_2, ..., t_n)) = f(\varepsilon\text{-}closure(t_1),\ \varepsilon\text{-}closure(t_2),\ ...,\ \varepsilon\text{-}closure(t_n))$

À partir de cette définition nous pouvons déduire la propriété suivante :

Propriété 1. *Pour touts arbres t_1, t_2, nous avons*
$\varepsilon\text{-}closure(t_1 ._c t_2) = [\varepsilon\text{-}closure(t_1)]._c\ \varepsilon\text{-}closure(t_2)$.

Cette propriété peut être aussi étendue aux ensembles par la propriété suivante :

Propriété 2. $\varepsilon\text{-}closure(t._c\{t_1,\ t_2,...\ t_k\ \}) =$
$[\varepsilon\text{-}closure(t)]._c\{\varepsilon\text{-}closure(t_1), \varepsilon\text{-}closure(t_2), ...\varepsilon\text{-}closure(t_k)\}$

Théorème 1. *Soit E une expression régulière d'arbre, Th_E l'automate de Thompson généré pour E. $[\![E]\!]$ est le langage reconnu par l'expression E et $\mathcal{L}(Th_E)$ est le langage reconnu par Th_E.*

Alors, nous avons $\varepsilon\text{-}closure\mathcal{L}(Th_E) \equiv [\![E]\!]$.

Pour prouver ce théorème nous allons prouver les deux lemmes suivants.

Lemme 1. *Pour tout* $t \in [\![E]\!]$, *il existe un arbre* $t' \in \mathcal{L}(Th_E)$ *tel que* ε-$closure(t') = t$.

Lemme 2. *Si* $t \in \mathcal{L}(Th_E)$, *alors* ε-$closure(t) \in [\![E]\!]$.

3.4.1 Preuve du Lemme 1

La preuve est accomplie par induction sur la structure de l'automate construit.

L'arbre feuille

$E = a$. Ce cas est évident car : $\mathcal{L}(Th_E) = \{a\}$, $[\![E]\!] = \{a\}$, donc $\mathcal{L}(Th_E) \equiv [\![E]\!]$, alors pour tout $t \in [\![E]\!]$, il existe bien $t' \in \mathcal{L}(Th_E)$ tel que
ε-$closure(t') = t$. □

La fonction d'arité

$E = f(E_1, E_2, ..., E_n)$. Où n est l'arité de la fonction f.
Soit $t \in [\![E]\!]$, alors $t = f(e_1, e_2, ..., e_n)$, tel que $e_1 \in [\![E_1]\!]$, $e_2 \in [\![E_2]\!]$, ..., $e_n \in [\![E_n]\!]$.

Selon l'hypothèse d'induction, il existe $t'_i \in \mathcal{L}(Th_{E_i})$ tel que
ε-$closure(t'_i) = t_i$ avec $i = 1...n$.
Construisons un terme $t' = f(t'_1, t'_2, ..., t'_n)$ et en remplaçant chaque symbole x d'arité 0 par $\varepsilon(x)$. C'est à dire $t' = (\ f(t'_1, t'_2, ..., t'_n)\ )._a\varepsilon(a)..._c\varepsilon(c)....$
Montrons maintenant que $t' \in \mathcal{L}(Th_E)$ et que ε-$closure(t') = t$.

Selon la construction de l'automate de Thompson pour la fonction d'arité, la transition $a \rightarrow q_a^{E_i}$ dans le chemin de reconnaissance de t'_i dans Th_{E_i} est remplacée par $a \rightarrow q_a^E$ et $\varepsilon(q_a^E) \rightarrow q_a^{E_i}$ pour tout $i = 1...n$ et $a \in \Sigma_0$.
Comme $\Delta^{E_i} \subset \Delta^E$ et $t'_i \in \mathcal{L}(Th_{E_i})$, c'est à dire : $\sigma^*_{E_i}(t'_i) = q_f^{E_i}$.

Donc : $\sigma^*_E(t'_i) = q_f^{E_i}$, $i = 1...n$.
De plus nous avons $f(q_f^{E_1}, q_f^{E_2}, ..., q_f^{E_n}) \rightarrow q_f^E \in \Delta^E$, alors : $\sigma^*_E(t') = q_f^E$.
Par conséquent t' est reconnu par $\mathcal{L}(Th_E)$.

D'autre part nous avons :

$$
\begin{aligned}
\varepsilon\text{-}closure(t') &= \varepsilon\text{-}closure(f(t'_1, t'_2, ..., t'_n)_{.a}\varepsilon(a)..._{.c}\varepsilon(c)...) \\
&= \varepsilon\text{-}closure(f(t'_1, t'_2, ..., t'_n))_{.a}(\varepsilon\text{-}closure(\varepsilon(a)))..._{.c}\varepsilon(\varepsilon\text{-}closure(\varepsilon(c)))... \\
&= \varepsilon\text{-}closure(f(t'_1, t'_2, ..., t'_n))_{.a}a, ...,_{.c}c. \\
&= \varepsilon\text{-}closure(f(t'_1, t'_2, ..., t'_n)) \\
&= f(\varepsilon\text{-}closure(t'_1), \varepsilon\text{-}closure(t'_2), ..., \varepsilon\text{-}closure(t'_n)) \\
&= f(t_1, t_2, ..., t_n) = t.
\end{aligned}
$$

Donc, il existe $t' \in \mathcal{L}(Th_E)$ tel que $\varepsilon\text{-}closure(t') = t$. □

L'union

$E = F + G$. Nous avons $[\![E]\!] = [\![F]\!] \cup [\![G]\!]$.
Soit $t \in [\![E]\!]$, sans perte de généralité, supposons que $t \in [\![F]\!]$.

Selon l'hypothèse d'induction, il existe $t'_F \in \mathcal{L}(Th_F) : \varepsilon\text{-}closure(t'_F) = t$. Construisons un terme $t' = \varepsilon(t'_F)$, ensuite nous remplaçons chaque symbole x d'arité 0 par $\varepsilon(x)$, c'est à dire : $t' = (t'_F)_{.a}\varepsilon(a)..._{.c}\varepsilon(c)....$
Montrons maintenant que $t' \in \mathcal{L}(Th_E)$ et que $\varepsilon\text{-}closure(t')= t$

D'après la construction de Thompson pour l'union, nous remplaçons chaque transition $a \to q^F_a$ par $a \to q^E_a$ et $\varepsilon(q^E_a) \to q^F_a$ tel que $a \in \Sigma_0$.
Comme $\Delta^F \subset \Delta^E$ et $t'_i \in \mathcal{L}(Th_F)$, c'est à dire : $\sigma^*_F(t'_F) = q^F_f$.
Donc : $\sigma^*_E(t'_F) = q^F_f$.

De plus nous avons $\varepsilon(q^F_f) \to q^E_f \in \Delta^E$, alors : $\sigma^*_E(t'_F) = q^E_f$, ce qui veut dire que t' est reconnu par $\mathcal{L}(Th_E)$. D'autre part nous avons :

$$
\begin{aligned}
\varepsilon\text{-}closure(t') &= \varepsilon\text{-}closure(\varepsilon(t'_F)_{.a}\varepsilon(a)..._{.c}\varepsilon(c)...) \\
&= \varepsilon\text{-}closure(t'_F)_{.a}a, ...,_{.c}c. \\
&= \varepsilon\text{-}closure(t'_F) = t.
\end{aligned}
$$

Donc il existe $t' \in \mathcal{L}(Th_E)$ tel que $\varepsilon\text{-}closure(t') = t$. □

La concaténation

$E = F._cG$. Soit $t \in [\![E]\!]$, $t \in [\![F]\!]._c[\![G]\!]$, c'est à dire $t \in \{(t_F)_{.c}\{t_1, ..., t_k\}\}$, tel que $t_i \in [\![G]\!]$, $i = 1...k$.

Selon l'hypothèse d'induction, il existe $t'_F, t_1, ..., t_k$ avec $t'_F \in \mathcal{L}(Th_F)$ et $t'_i \in \mathcal{L}(Th_G)$, $i = 1...k$, tel que $\varepsilon\text{-}closure(t'_F)= t_F$ et $\varepsilon\text{-}closure(t'_k)= t_k$.

Construisons les deux termes t''_F et t''_j en remplaçant chaque symbole x d'arité 0 par $\varepsilon(x)$, c'est à dire :
$t''_F = (t'_F)_{.a}\varepsilon(a)..._{.c}\varepsilon(c)....$
$t''_j = (t'_j)_{.a}\varepsilon(a)..._{.c}\varepsilon(c)....$
$t' \in \{\varepsilon(t''_F)_{.c}\{t''_1, t''_2, ..., t''_k\}\}$.

Montrons que $t' \in \mathcal{L}(Th_E)$.

D'après la construction de l'automate de Thompson de la concaténation nous avons remplacé la transition $a \to q_a^G$ dans Th_G par $a \to q_a^E$ et $\varepsilon(q_a^E) \to q_a^G$ dans Th_E avec $a \in \Sigma_0$.
Et comme $\Delta^G \subset \Delta^E$ et $\sigma^*_G(t'_j) = q_f^G$, alors $\sigma^*_E(t''_F) = q_f^G$.

D'après la même construction, nous avons remplacé la transition $a \to q_a^F$ par $a \to q_a^E$ et $\varepsilon(q_a^E) \to q_a^F$ pour $a \neq c$, c étant le symbole de la concaténation.
Si $a = c$, cette transition $(c \to q_c^F)$ est remplacée par $\varepsilon(q_f^G) \to q_c^F$.

Puisque t'_F est reconnu par Th_F selon l'hypothèse d'induction, alors : $\sigma^*_F(t'_F) = q_f^F$.
Comme nous avons $\varepsilon(q_a^E) \to q_a^F \in \Delta^E$ pour $a \neq c$ et $\varepsilon(q_f^G) \to q_c^F \in \Delta^E$ dans le cas contraire (si $a = c$) et $(\Delta^F \setminus \{c \to q_c^F\}) \subset \Delta^E$, alors nous avons $\sigma^*_E(t') = q_f^F$.

Par conséquent $\sigma^*_E(\varepsilon(t')) = q_f^E$ car $\varepsilon(q_f^F) \to q_f^E \in \Delta^E$. Donc t' est reconnu par $\mathcal{L}(Th_E)$.

D'autre part nous avons :

$$\begin{array}{lcl} \varepsilon\text{-}closure(\varepsilon(t')) & = & \varepsilon\text{-}closure(t'). \\ \varepsilon\text{-}closure(t') & \in & \{\varepsilon\text{-}closure(\varepsilon(t''_F)._c(t''_1, ..., t''_k))\} \\ & \in & \{\varepsilon\text{-}closure(t''_F)_{.c}\varepsilon\text{-}closure(t''_1),\ ...,\ \varepsilon\text{-}closure(t''_k)\} \end{array}$$

Remplaçons les termes t''_j et t''_F respectivement par $\varepsilon\text{-}closure(t'_j)_{.a}\varepsilon(a), ...,_{.c}\varepsilon(c)$ et $\varepsilon\text{-}closure(t'_F)_{.a}\varepsilon(a), ...,_{.c}\varepsilon(c)$. .
On aura : $\varepsilon\text{-}closure(t') \in \varepsilon\text{-}closure(t'_F)_{.c}\{\varepsilon\text{-}closure(t'_1),\ ...,\ \varepsilon\text{-}closure(t'_k)\}$.

Donc $t \in \{(t_F)_{.c}\{t_1, ..., t_k\}\}$. □

La fermeture

$E = F^{*c}$. Soit $t \in [\![F^{*c}]\!]$:

$$t \in \begin{cases} [\![F^{0,c}]\!] & \text{si } n = 0, \\ [\![F^{n,c}]\!], n \in \mathbb{N}^* & \text{sinon.} \end{cases}$$

Montrons que pour tout $t \in F^{*c}$, il existe $t' \in \mathcal{L}(Th_E)$, tel que $\varepsilon\text{-}closure(t')= t$.

Pour le cas $n = 0$, $t \in [\![F^{0,c}]\!]$, $t^0 = c$, ce cas est évident car nous avons : $\{c \to q_c^E, \varepsilon(q_c^E) \to q_f^E\} \in \Delta^E$.
Alors $t'^0 = \varepsilon(t^0)$.
Donc t'^0 est reconnu par l'automate Th_E et $\varepsilon\text{-}closure(t'^0) = t^0$.

Pour $n = 1$, $t \in [\![F^{1,c}]\!]$, $t^1 = \text{c}._c[\![F]\!]$, nous démontrons qu'il existe $t'^1 \in \mathcal{L}(Th_E)$ et $\varepsilon\text{-}closure(t'^1)$= t^1.
$t'^1 \in \{\text{c}._c\{t_i'^1\}/t_i'^1 \in [\![F]\!]\}$ alors $t'^1 \in \{\text{c}._c\{t_1'^1, ..., t_j'^1\}\}$.

Selon l'hypothèse d'induction $\{c, t_i'^1\} \in \mathcal{L}(Th_F)$, c'est à dire $\sigma_F^*(t_i'^1) = q_f^F$. Et d'après la construction de l'automate de Thompson pour l'opération de fermeture nous avons $\varepsilon(q_f^F) \to q_f^E \in \Delta^E$, donc $\sigma_E^*(t_i'^1) = q_f^E$. Par conséquent, $t'^1 \in \mathcal{L}(Th_E)$.

D'autre part, nous avons :

$$\begin{aligned}
\varepsilon\text{-}closure(t_i'^1) &\in \{\varepsilon\text{-}closure(\text{c}._c\{t_1'^1, ..., t_j'^1\})\} \\
&\in \{\varepsilon\text{-}closure(c)._c\{\varepsilon\text{-}closure(t_1'^1, ..., t_j'^1)\}\} \\
&\in \{c._c\{t_1^1, ..., t_j^1\}\}
\end{aligned}$$

Pour le cas $(1 < k \leq n)$, l'opération de fermeture se ramène à une concaténation. $t^k = t^{k-1}._c[\![F]\!]$.
Donc il existe $t'^k \in \mathcal{L}(Th_E)$ tel que $\varepsilon\text{-}closure(t'^k) = t^k$. $\square$

3.4.2 Preuve du Lemme 2

Rappelons que la preuve de ce lemme est accomplie par induction sur la structure de l'automate construit.

L'arbre feuille

$E = a$. Ce cas est évident car $t = a$ et $\varepsilon\text{-}closure(t) = [\![E]\!]$. $\square$

La fonction d'arité

$E = f(E_1, E_2, ..., E_n)$. Nous avons $t \in \mathcal{L}(Th_E)$ veut dire que $t = f(t_1, t_2, ..., t_n)$.
Selon l'hypothèse d'induction, il existe $t_1, t_2, ..., t_n$ tel que :
$t_1 \in \mathcal{L}(Th_{E_1})$ et $\varepsilon\text{-}closure(t_1) \in [\![E_1]\!]$.
$t_2 \in \mathcal{L}(Th_{E_2})$ et $\varepsilon\text{-}closure(t_2) \in [\![E_2]\!]$.
$\vdots$
$t_n \in \mathcal{L}(Th_{E_n})$ et $\varepsilon\text{-}closure(t_n) \in [\![E_n]\!]$.

Nous avons :
$f(\varepsilon\text{-}closure(t_1), \varepsilon\text{-}closure(t_2), ..., \varepsilon\text{-}closure(t_n)) \in [\![f(E_1, E_2, ..., E_n)]\!]$.
Selon la définition 14 de la fonction $\varepsilon\text{-}closure(t)$, nous avons :

$f(\varepsilon\text{-}closure(t_1), \varepsilon\text{-}closure(t_2), ..., \varepsilon\text{-}closure(t_n)) = \varepsilon\text{-}closure(f(t_1, t_1, ..., t_n))$.
Donc, $\varepsilon\text{-}closure(f(t_1, t_1, ..., t_n)) \in [\![f(E_1, E_2, ..., E_n)]\!]$.
Et par conséquent, $\varepsilon\text{-}closure(f(t_1, t_1, ..., t_n)) \in [\![E]\!]$.

D'autre part, nous avons $t = f(t_1, t_2, ..., t_n)$, alors :
$\varepsilon\text{-}closure(t) \in [\![E]\!]$. □

L'union

$E = F + G$. Nous avons $t \in \mathcal{L}(Th_E)$ veut dire que $t = t_f$ ou $t = t_g$, où $t_f \in \mathcal{L}(Th_F)$ et $t_g \in \mathcal{L}(Th_G)$.

Selon l'hypothèse d'induction :
$t_f \in \mathcal{L}(Th_F)$ et $\varepsilon\text{-}closure(t_f) \in [\![F]\!]$.
$t_g \in \mathcal{L}(Th_G)$ et $\varepsilon\text{-}closure(t_g) \in [\![G]\!]$.
Sans perte de généralité, supposons que $t = t_f$.

Nous avons $\varepsilon\text{-}closure(t_f) \in [\![F]\!]$, alors $\varepsilon\text{-}closure(t_f) \in [\![F + G]\!]$.

Donc, $\varepsilon\text{-}closure(t) \in [\![F + G]\!]$.
Et comme $E = F + G$, alors $\varepsilon\text{-}closure(t) \in [\![E]\!]$. □

La concaténation

$E = F._cG$. Si $t \in \mathcal{L}(Th_E)$ alors t s'écrit sous la forme $t = t_{f.c}t_g$.

Selon l'hypothèse d'induction :
$t_f \in \mathcal{L}(Th_F)$ et $\varepsilon\text{-}closure(t_f) \in [\![F]\!]$.
$t_g \in \mathcal{L}(Th_G)$ et $\varepsilon\text{-}closure(t_g) \in [\![G]\!]$.

Nous avons $\varepsilon\text{-}closure(t_f)_{.c}\ \varepsilon\text{-}closure(t_g) \in [\![F._cG]\!]$.
Et selon la propriété 2, nous avons
$\varepsilon\text{-}closure(t_f._ct_g) = \varepsilon\text{-}closure(t_f)_{.c}\ \varepsilon\text{-}closure(t_g)$.
Donc, $\varepsilon\text{-}closure(t) \in [\![F._cG]\!]$.

Et par conséquent, $\varepsilon\text{-}closure(t) \in [\![E]\!]$. □

La fermeture

$E = F^{*c}$. Si $t \in \mathcal{L}(Th_E)$, alors t est de la forme : $t = {t_f}^{n,c}$, $t_f \in [\![F]\!]$ et

$${t_f}^{n,c} = \begin{cases} {t_f}^{0,c} & = & c & \text{si } n = 0, \\ {t_f}^{1,c} & = & ({t_f}^{0,c})_{.c}t_f & \text{si } n = 1, \\ {t_f}^{i,c} & = & ({t_f}^{i-1,c})_{.c}t_f & \text{si } n > 2. \end{cases}$$

D'après l'hypothèse d'induction, nous avons $t_f \in \mathcal{L}(Th_F)$ et $\varepsilon\text{-}closure(t_f) \in [\![F]\!]$.

Pour le cas $t = {t_f}^{0,c} = c$, nous avons $\varepsilon\text{-}closure(t_f) = \varepsilon\text{-}closure(c) = c$.

Nous avons aussi $c \in [\![F]\!]$, donc $c \in [\![F^{0,c}]\!]$.

Par conséquent, $\varepsilon\text{-}closure(t) \in [\![E]\!]$.

Pour $t = {t_f}^{1,c} = ({t_f}^{0,c})_{.c}t_f = t_f$, nous avons $\varepsilon\text{-}closure(t) = \varepsilon\text{-}closure(t_f)$. $\varepsilon\text{-}closure(t_f) \in [\![F]\!]$ selon l'hypothèse d'induction alors, $\varepsilon\text{-}closure(t_f) \in [\![F^{1,c}]\!]$, car $[\![F^{1,c}]\!] = [\![F^{0,c}]\!] \cup [\![F^{0,c}_{.c}F]\!]$ selon la propriété de fermeture des langages réguliers.

Et par conséquent, $\varepsilon\text{-}closure(t) \in [\![E]\!]$.

Pour ${t_f}^{n,c}$ avec $n > 1$, la fermeture est un cas particulier de l'opération de concaténation, et elle a été démontrée précédemment.

Donc, pour $t = {t_f}^{n,c}$ et $E = F^{*c}$, $\varepsilon\text{-}closure(t) \in [\![E]\!]$. □

3.5 L'algorithme de recherche de motifs proposé

Par analogie à l'automate de Thompson pour la recherche de motif de mots, nous allons utiliser l'automate de Thompson construit ci-dessus pour la recherche de motifs d'arbre. Nous pouvons utiliser les deux stratégies suivantes :

1. Soit construire l'automate d'arbre descendant de l'expression régulière $E_{.v}\Sigma^*$, où E est l'expression régulière de l'arbre motif et $v \in \Sigma_0$ est une feuille variable.
2. Soit construire l'automate d'arbre ascendant pour seulement l'expression régulière E, nous n'avons pas besoin de l'augmenter pour reconnaître les sous-arbres qui ne sont pas des motifs, car si le motif existe il est sûrement un sous arbre de l'arbre objet. Et vu que l'on commence la reconnaissance par les feuilles, il suffit que l'on rencontre l'état final.

3.5.1 Construction de l'automate descendant de RMA

Afin de faire la recherche de motifs nous avons besoin de construire l'automate de Thompson descendant de l'expression régulière $E_{.v}\Sigma^*$. Le fragment d'automate qui représente Σ^* permet de générer tous les nœuds de

l'arbre qui ne font pas partie du motif, et la concaténation à travers le symbole v permet de reconnaitre les motifs. Cet automate est construit comme suit :

1. pour tout $a \in \Sigma_0$, ajouter un état q_a et une transition feuille $a \to q_a$;
2. Ajout d'un état q_Σ ;
3. Ajout de $\varepsilon-$ transition $\varepsilon(q_a) \to q_\Sigma$ entre q_Σ et q_a, pour tout $a \in \Sigma_0$;
4. Pour tout $x \in (\Sigma \setminus \Sigma_0)$ ajouter une transition réflexive de q_Σ étiquetée par x.

Formellement :

$$\begin{array}{lll} Q^{E.v\Sigma^*} = & Q^E & \cup\{q_\Sigma\} \\ \Delta^{E.v\Sigma^*} = & \Delta^E & \cup\{\varepsilon(q_a^E) \to q_\Sigma, a \in \Sigma_0\} \\ & & \cup\{x\underbrace{(q_\Sigma, \ldots, q_\Sigma)}_{r(x) fois} \to q_\Sigma, x \in (\Sigma \setminus \Sigma_0)\} \\ q_T^{E.v\Sigma^*} = & q_T^E & \\ Q_{\Sigma_0}^{E.v\Sigma^*} = & Q_{\Sigma_0}^E & \end{array}$$

Exemple

La figure 3.15 schématise l'automate descendant construit pour l'expression régulière $E_{.v}\Sigma^*$ où $E = a(b(c), d)$.

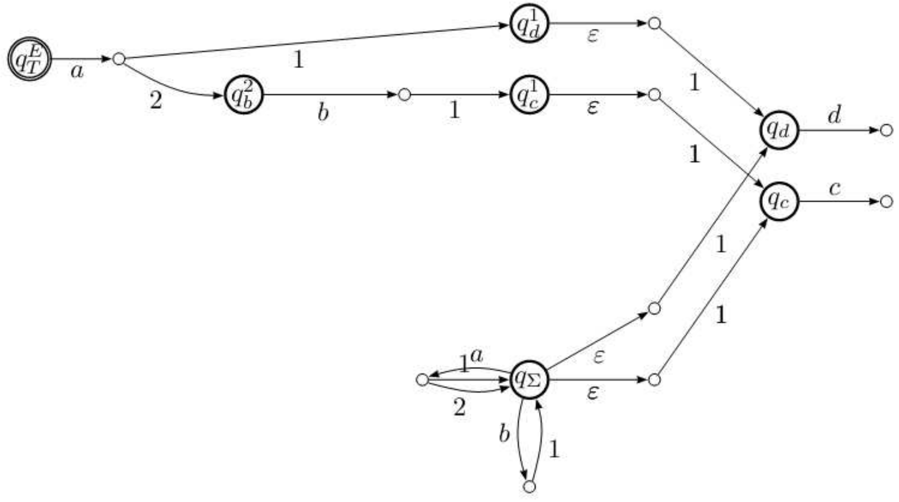


FIGURE 3.15 – Automate descendant pour la RMA de l'expression E.

3.5.2 Construction de l'automate ascendant de RM

Afin d'effectuer la recherche de motif avec la technique de Thompson en choisissant une reconnaissance ascendante, nous avons besoin de construire

seulement l'automate de Thompson de l'expression régulière du motif à rechercher.

La figure 3.16 montre l'automate ascendant de Thompson pour la recherche de motif de l'expression régulière de l'exemple précédent.

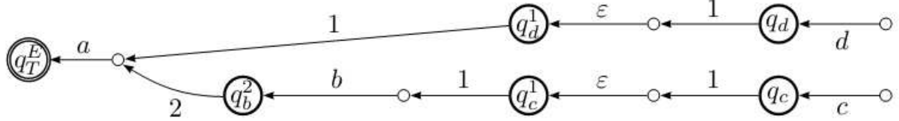


FIGURE 3.16 – Automate ascendant pour la RMA de l'expression E .

3.5.3 Algorithme ascendant de reconnaissance de motifs

Notre algorithme de reconnaissance des motifs d'arbre utilise la fonction φ définie comme suit :

Soient Q_i, des sous-ensembles d'états de Q avec $i = 1 \ldots n$. Soit $t \in \Sigma$ un symbole d'arité r et soit Π le produit cartésien $Q_1 \times Q_2 \times ... \times Q_r$.

$$\varphi(t, \Pi) = \bigcup_{\pi \in \Pi} \{q \,/\, (t(\pi) \longrightarrow q) \in \Delta\}$$

$\varphi(t, \Pi)$ c'est l'ensemble de tous les états atteignables possibles à partir des configurations Π en lisant t.

Soit E une expression régulière représentant le motif d'arbre recherché. Soit t l'arbre objet. L'algorithme 1 représente notre algorithme de recherche de motif d'arbre. Dans cet algorithme nous utilisons :

- l'automate de Thompson ascendant construit pour l'expression régulière du motif E,
- la fonction φ définie précédemment,
- et la forme post-fixée de l'arbre objet t.

La sortie de l'algorithme est l'ensemble *set–match* constitué des nœuds correspondants au motif.

Algorithme 1 Algorithme ascendant de RMA

Entrées: $Th(Q, \Delta)$: Automate de Thompson de E.
t : un pointeur vers l'arbre objet (en forme post-fixée).
Sorties: *set–match* : ensemble des nœuds correspondant au motif.

set–match $= \{\}$;
tantque ($t <> nil$) **faire**
  si ($r(t) = 0$) **alors**
    Empiler ($\Delta(t)$) ;
  sinon
    pour ($i = 1...r(t)$) **faire**
        Q_i = Dépiler () ;
    fin pour
    pour tout ($q \in \varphi(t, Q_1 \times Q_2 \times ... \times Q_r)$) **faire**
      Empiler (q) ;
      si ($q = q_f$) **alors** // reconnu par l'automate
        Ajouter (*set–match*, t) ;
      finsi
    fin pour
  finsi
  $t = t.suivant$;
fin tantque
Afficher les éléments de *set–match* ;

Exemple illustratif de RMA

Reprenons l'expression régulière du motif de l'exemple précédent,
$E = a(b(c), d)$, et soient les arbres objets t_1, t_2, et t_3 contenants respectivement zéro, une et deux instances du motif.

$t_1 = a(a(d, c), c)$
$t_2 = a(a(c, b(c), a(b(c), d))$
$t_3 = a(a(b(c), d), a(a(b(c), d), d))$

La notation post–fixée des arbres objets est comme suit :

$t_1 = dcaca$
$t_2 = ccbacbdaa$
$t_3 = cbdaacbdada$

La figure 3.17 montre le déroulement de l'algorithme 1 sur le motif exprimé par l'expression régulière E et l'arbre objet t_2. Notons qu'avant tout empilement d'un état, nous lui appliquons la fonction ε-closure qui permet d'éliminer les ε-transitions.

Pointeur (t)	Actions	État de la pile
c cbacbdaa ↑	Empiler (Δ (c))	$\{q^1_c\}$
c c bacbdaa ↑	Empiler (Δ (c))	$\{q^1_c\}$ $\{q^1_c\}$
cc b acbdaa ↑	$Q_1 =$ Dépiler () Empiler $(\Delta(b(Q_1)))$	$\{q^2_b\}$ $\{q^1_c\}$
ccb a cbdaa ↑	$Q_1 =$ Dépiler () $Q_2 =$ Dépiler () Empiler $(\Delta(a(Q_1, Q_2)))$	$\{q^1_a\}$
ccba c bdaa ↑	Empiler (Δ (c))	$\{q^1_c\}$ $\{q^1_a\}$
ccbac b daa ↑	$Q_1 =$ Dépiler () Empiler $(\Delta(b(Q_1)))$	$\{q^2_b\}$ $\{q^1_a\}$
ccbacb d aa ↑	Empiler (Δ (d))	$\{q^1_d\}$ $\{q^2_b\}$ $\{q^1_a\}$
ccbacbd a a ↑	$Q_1 =$ Dépiler () $Q_2 =$ Dépiler () Empiler $(\Delta(a(Q_1, Q_2)))$	$\{q_f\}$ $\{q^1_a\}$ ↑ Le nœud **a** correspond à un motif.
. . .	. . .	. . .

FIGURE 3.17 – Exemple de déroulement de notre algorithme de RMA.

3.6 Complexité de l'algorithme proposé

D'après l'algorithme 1, pour effectuer la recherche d'un motif exprimé par l'expression régulière d'arbre E dans l'arbre objet t, il faut pour chaque nœud x de l'arbre représentant E vérifier la correspondance de tous ces fils, à savoir l'arité du nœud x $(r(x))$ fois. Nous majorons cette dernière par l'arité maximale de l'alphabet notée R, ceci nous permet de conclure que la complexité de l'algorithme de recherche de motif est de l'ordre de $O(R^n)$ au pire des cas, où n est le nombre des nœuds de l'arbre motif, donc la longueur alphabétique de E.

L'algorithme ainsi présenté est d'une complexité théorique exponentielle tout comme l'algorithme de Thompson pour les mots. Par conséquent, il ne semble pas pratique. Mais si nous mettons en balance la performance (complexité) des algorithmes avec la difficulté à les implémenter, l'algorithme de Thompson est simple à implémenter.

Bien que la complexité de l'algorithme de Thompson pour les mots est de l'ordre de $O(2^n)$, il est largement utilisé dans la recherche de motif des mots, et la fameuse commande 'grep' de Linux en est le meilleur exemple.

En effet, en pratique sa performance a été prouvée. Selon des expérimentations effectuées sur le temps de traitement, il a été constaté que l'algorithme de Thompson traite une chaîne de 100 caractères en 200 microsecondes, alors que l'algorithme de recherche de motif du langage *Perl* en met plus de 10^{15} années [Cox07] ! Ce fait est espéré vrai en pratique pour notre approche.

Afin d'évaluer notre algorithme de RMA et de voir son comportement en pratique, il nous faudra un générateur d'arbres objets et le choix d'expression régulière de motif à rechercher.

Concernant les générateurs d'arbre, dans la littérature on en recense très peu et de plus ils sont peu performants [Dre10]. Pour cette raison et vu le temps imparti à ce travail, ceci fera l'objet de travaux futurs.

3.7 Conclusion

Dans ce chapitre, nous avons présenté notre approche de recherche de motif d'arbre. Notre contribution est basée sur la construction d'un automate d'arbre de Thompson depuis une expression régulière d'arbre par analogie à celui qui existe pour les mots. Nous avons donné les constructions des différents automates ainsi que les preuves d'équivalence entre les automates crées et les expressions régulières de départ. L'algorithme de recherche de motif utilisant ce genre de construction n'est pas optimal tenant compte de

la grande complexité dans les pires des cas, mais la facilité et la souplesse de l'implémentation justifie ce choix. Rappelons que le seul algorithme avec lequel on pourra comparer la performance et la complexité de notre approche est celui de Polách et al. [Pol11] présenté dans le second chapitre, qui est lui même d'une complexité exponentielle.

Conclusion et perspectives

La recherche de motif d'arbre est un problème très important en informatique. Il permet d'identifier les occurrences d'un ou plusieurs sous-arbres (motifs) dans un arbre objet. Ce problème est plus ou moins une généralisation de celui des mots. Parmi les applications de la RMA, on trouve le traitement de documents XML, la réécriture de termes, la génétique et particulièrement la recherche de motifs ADN/ARN...

Depuis 1982, plusieurs travaux ont tenté de résoudre le problème de recherche de motifs d'arbre. La plupart de ces travaux font appel à des techniques de recherche de motifs pour les mots pour les instrumenter dans leurs approches. En effet, les algorithmes utilisant la technique de 'stringpath' linéarisent l'arbre en un ensemble de chemins de caractères. De même, les algorithmes se basant sur les 'match-set' utilisent des ensembles générés à partir de la grammaire d'arbre.

D'autres algorithmes de recherche de motif utilisent des techniques modifiant la structure de l'arbre objet et l'arbre motif, tel que les arbres de suffixe, la convolution d'arbres...

Pour ce problème, notre approche a été différente. Nous avons voulu exploiter la force de l'algorithme de Thompson pour les mots en le généralisant aux arbres. En effet, l'algorithme de RM de Thompson pour les mots, basé sur la construction d'un automate à partir d'une expression régulière de mots, a eu un large champ d'application vu sa facilité d'implémentation et son efficacité.

La performance d'une telle approche est aussi ressenti par le fait que le motif est exprimé via les expressions régulières, une représentation connue pour sa puissance.

La particularité de notre approche est à souligner vu que dans la littérature nous ne trouvons pas de travaux ayant traité le problème de RMA en utilisant les expressions régulières. En effet, l'unique travail recensé [Pol11] utilise les automates à pile. Cependant, ce travail réalisé dans le cadre d'un Master, est incomplet et ne présente aucune preuve.

Notre algorithme de RMA est réalisé en deux phases : nous commençons par la construction d'un automate d'arbre particulier, similaire à l'automate de Thompson pour les mots, reconnaissant l'expression régulière d'arbre représentant le motif. Cette construction se fait par induction sur la structure de l'expression régulière comportant les opérations d'arité, d'union, de concaténation et de fermeture.

La seconde phase est celle de la recherche de motif, où nous avons proposé un algorithme utilisant l'automate de Thompson ascendant construit précédemment. Cet algorithme permet d'identifier toutes les occurrences du motif dans l'arbre objet à travers le parcours de cet automate.

Dans cette première version, notre algorithme de recherche est d'une complexité théorique de l'ordre de $O(R^n)$ au pire des cas, où n est nombre des nœuds du motif et R est l'arité maximale de Σ. Bien que cette complexité est exponentielle, l'algorithme est simple à implémenter.

Perspectives

Bien que notre approche pour la résolution du problème de RMA est particulière et simple à implémenter, nous prévoyons d'améliorer cet apport à travers les travaux suivants :

- Effectuer une étude expérimentale de notre approche. En effet, cette étude nous donnera une idée sur le comportement en pratique de notre algorithme d'une part. D'autre part, ceci nous permettra de comparer notre approche de RMA à celles de la littérature en termes de complexité temporelle et spatiale. Cette étude nécessite le choix d'un générateur d'arbres et le choix d'un échantillon d'expressions régulières de motifs à rechercher représentatives. Cet axe est déjà entamé.
- Améliorer la complexité de notre algorithme de RMA en améliorant la construction de l'automate, notamment celle de la fonction d'arité vu que c'est la construction qui fait exploser le nombre d'états de l'automate, et par conséquent augmente le temps de son parcours durant la recherche de motif.
- Déterminisation (au sens : élimination des transitions ε) de l'automate de Thompson construit et la comparaison entre cet automate et l'automate de position [LSZ13] de façon similaire à ce qu'a été effectuée pour les mots.

Bibliographie

[AC75] Alfred V. Aho and Margaret J. Corasick. Efficient string matching : An aid to bibliographic search. *Commun. ACM*, 18(6) :333–340, June 1975.

[AG85] Alfred V. Aho and Mahadevan Ganapathi. Efficient tree pattern matching (extended abstract) : An aid to code generation. In *Proceedings of the 12th ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages*, POPL '85, page 334–340, New York, NY, USA, 1985. ACM.

[AGT89] Alfred V. Aho, Mahadevan Ganapathi, and Steven W. K. Tjiang. Code generation using tree matching and dynamic programming. *ACM Trans. Program. Lang. Syst.*, 11(4) :491–516, October 1989.

[Ber05] Jean Berstel. Support de cours, automates et grammaires. Université de Marne-la-Vallée, France, 2004-2005.

[BKMW01] Anne Brüggemann-Klein, Makoto Murata, and Derick Wood. Regular tree and regular hedge languages over unranked alphabets : Version 1, 2001.

[Bor03] Björn Borchardt. The myhill-nerode theorem for recognizable tree series. In Zoltán Ésik and Zoltán Fülöp, editors, *Developments in Language Theory*, volume 2710 of *Lecture Notes in Computer Science*, pages 146–158. Springer, 2003.

[Büc62] Julius R. Büchi. On a decision method in restricted second order arithmetic. In Ernest Nagel, Patrick Suppes, and Alfred Tarski, editors, *Proceedings of the 1960 International Congress on Logic, Methodology and Philosophy of Science (LMPS'60)*, pages 1–11. Stanford University Press, June 1962.

[CDF07] Rafael Carrasco, Jan Daciuk, and Mikel Forcada. An implementation of deterministic tree automata minimization. In Jan Holub and Jan Žďárek, editors, *Implementation and Application of Automata*, volume 4783 of *Lecture Notes in Computer Science*, pages 122–129. Springer Berlin Heidelberg, 2007.

[CDG+08] H. Comon, M. Dauchet, R. Gilleron, C. Loding, F. Jacquemard, D. Lugiez, S. Tison, and M. Tommasi. Tree automata techniques and applications. Available on : `http://www.grappa.univ-lille3.fr/tata`, 2008. release November, 18th 2008.

[CH03] Richard Cole and Ramesh Hariharan. Tree Pattern Matching To Subset Matching In Linear Time. *SIAM J. COMPUT. Society for Industrial and Applied Mathematics*, 2003.

[Cha87] D. R. Chase. An improvement to bottom-up tree pattern matching. In *Proceedings of the 14th ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages*, POPL '87, page 168–177, New York, NY, USA, 1987. ACM.

[CHI99] Richard Cole, Ramesh Hariharan, and Piotr Indyk. Tree pattern matching and subset matching in deterministic o(n log3 n)-time. In *Proceedings of the Tenth Annual ACM-SIAM Symposium on Discrete Algorithms*, SODA '99, page 245–254, Philadelphia, PA, USA, 1999. Society for Industrial and Applied Mathematics.

[Chu63] Alonzo Church. Logic, arithmetic, and automata. In *Proceedings of the International Congress of Mathematicians (ICM'62)*, pages 23–35, 1963.

[CHZ05] Loek G. Cleophas, Kees Hemerik, and Gerard Zwaan. A missing link in root-to-frontier tree pattern matching. In Jan Holub and Milan Simánek, editors, *Stringology*, pages 216–230. Department of Computer Science and Engineering, Faculty of Electrical Engineering, Czech Technical University, 2005.

[Cle08] Loeck Cleophas. *Tree Algorithms : Two Taxonomies and a Toolkit.* Phd, Technische Universiteit Eindhoven, Department of Mathematics and Computer Science, 2008.

[Cox07] Russ Cox. Regular expression matching can be simple and fast, 2007.

[DGM94] Moshe Dubiner, Zvi Galil, and Edith Magen. Faster tree pattern matching. *J. ACM*, 41(2) :205–213, March 1994.

[DH03] Frank Drewes and Johanna Högberg. Learning a regular tree language from a teacher. In *Proceedings of the 7th International Conference on Developments in Language Theory*, DLT'03, page 279–291, Berlin, Heidelberg, 2003. Springer-Verlag.

[Don65] J. E. Doner. Decidability of the weak Second-Order theory of two successors. *Notices Amer. Math. Soc.*, 12, March 1965.

[Don70] John Doner. Tree acceptors and some of their applications. *J. Comput. Syst. Sci.*, 4(5) :406–451, 1970.

[Dre10] Frank Drewes. Towards the tree automata workbench marbles. *Electronic Communications of the EASST*, 2010.

[E.F56] E.F. Moore. Gedanken-experiments on sequential machines. In C.E. Shannon and J. MacCarthy, editors, *Automata Studies*, pages 129–153, Princeton, New Jersey, 1956. Princeton University Press.

[ÉF03] Zoltán Ésik and Zoltán Fülöp, editors. *Developments in Language Theory, 7th International Conference, DLT 2003, Szeged, Hungary, July 7-11, 2003, Proceedings*, volume 2710 of *Lecture Notes in Computer Science*. Springer, 2003.

[GK00] Thomas Genet and Francis Klay. Rewriting for cryptographic protocol verification. In David A. McAllester, editor, *CADE*, volume 1831 of *Lecture Notes in Computer Science*, pages 271–290. Springer, 2000.

[Glu61] Victor M. Glushkov. The abstract theory of automata. *Russian Mathematical Surveys*, 16(5) :1–53, 1961.

[Gou00] Jean Goubault-Larrecq. A method for automatic cryptographic protocol verification (extended abstract). In José D. P. Rolim, editor, *Proceedings of the Workshops of the 15th International Parallel and Distributed Processing Symposium*, volume 1800 of *Lecture Notes in Computer Science*, pages 977–984, Cancun, Mexico, May 2000. Springer.

[HC86] Philip J. Hatcher and Thomas W. Christopher. High-quality code generation via bottom-up tree pattern matching. In *POPL*, pages 119–130. ACM Press, 1986.

[HK86] C. Hemerik and J. P. Katoen. Bottom-up tree acceptors. In *Science of Computer Programming*, pages 13–51, 1986.

[HO82a] Christoph M. Hoffmann and Michael J. O'Donnell. Pattern matching in trees. *J. ACM*, 29(1) :68–95, January 1982.

[HO82b] Christoph M. Hoffmann and Michael J. O'Donnell. Programming with equations. *ACM Trans. Program. Lang. Syst.*, 4(1) :83–112, January 1982.

[Kle56] S. C. Kleene. Representation of events in nerve nets and finite automata. In Claude Shannon and John McCarthy, editors, *Automata Studies*, pages 3–41. Princeton University Press, Princeton, NJ, 1956.

[KM08] Dietrich Kuske and Ingmar Meinecke. Construction of tree automata from regular expressions. *Developments in Language Theory, Springer*, 2008.

[KMP77] Donald E. Knuth, J.H. Morris, and Vaughan R. Pratt. Fast pattern matching in strings. *SIAM Journal of Computing*, 6(2) :323–350, 1977.

[Kos89] S.R. Kosaraju. Efficient tree pattern matching. In *Foundations of Computer Science, 1989., 30th Annual Symposium on*, pages 178–183, 1989.

[LSZ13] Éric Laugerotte, Nadia Ouali Sebti, and Djelloul Ziadi. From regular tree expression to position tree automaton. In *LATA*, 2013.

[Mea55] George H. Mealy. A method for synthesizing sequential circuits. *Bell System Technical Journal*, 34(5) :1045–1079, 1955.

[MLMK05] Makoto Murata, Dongwon Lee, Murali Mani, and Kohsuke Kawaguchi. Taxonomy of XML Schema Languages Using Formal Language Theory. *ACM Trans. Internet Technol.*, 5(4) :660–704, November 2005.

[Mon99] David Monniaux. Abstracting cryptographic protocols with tree automata. In Agostino Cortesi and Gilberto Filé, editors, *SAS*, volume 1694 of *Lecture Notes in Computer Science*, pages 149–163. Springer, 1999.

[MP88] Warren S. McCulloch and Walter Pitts. Neurocomputing : Foundations of research. chapter A Logical Calculus of the Ideas Immanent in Nervous Activity, pages 15–27. MIT Press, Cambridge, MA, USA, 1988.

[Mur99] Makoto Murata. Hedge automata : a formal model for XML schemata, 1999.

[MY60] R. McNaughton and H. Yamada. Regular expressions and finite state graphs for automata. *IRE Trans. on Electronic Comput*, EC-9(1) :38–47, 1960.

[Nev02a] Frank Neven. Automata, Logic, and XML. In *In CSL'02 - Annual Conference of the European Association for Computer Science Logic (invited talk*, pages 2–26. Springer, 2002.

[Nev02b] Frank Neven. Automata Theory for XML Researchers. *SIGMOD Rec.*, 31(3) :39–46, September 2002.

[O'D85] Michael J. O'Donnell. *Equational logic as a programming language.* MIT Press series in the foundations of computing. Cambridge, Mass. MIT Press, 1985. Includes index.

[Pol11] Radomír Polách. *Tree Pattern Matching and Tree Expressions.* Matser, Czech Technical University in Prague, Faculty of Electrical Engineering, Department of Computer Science and Engineering, 2011.

[Rab69] Michael O. Rabin. Decidability of Second Order Theories and Automata on Infinite Trees. *Transactions of the American Mathematical Society*, 141 :1–35, 1969.

[RR92] R. Ramesh and I. V. Ramakrishnan. Nonlinear pattern matching in trees. *Journal of the ACM*, 39 :295–316, 1992.

[Sch07] T Schwentick. Automata for XML– a survey. *Journal of Computer and System Sciences*, January 2007.

[Str07] Roger Strolenberg. *ForestFIRE and FIREWood A Toolkit & GUI for Tree Algorithms.* Master, Technische Universiteit Eindhoven, Department of Mathematics and Computer Science, 2007.

[SZ90] Bruce A. Shapiro and Kaizhong Zhang. Comparing multiple rna secondary structures using tree comparisons. *Computer Applications in the Biosciences*, 6(4) :309–318, 1990.

[Tha67] James W. Thatcher. Characterizing derivation trees of context-free grammars through a generalization of finite automata theory. *J. Comput. Syst. Sci.*, 1(4) :317–322, 1967.

[Tha70] James W. Thatcher. Generalized sequential machine maps. *J. Comput. Syst. Sci.*, 4(4) :339–367, 1970.

[Tha73] J. Thatcher. Tree automata : An informal survey. In A. Aho, editor, *Currents in the Theory of Computing*, pages 143–172. Prentice-Hall, 1973.

[Tho68] Ken Thompson. Programming techniques : Regular expression search algorithm. *Commun. ACM*, 11(6) :419–422, June 1968.

[Tra95] Boris A. Trakhtenbrot. Origins and metamorphoses of the trinity : Logic, nets, automata. In *Proceedings of the Tenth Annual IEEE Symposium on Logic in Computer Science (LICS 1995)*, pages 506–507. IEEE Computer Society Press, June 1995. Invited Talk.