Recherche de sous-structures arborescentes ordonnées fréquentes au sein de bases de données semi-structurées

Federico del Razo Lopez

Numéro d'identification :

UNIVERSITÉ MONTPELLIER II
— SCIENCES ET TECHNIQUES DU LANGUEDOC —

THÈSE

pour obtenir le grade de
DOCTEUR de l'Université Montpellier II

Discipline : **Informatique**
École Doctorale : **Information, Structures, Systèmes**

présentée et soutenue publiquement par

Federico DEL RAZO LÓPEZ

le 16 juillet 2007

Recherche de sous-structures arborescentes ordonnées fréquentes au sein de bases de données semi-structurées

Jury :

M. Georges GARDARIN, Professeur, Université Versailles Saint-Quentin, ... Président
M. Pascal PONCELET, Professeur, Ecole des Mines d'Alès, ... Directeur de thèse
Mme. Maguelonne TEISSEIRE, MCF, Université Montpellier II, ... Co-directrice
Mme. Anne LAURENT, MCF, Université Montpellier II, ... Co-directrice
M. Jean-Daniel ZUCKER, Professeur, Université Paris 13 (Nord), ... Rapporteur
M. Mohand-Said HACID, Professeur, Université Claude Bernard - Lyon, ... Rapporteur

À Claudia,
Fred, Fer, Icar, [1]
à mes Parents et Frères. [2]

[1] Je voudrais dédier ce travail à mon épouse et mes enfants pour m'avoir soutenu pendant ces 4 années d'études et pour avoir accepté de sacrifier autant de choses pour moi.

[2] a mis padres y hermanos.

Remerciements

Je tiens à remercier Monsieur Jean-Daniel ZUCKER et Monsieur Mohand-Saïd HACID pour avoir accepté de rapporter cette thèse et avoir consacré du temps à sa lecture. Leurs observations m'ont été précieuses pour améliorer la qualité de mon mémoire.

Je remercie également Monsieur Georges GARDARIN pour avoir accepté la présidence de mon jury de thèse.

Je remercie infiniment mes directeurs de thèse Monsieur Pascal PONCELET, Madame Maguelonne TEISSEIRE et Madame Anne LAURENT, pour avoir mis à ma disposition toutes leurs compétences scientifiques et surtout pour leur patience. J'apprécie la confiance qu'ils m'ont toujours accordée pendant mes travaux de recherche.

Pour mes collègues Simon JAILLET, Céline FIOT, Chedy RAISSI, Marc PLANTEVIT, Stéphane SANCHEZ et Khalid SALEM, je souhaite qu'ils trouvent ici l'expression de ma plus sincère sympathie et de mon amitié.

Je remercie également Madame Mary LOGAY et Cécile LOW KAM pour leurs corrections sur mon manuscrit.

Je souhaite m'adresser plus spécialement à mes beaux-parents en les remerciant pour leur soutien et leurs encouragements tout au long de ces années en France[3].

Je remercie la "Dirección General de Educación Superior Tecnológica (DGEST - Mexinque)" et particulièrement le "Instituto Tecnológico de Toluca", pour m'avoir accordé le financement indispensable à la réalisation de ce projet. Je voudrais également profiter de cette page pour remercier Javier GOMEZ LUGO, Concepción RODRIGUEZ SAAVEDRA et Rogelio CRUZ pour leur appui.

[3]Deseo dirigirme espécialmente a mis suegros agradeciéndoles todo su aliento y apoyo a lo largo de nuestra estancia en Francia.

Résumé

La recherche de structures arborescentes fréquentes, également appelée fouille d'arbres, au sein de bases de données composées de documents semi-structurés (e.g. XML) est un sujet de recherche actuellement très actif. Ce processus trouve de nombreux intérêts dans le contexte de la fouille de données, comme par exemple la construction automatique d'un schéma médiateur à partir de schémas XML, ou bien l'analyse des structures des sites Web afin d'étudier leur usage ou modifier leur contenu.

L'objectif de cette thèse est de proposer une méthode d'extraction d'arborescences ordonnées et étiquetées fréquentes. Cette approche est basée sur une représentation compacte des arborescences cherchant à diminuer la consommation de mémoire dans le processus de fouille. En particulier, nous présentons une nouvelle technique de génération d'arborescences candidates visant à réduire leur nombre. Par ailleurs, nous proposons différents algorithmes pour valider le support des arborescences candidates dans une base de données selon divers types de contraintes d'inclusion d'arbres : induite, incrustée et floue. Finalement nous appliquons nos algorithmes à des jeux de données synthétiques et réels, et nous présentons les résultats obtenus.

Mots-clés : Extraction de connaissances, Fouille de données, fouille d'arbres, XML, inclusion d'arbres, schéma médiateur, sous-arbres, énumération d'arborescences.

Abstract

Tree mining is defined as finding frequent tree structures in large set of trees. Currently, its application in the field of semi-structured databases (e.g. XML) is a very active research problem. It finds many interesting applications in this context. For example, automatic construction of mediated schema for a large set of XML schemas, or the analysis of browsing patterns of web sites in order to modify their contents for better response.

The objective of this thesis is to propose a method for mining frequent rooted, ordered and labeled tree structures. Our approach is based on a compact tree structure representation for minimizing the memory consumption in the process of mining. In particular we present a new technique for reducing the number of candidate tree structures. In addition, we propose several algorithms to validate the support of candidate tree structures according to various types of tree inclusion constraints : induced, embedded and fuzzy. Finally we demonstrate the application and performance of our algorithms on real and synthetic data sets.

Key words : Knowledge data discovery, data mining, tree mining, XML, tree inclusion, mediated schema, sub-trees, tree generation

Table des matières

1. Introduction **1**

1.1. Les règles d'association . . . 3
1.2. Les données semi-structurées . . . 4
1.3. Extraction de sous-structures fréquentes . . . 5
1.4. Motivation . . . 5
1.5. Contributions . . . 6
1.6. Organisation du mémoire . . . 7

2. Fondements et problématique **9**

2.1. Introduction . . . 10
2.2. Fouille de données transactionnelles . . . 11
2.2.1. Recherche d'itemsets fréquents . . . 11
2.2.2. L'algorithme Apriori . . . 13
2.2.3. Recherche de règles d'association . . . 16
2.3. Fouille de données semi-structurées . . . 17
2.3.1. Définition et propriétés des arbres . . . 18
2.3.2. L'algorithme Apriori sur les arbres . . . 23
2.3.3. Génération d'arbres candidats . . . 24
2.3.4. Validation d'arborescences candidates . . . 29
2.3.5. Occurrences d'un arbre motif dans une base de données . . . 33
2.4. Énoncé du problème . . . 34
2.5. Conclusion . . . 35

3. Approches existantes **37**

3.1. Introduction . . . 38
3.2. L'algorithme Freqt, une approche Apriori en largeur . . . 39
3.2.1. Génération de candidats . . . 42
3.2.2. Validation du support . . . 42
3.3. L'algorithme TreeMiner, une approche Apriori en profondeur . . . 44

3.3.1. Génération de candidats . . . 46
3.3.2. Validation du support . . . 48
3.4. L'algorithme TIDES, une approche PATTERN-GROWTH . . . 52
3.4.1. Algorithme . . . 53
3.5. L'algorithme TREEFINDER, une méthode logique inductive . . . 57
3.5.1. Algorithme . . . 58
3.6. Conclusion . . . 61

4. Le modèle de données **63**
4.1. Introduction . . . 64
4.2. Le Modèle Objet de Document (DOM) . . . 65
4.2.1. Les composants du DOM . . . 67
4.3. Une structure de données efficace pour la représentation d' arbres . . . 72
4.3.1. Notre proposition . . . 73
4.3.2. Représentation par un tableau de pères . . . 73
4.3.3. Représentation binaire . . . 74
4.4. Conclusion . . . 76

5. RSF : une approche pour la fouille d'arbres **77**
5.1. Introduction . . . 78
5.2. Algorithme RSF . . . 79
5.3. Génération de candidats . . . 82
5.3.1. Génération de candidats par la branche la plus à droite . . . 82
5.3.2. Pivot : une nouvelle méthode pour la génération de candidats . . . 83
5.3.3. Preuve de complétude . . . 94
5.4. Validation du support des candidats . . . 95
5.4.1. RSF_i, une approche *induite* . . . 95
5.4.2. RSF_e, une approche *incrustée* . . . 97
5.4.3. RSF_f, une approche *floue* . . . 97
5.5. Conclusion . . . 107

6. Expérimentations **109**
6.1. Implémentation . . . 110
6.2. Expérimentations sur des données synthétiques . . . 110
6.2.1. Jeux de données . . . 110
6.3. Expérimentations sur des données réelles . . . 112
6.3.1. Jeux de données . . . 112
6.4. Conclusions . . . 113

7. Conclusion et Perspectives **123**
7.1. Introduction 123
7.2. Contributions 123
7.3. Perspectives 125

Bibliographie **129**

Bibliographie Personnelle **139**

Chapitre 1

Introduction

Les données ont d'abord été traitées par les entreprises comme des *données brutes*[1] servant aux taches quotidiennes (e.g. registres des présences horaires, gestion des salaires, gestion des commandes). Dans ce contexte, les bases de données ont permis aux sociétés de traiter les données de manière efficace et cohérente. Plus récemment les entreprises ont voulu s'appuyer sur ces données pour obtenir de l'information[2]. Les applications, regroupées sous le terme d'*informatique opérationnelle*, ont été conçues pour privilégier les performances d'opérations répétitives de lecture, ajout, suppression et de modification que les utilisateurs appliquent sur les données. Une application opérationnelle est capable de donner une réponse précise à des questions simples du type : qui ?, quand ?, quoi ?, où ?

Motivées pour diverses raisons telles que les obligations légales, la sécurité (détection de fraudes) ou le suivi des clients ou des marchés, les entreprises ont décidé de stocker l'historique de l'ensemble de leurs données. Bientôt, celles-ci se sont rendu compte que ces données contenaient également des connaissances potentiellement intéressantes.

L'architecture client-serveur et le stockage de données ont permis aux organisations la création de nouveaux systèmes d'information nommés *entrepôts de données*[3]. Ceux-ci ont alors été utilisés pour la production de rapports qui constituaient une aide à la prise de décisions, nécessitant le développement des systèmes d'information qui peuvent offrir une aide à la prise de décisions. On parle alors *d'informatique décisionnelle*.

Parallèlement à l'avènement des entrepôts de données, les communautés statistiques et informatiques (intelligence artificielle, base de données) ont développé des méthodes d'extraction de connaissances, pour l'apprentissage automatique, l'analyse de données et la fouille de données. Cette connaissance cherche à répondre à des questions plus ouvertes

[1] de l'anglais *raw data*

[2] à la différence des données brutes qui par eux-mêmes manquent de sens, l'information est définie par [CS90] comme « des données plus du sens » (de l'anglais *information = data + meaning*)

[3] de l'anglais *datawarehouse*

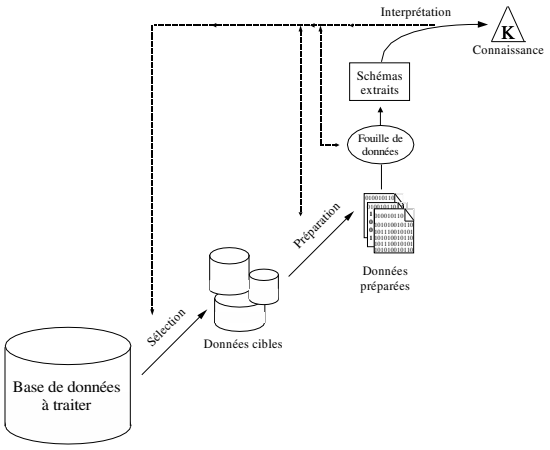


FIG. 1.1 – Les étapes du processus d'extraction de connaissances dans les bases de données [UGP96]

et complexes dont la solution ne peut être trouvée par des langages de requêtes traditionnels(e.g. *SQL*).

Le processus *d'Extraction de Connaissances dans les bases de Données* (ECD) [4] est défini par [FPSS96] comme : « **un processus non trivial qui consiste à identifier, dans les données, des schémas nouveaux, valides, potentiellement utiles et surtout compréhensibles et utilisables** ». Grâce aux bons résultats obtenus et à la maturité des techniques pour l'extraction de connaissance, l'ECD a dépassé son domaine d'origine (les entreprises) pour être employée dans divers champs d'application tels que la médecine, la biologie, la finance, etc. L'ECD est un processus interactif et itératif composé de différentes étapes (c.f. Figure 1.1) groupées globalement dans les trois phases suivantes :

- **La préparation des données** où l'objectif consiste à sélectionner les données potentiellement utiles.
- **La fouille de données**[5] pour l'extraction de connaissances à partir des données.
- **Interprétation des résultats** avec pour but l'interprétation de la connaissance extraite lors de la phase précédente, pour la rendre lisible et compréhensible par l'utilisateur et permettre ainsi de l'intégrer dans le processus de décision.

Afin de définir le contexte de notre problématique, nous nous situons dans la phase de la « fouille de données » considérée comme le cœur du processus ECD. Cette étape met en évidence des relations (i.e. des modèles, des motifs ou des schémas) contenus implicitement dans de grandes collections de données[Han01, HSM01]. Notons que la fouille de données

[4] Le sigle ECD est une traduction de l'anglais *Knowledge Discovery in Databases (KDD)*) qui fut introduit par Piatetsky-Shapiro en 1989 lors d'un workshop de la conférence IJCAI'89.

[5] de l'anglais *data mining*

se décompose essentiellement en trois grandes familles de méthodes : 1) les *méthodes descriptives*, issues de la statistique descriptive et de l'analyse des données, et adaptées ou augmentées à l'aide de techniques de visualisation graphique, 2) les *méthodes de structuration* qui regroupent toutes les techniques d'apprentissage non supervisé et de classification automatique, et 3) les *méthodes explicatives* qui cherchent à établir un modèle décrivant un phénomène. Parmi les techniques employées pour la fouille de données dans les méthodes décrites au-dessus, nous pouvons citer :

La Classification supervisée qui analyse de nouvelles données et les affecte, en fonction de leurs caractéristiques ou attributs, à telle ou telle classe pré-définie [BFSO84, WK91].

La Classification non supervisée ou *Clustering*, qui est différente de la classification supervisée dans le sens où il n'existe pas de classes prédéfinies[JD88] : l'objectif est de grouper des enregistrements qui semblent similaires dans une même classe.

Les Dépendances Fonctionnelles, qui recherche les dépendances entre les données et offre ainsi un outil d'aide à la décision pour les administrateurs de bases de données, les développeurs d'applications, concepteurs et intégrateurs de systèmes d'information. En effet, les applications relèvent de l'administration et du contrôle des bases de données, de l'optimisation de requêtes ou encore de la rétro-conception de systèmes d'information [KMRS92, HKPT98, LMP00, NC00].

Les Séries Chronologiques, dont l'objectif est de trouver des portions de données (ou séquences) similaires à une portion de données précise, ou encore de trouver des groupes de portions similaires issues de différentes applications [AFS93, ALSS95, FRM94].

1.1. Les règles d'association

Dans ce travail de thèse nous sommes tout particulièrement intéressés à la technique de la découverte de « règles d'association » permettant la recherche de corrélations, de liens et d'implications entre les données. Motivés par la nécessité d'analyser les transactions dans un supermarché afin d'examiner le comportement des clients en terme de produits achetés, Agrawal et al. dans [AIS93] ont introduit le problème de l'extraction des règles d'association. Ce problème, traditionnellement lié au secteur de la distribution dans les grandes surfaces, a par objectif l'analyse du comportement des consommateurs (i.e. en analysant les tickets de caisse), afin d'améliorer les ventes, réduire les coûts, optimiser les offres ou bien améliorer la gestion de stocks. On parle alors du problème du panier de la ménagère[6], résumé comme suit [AIS93] :

[6] *The market basket problem*

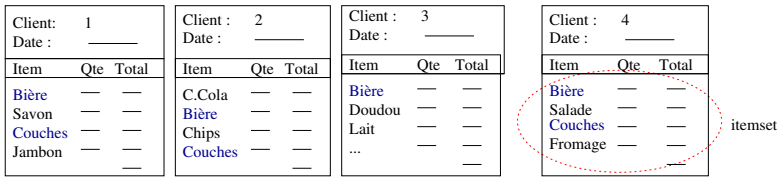


FIG. 1.2 – L'ensemble des transactions d'un consommateur

Problème 1 (Panier de la ménagère) *Étant donné une base de transactions (les paniers), chacune composée de produits ou d'articles appelés* items*, la découverte de règles d'association consiste à chercher des ensembles d'items, fréquemment liés dans une même transaction, ainsi que des règles les combinant.*

La découverte de règles d'association est conditionnée par l'extraction d' « itemsets fréquents », c'est-à-dire des ensembles de produits achetés par un client au cours de la même transaction et qui sont présents dans plus d'un certain seuil appelé support minimal.

Exemple 1 *Dans la figure 1.2, on extrait la règle d'association qui révèle que « 75% des gens qui achètent de la bière, achètent également des couches ». Cette association est confirmée si on regarde l'ensemble des items (articles) achetés pour chaque client et on constate que trois des quatre clients qui achètent de la bière achètent aussi des couches.*

1.2. Les données semi-structurées

Comme nous venons de le voir ci-dessus, l'extraction de connaissance à partir des bases de données se fait en employant différentes techniques. Cependant ces bases ont évolué et à présent elles se situent dans un cadre plus générique qui intègre des données de type brute outre des données semi-structurées, et structurées. Les données brutes sont les données dans l'état où elles sont recueillies, c'est-à-dire, qu'il s'agit de données non classifiées, non-traitées, non-agrégées ou non-comprimées (i.e. des fichiers contenant du texte brut, du son, des images, etc.). Les données structurées ont une structure rigide ou régulière (i.e. les différents modèles traditionnels de bases de données). Les données semi-structurées, quant à elles, concernent les données caractérisées par des structures irrégulières, dynamiques ou inconnues, c'est-à-dire que les données semi-structurées ne se conforment pas à un schéma fixe [Abi97].

Une caractéristique des données semi-structurées est leur flexibilité pour pouvoir représenter n'importe quel type d'information, ce qui les rend idéales pour représenter n'importe quel type d'entité. En revanche, l'intégration de données semi-structurées (avec une absence de schéma pré-défini ainsi qu'une structure arborescente), constitue une base de données fortement hétérogène difficile à être manipulée, interrogée et représentée par des traitements

classiques. D'où la nécessité de proposer des améliorations ou adaptations, ou bien d'inventer de nouveaux concepts et algorithmes pour la fouille de données au sein des bases de données semi-structurées.

De nos jours, le langage XML (permettant la représentation des données semi-structurées) est devenu le standard prépondérant sur Internet, donc la recherche de moyens d'intégration de tels schémas est indispensable. Dans le but de proposer une approche permettant de répondre à cette dernière problématique, nous nous focalisons sur la recherche de sous-structures fréquentes au sein d'une base de données de schémas XML.

1.3. Extraction de sous-structures fréquentes

Considérant que les documents semi-structurés ou XML sont représentés par des structures arborescentes nous traitons les bases de données semi-structurées comme des bases de données composées de telles structures appelées simplement *bases arborescentes*. C'est dans ce contexte que nous situons l'objectif principal de ce travail de thèse : « La fouille sous-structures fréquentes au sein de bases arborescentes ».

Une sous-structure fréquente est un sous-arbre se trouvant dans *la plupart* des schémas XML considérés. C'est une fréquence au sens d'un *support* qui correspond à un nombre minimal d'arborescences de la base dans lesquelles doit se retrouver le sous-arbre pour être considéré comme *fréquent*.

1.4. Motivation

La fouille de données arborescentes est une problématique actuellement très active [AAK+02, Zak02, TRS02, CYM04a, TRS04, WYZ+04] qui suscite de nombreux intérêts dans le contexte de la fouille de données comme par exemple :

La construction de schémas médiateurs L'explosion du volume de données disponibles sur Internet conduit aujourd'hui à réfléchir sur les moyens d'interroger les grosses masses d'information afin de retrouver les informations souhaitées. Les utilisateurs ne pouvant pas connaître les modèles sous-jacents des données auxquelles ils souhaitent accéder, il est nécessaire de leur fournir les outils automatiques de définition de schémas médiateurs. Un schéma médiateur fournit une interface permettant l'interrogation des sources de données par l'utilisateur au travers de requêtes. L'utilisateur pose alors ses requêtes de manière transparente à l'hétérogénéité et la répartition des données. XML étant maintenant prépondérant sur Internet, la recherche de moyens d'intégration de tels schémas est indispensable. Si les recherches permettant l'accès aux données quand un schéma d'interrogation est connu sont maintenant bien avancées [Xyl01], les recherches concernant la définition automatique d'un schéma médiateur restent incomplètes et sont donc non satisfaisantes [TBBT04].

La phylogénie, dont le but est la reconstruction de l'histoire des espèces. Elle suppose une évolution biologique au cours du temps se traduisant par des mutations du code génétique des individus et peut se représenter sous forme arborescente. La phylogénie se sert de la fouille de données pour trouver des similitudes dans les arbres phylogénétiques [SWZ04, ZW06, RK04].

1.5. Contributions

Un des objectifs majeurs de notre travail est de développer une approche permettant de répondre à la problématique de l'extraction de sous-structures arborescentes fréquentes, en nous focalisant sur la recherche de sous-structures fréquentes. Ainsi dans cette thèse nous proposons :

Un modèle de données pour la fouille d'arbres. L'extraction ou recherche d'arborescences fréquentes est complexe dans la mesure où il est nécessaire de traduire l'ensemble des schémas en une structure aisément manipulable. Cette transformation des données conduit parfois à doubler ou tripler la taille de la base initiale dès lors que l'on souhaite utiliser des propriétés spécifiques permettant d'améliorer le processus de fouille. Il n'existe pas de solution efficace à ce problème alliant une représentation compacte à des propriétés intéressantes. Un des buts de ce mémoire est la définition d'une structure répondant à cet objectif [LLT05, LLT06, LSL+07] puisque les propriétés de la représentation proposée permettent une génération des candidats et un élagage aussi performants que les approches de référence [AAK+02, Zak02, TRS02].

Un algorithme optimisé de génération de candidats Les approches traditionnelles de fouille d'arbres permettent d'obtenir des sous-structures ou des sous-arbres fréquents en utilisant des stratégies de type « générer-élaguer ». Même si elles sont efficaces, elles sont cependant pénalisées par le fait que le nombre de candidats générés est trop important. L'algorithme Pivot est proposé pour réduire considérablement le nombre de sous-arbres candidats générés. L'originalité de cette technique réside dans l'utilisation d'un *pivot* [LLPT07b] qui permet de définir des classes d'équivalence d'arbres destinées à optimiser l'étape de génération de candidats.

Un ensemble d'algorithmes pour la fouille de sous-structures fréquentes à partir des bases de données semi-structurées (i.e. des arborescences enracinées, étiquetées et ordonnées). Ces algorithmes sont groupés sous l'appellation RSF(*Recherche de sous-Structures Fréquentes*) [LLPT07a, LLPT05, LLPT06]. Les algorithmes à l'intérieur de l'approche RSF sont basés sur l'approche Apriori[AIS93](une méthode d'extraction de motifs de type *générerélaguer*) mais nous proposons aussi une implantation de type Pattern-Growth (dite *sans génération de candidats*) présentée par [TPK06]. Nos

algorithmes sont basés sur le modèle de données mentionné ci-dessus dont nous exploitons les propriétés de la représentation vectorielle des arbres. Cette représentation séquentielle est utilisée de façon optimale dans l'étape de *génération de candidats* et dans la validation du support des candidats en considérant des inclusions de type : *induite*, *incrustée*, et *floue*.

1.6. Organisation du mémoire

Ce mémoire est structuré de la manière suivante :

- le chapitre 2, « Fondements et problématique », expose les principes de fouille de données appliqués à l'extraction de règles d'association sur des données transactionnelles et semi-structurées. Nous introduisons quelques définitions et propriétés pour le traitement des arbres. Nous présentons la problématique liée à l'inclusion des arbres et finalement nous exposons formellement la problématique de la fouille sous-structures arborescentes fréquentes.
- le chapitre 3, « Approches existantes », présente les principaux travaux réalisés autour de l'extraction fouille d'arbres, en considérant l'approche Apriori avec différentes contraintes d'inclusion (e.g. induite, incrustée), l'approche dite par subsomption et l'approche Pattern-Growth.
- le chapitre 4, « Le modèle de données », détaille notre proposition pour le traitement des arbres en définissant une structure de données pour stocker les arbres ainsi qu'une *interface de programmation d'applications-(API)* (ensemble de fonctions appliqué sur les structures de données).
- le chapitre 5, « Rsf : une approche pour la fouille d'arbres », expose nos contributions pour la fouille d'arbres basée sur le modèle traité dans le chapitre précèdent. Dans un premier temps, l'algorithme Pivot détaille notre contribution visant à réduire le nombre de structures arborescentes candidates dans la étape de génération de candidats. Ensuite nous exposons nos contributions pour la fouille d'arbres en prenant compte les contraintes d'inclusion d'arbres : induite, incrustée et floue.
- le chapitre 6, « Expérimentations », présente l'ensemble des expérimentations amenées sur des données synthétiques et réelles pour valider nos approches.
- le chapitre 7, « Conclusion et perspectives », contient les conclusions que nous pouvons tirer de nos travaux ainsi que les perspectives qui permettraient de compléter notre approche.

Chapitre 2

Fondements et problématique

Dans ce chapitre, nous présentons d'abord l'approche appelée *Apriori* appliquée à la découverte de motifs fréquents au sein de bases de données transactionnelles. Ensuite, comme les données semi-structurées peuvent être décrites par des structures arborescentes, nous introduisons différentes fonctions utilisées pour traiter ces structures. Puis nous présentons une méthode pour la génération d'arborescences étiquetées et ordonnées et nous introduisons la problématique de l'inclusion des arbres. L'importance de ces deux opérations, la génération et l'inclusion des arbres, vient du fait qu'elles sont indispensables pour l'extraction d'arbres fréquents. Nous concluons en énonçant la problématique de la *recherche de structures arborescentes fréquentes* ou *fouille d'arbres*[a].

[a] de l'anglais *Tree mining*

Sommaire

2.1. Introduction **10**
2.2. Fouille de données transactionnelles **11**
2.3. Fouille de données semi-structurées **17**
2.4. Énoncé du problème **34**
2.5. Conclusion **35**

2.1. Introduction

Les techniques de fouille de données, en particulier la recherche d'itemsets fréquents et la recherche de règles d'association, ont été employées avec beaucoup de succès sur les données classiques appelées « données simples ». Ces données sont souvent représentées par des tableaux[HSM01] (e.g. les tables dans les SGBDR[1]). Les tableaux représentent un ensemble d'objets arrangés dans une matrice de taille $n \times c$. La première dimension correspond au nombre d'objets n dans l'ensemble, et chaque ligne représente des éléments référés comme des objets, individus, entités, ou registres selon le contexte. La deuxième dimension c, représente un ensemble de propriétés associées à chaque objet et appelées également attributs, champs ou dimensions selon le contexte. Considérant une représentation des données par des tableaux, nous appellerons « données transactionnelles » des données telles qu'une liste d'achats réalisés par un consommateur ou des données extraites à partir des habitudes de navigation et stockées sur les serveurs hébergeant les sites Web. Ce modèle de données réalise aujourd'hui de grandes performances dans les applications comme les bases de données relationnelles, où par l'intermédiaire d'un langage comme SQL on peut récupérer, trier, chercher une information très rapidement.

Cependant, les données transactionnelles posent certains problèmes pour la représentation des entités du monde réel. Nous pouvons par exemple citer : les données manquantes ou incomplètes, les attributs multi-valués (i.e. il y a plusieurs informations pour un attribut) ou les typages différents (i.e. l'information est donnée sous différentes formes). De plus, les nouvelles technologies de l'information ont permis l'arrivée de nouveaux types de documents comme les images, le son, la vidéo ou les documents multimédia. Les usages ont beaucoup évolué, et l'échange de documents numériques est chaque jour plus important. Ainsi, les bases de données s'adaptent en proposant un cadre plus générique cherchant à intégrer des données fortement hétérogènes et pour la plupart non-structurées formant une nouvelle catégorie de données appelée « données complexes ». Les données complexes regroupent des données quantitatives, qualitatives, temporelles, floues ou des données à contenu sémantique.

Au cours de cette thèse nous nous intéressons aux données complexes caractérisées par des schémas ou structures irrégulières, dynamiques ou inconnues ce qui leur vaut le nom de « semi-structurées ». On trouve un exemple de ce type d'information dans les données du Web (*World Wide Web*). Abiteboul, dans [Abi97], présente les caractéristiques des données semi-structurées décrites ci-dessous :

– *la structure est irrégulière :* cette caractéristique est due à l'hétérogénéité des éléments composant les données. Autrement dit, il existe des éléments qui apportent des informations supplémentaires ou bien il existe des éléments manquants. Ces éléments

[1] Système de Gestion de Base de Données Relationnel, en anglais *Relational DataBase Management System (RDBMS)*

peuvent être aussi mono-valués ou multi-valués.

- *la structure est implicite :* dans certains cas, la structure est donnée de façon implicite, donc il faut l'extraire, en analysant et en interprétant les données.
- *la structure est partielle :* il arrive qu'une partie des données soit composée d'informations n'ayant pas une structure définie comme par exemple les images, le texte brut.
- *le typage est irrégulier :* du fait de l'hétérogénéité des données, le typage n'est pas strict.
- *le schéma est lâche :* à la différence des SGBD où s'impose un typage strict, dans les données semi-structurées sont permises des transgressions sur le typage. Elles conduisent à l'altération du schéma.
- *le schéma antérieur ou postérieur :* c'est-à -dire, la notion de schéma peut être antérieure ou postérieure à l'existence des données.
- *le schéma est large :* le schéma est fréquemment très large pour englober toutes les informations des différentes instances des données.
- *le schéma est parfois ignoré :* des requêtes sont posées sur les données semi-structurées pour la recherche de mots ou motifs sans indication précise, en ignorant le schéma.
- *le schéma évolue rapidement :* car les sources de données semi-structurées sont généralement dynamiques.
- *le type des éléments de données est éclectique :* car la structure dépend du point de vue.
- *la distinction entre schéma et donnée est floue :* contrairement aux SGBD traditionnels où la différence entre schéma et données est bien définie, dans les données semi-structurées cette différence devient floue lorsque les schémas se modifient, sont larges ou lorsque les requêtes portent aussi bien sur les données que sur les structures.

Dans la section suivante nous présentons le processus d'extraction d'itemsets fréquents à partir de données transactionnelles. Les concepts définis ici dans le cadre de données classiques, serviront en effet de base pour définir formellement le problème de la recherche de structures fréquentes dans le cas de données complexes dans lequel nous nous situons.

2.2. Fouille de données transactionnelles

2.2.1. Recherche d'itemsets fréquents

La recherche d'itemsets fréquents est décrite de la façon suivante : Soit $\mathcal{I}$ un ensemble d'items. Un ensemble non-vide d'items $X = \{i_1, \ldots, i_k\} \subseteq \mathcal{I}$ est appelé *itemset* ou *k-itemset* s'il contient k items. Une *transaction*[2] sur $\mathcal{I}$ est un couple $T = (tid, I)$ où tid est

[2]Définie comme l'ensemble des items achetés par un client à une même date.

l'identifiant de la transaction et I est un itemset. On dit qu'une transaction $T = (tid, I)$ supporte un itemset $X \subseteq \mathcal{I}$, si $X \subseteq I$. Une *base de données de transactions* $\mathcal{D}$ sur $\mathcal{I}$ est un ensemble de transactions sur $\mathcal{I}$. La *couverture* d'un itemset X dans $\mathcal{D}$, notée $cover(X, \mathcal{D})$, est composée de l'ensemble des identificateurs des transactions de $\mathcal{D}$ qui supportent X, c'est-à-dire $cover(X, \mathcal{D}) = \{tid | (tid, I) \in \mathcal{D}, X \subseteq I\}$. Le support d'un itemset X dans $\mathcal{D}$ correspond au nombre de transactions qui couvrent X dans la base de données $\mathcal{D}$, $support(X, \mathcal{D}) = |cover(X, \mathcal{D})|$. La fréquence d'un itemset X est le rapport du nombre de transactions de la base de données couvrant X au nombre de transactions constituant la base de données. $freq(X, \mathcal{D}) = \frac{support(X,\mathcal{D})}{|\mathcal{D}|}$. Un itemset est appelé *fréquent* si son support est plus grand qu'un support minimal absolu σ (seuil spécifié par l'utilisateur) où $0 \leq \sigma \leq |\mathcal{D}|$. Si on considère les fréquences des itemsets à la place des supports, alors le seuil sera un support minimal relatif σ_{rel} tel que $0 \leq \sigma_{rel} \leq 1$. Étant donné une base de données de transactions $\mathcal{D}$ et un support minimal absolu σ, l'ensemble d'itemsets fréquents dans $\mathcal{D}$ en considérant σ est défini par $\mathcal{F}(\mathcal{D}, \sigma) := \{X \subseteq \mathcal{I} | support(X, \mathcal{D}) \geq \sigma\}$.

Problème 2 (Recherche d'itemsets fréquents) *Étant données un ensemble d'items $\mathcal{I}$, une base de données de transactions $\mathcal{D}$ sur $\mathcal{I}$, et un support minimal σ, le problème consiste à extraire l'ensemble d'itemsets fréquents $\mathcal{F}(\mathcal{D}, \sigma)$, noté simplement $\mathcal{F}$.*

Exemple 2 *Considérons la base de données $\mathcal{D}$ sur l'ensemble $\mathcal{I} = \{a, b, c, d\}$, illustrée par le tableau 2.1. Avec un support minimal absolu établi $\sigma = 1$, le tableau 2.2 montre les itemsets fréquents dont le support est supérieur ou égal à σ. L'espace de recherche de tous les itemsets potentiellement fréquents contient exactement $2^{|\mathcal{I}|}$ itemsets différents. Considérons une approche naïve pour générer et compter le support de tous les itemsets dans une base de données. Il est évident que si $\mathcal{I}$ est suffisamment grand, alors ce programme ne pourra pas finir dans une période de temps raisonnable. Par exemple, soit une base de données transactionnelle sur $\mathcal{I}$ où $|\mathcal{I}| = 50$. Afin d'obtenir l'ensemble d'itemsets fréquents, utilisons un programme basé sur l'algorithme 1 qui a pour but de générer et de valider le support de chaque itemset appartenant à l'ensemble des parties de $\mathcal{I}$. Si le temps d'exécution de l'algorithme 1 pour traiter chaque itemset s'exprime en micro-secondes, alors la fin de cette fonction sera attendue 35.7 ans après le lancement du programme.*

La recherche d'itemsets fréquents est accomplie en deux phases principales : la *génération* et la *validation* des itemsets candidats. La première phase consiste à proposer un ensemble d'itemsets, appelés *candidats*, susceptibles d'être fréquents (cet ensemble étant idéalement le plus petit possible pour éviter des calculs inutiles) ; tandis que la deuxième sert à vérifier le support de ces candidats. Le processus général consiste à générer des candidats de plus en plus détaillés en fonction des fréquents trouvés à l'étape précédente et à répéter étape par étape, les phases de génération/validation jusqu'à ne plus rien pouvoir trouver.

TAB. 2.1 – Exemple d'une base de données de transactions $\mathcal{D}$

tid	I
1	{a, b, d}
2	{a, b}
3	{c, d}
4	{b, c}

TAB. 2.2 – Ensembles d'itemsets de $\mathcal{F}$ et leur supports ($\sigma \geq 1$) dans $\mathcal{D}$

Itemset X	$cover(X, \mathcal{D})$	$support(X, \mathcal{D})$	$freq(X, \mathcal{D})$
$\emptyset$	$\{1, 2, 3, 4\}$	4	100%
$\{a\}$	$\{1, 2\}$	2	50%
$\{b\}$	$\{1, 2, 4\}$	3	75%
$\{c\}$	$\{3, 4\}$	2	50%
$\{d\}$	$\{1, 3\}$	2	50%
$\{a, b\}$	$\{1, 2\}$	2	50%
$\{a, d\}$	$\{1\}$	1	25%
$\{b, c\}$	$\{4\}$	1	25%
$\{b, d\}$	$\{1\}$	1	25%
$\{c, d\}$	$\{3\}$	1	25%
$\{a, b, d\}$	$\{1\}$	1	25%

2.2.2. L'algorithme APRIORI

Afin d'optimiser le processus de recherche d'itemsets fréquents, Agrawal propose l'algorithme APRIORI dans [AS94]. APRIORI (c.f. Algorithme 2) prend comme paramètres une base de données transactionnelle et un seuil pour indiquer un support ou une fréquence minimal σ. Tous les itemsets fréquents sont alors extraits de la base de données. Dans un premier temps, l'algorithme initialise les candidats $\mathcal{C}_1$ avec tous les items dans $\mathcal{I}$ (ligne 2). Ensuite, des phases de validation et génération de candidats sont réalisées par niveau où chaque niveau correspond à une k-ième itération. Pour compter le support de tous les k-itemset candidats, chaque transaction de la base de données est examinée et tous les supports des itemsets inclus dans cette transaction sont augmentés (lignes 6-9). Puis, tous les itemsets candidats dont le support est plus grand que σ sont insérés dans $\mathcal{F}_k$ (ligne 11), ils sont donc fréquents. Avant de recommencer une nouvelle itération, l'algorithme APRIORI-GEN (Algorithme 3) génère les candidats de taille $k + 1$ (ligne 13). Les items appartenant à une transaction sont maintenus ordonnés par ordre lexicographique, afin d'éviter des redondances dans les itemsets candidats produits.

Entrées : $\mathcal{D}$ *//une base de données transactionnelle sur* $\mathcal{I}$,
$0 < \sigma \leq 1$ *//un seuil ou support minimal*
Sorties : $\mathcal{F}$ *//l'ensemble de tous les items fréquents dans* $\mathcal{D}$

```
1 début
2     pour chaque X ∈ 2^I faire
3         si (support(X, D) ≥ σ) alors
4             F ← X;
5     retourner F;
6 fin
```

Algorithme 1 : FREQITEMSET_NAIVE

Dans APRIORIGEN, la notation $X[i]$ représente le $i$ème item dans un itemset X. Le *k-préfixe* d'un itemset X correspond à l'itemset $\{X[1], X[2], \ldots, X[k]\}$. Cet algorithme est constitué de deux étapes : la première appelée étape de *jointure*, réalise la jointure de $\mathcal{F}_k$ avec lui même pour générer les candidats (ligne 2). Considérant les itemsets $X, Y \in \mathcal{F}_k$, l'union de $X \cup Y$ est réalisée seulement s'ils ont un $(k-1)$-préfixe commun et si l'item $X[k]$ est inférieur à celui de $Y[k]$ (lignes 4-5). La deuxième étape, appelée *élagage*(lignes 7-8), sert à réduire le nombre d'itemsets candidats pour l'étape de validation. Elle s'appuie sur la propriété d'anti-monotonie (c.f. Propriété 1) qui permet d'éliminer tous les sur-ensembles d'un itemset infréquent. L'union $X \cup Y$ est insérée dans les candidats $\mathcal{C}_{k+1}$ seulement si tous ses k-subsets sont inclus dans $\mathcal{F}_k$.

Propriété 1 (anti-monotonie) *Étant donné une base de données transactionnelle sur* $\mathcal{I}$, *et soient* $X, Y \subseteq \mathcal{I}$ *deux itemsets, alors,* $X \subseteq Y \rightarrow support(Y, \mathcal{D}) \leq support(X, \mathcal{D})$, *par conséquent, si un itemset est infréquent, alors tous ses sur-ensembles sont infréquents.*

Exemple 3 *Considérons la base de données* $\mathcal{D}$ *illustrée par le tableau 2.1 et un support minimal absolu* $\sigma = 1$. *Si on exécute l'algorithme* APRIORI *sur les paramètres précédents, on obtiendra le treillis illustré par la figure 2.1. Les itemsets infréquents sont entourés d'un carré en pointillé. Pour la première itération, on trouve que tous les itemsets* $\mathcal{F}_1$ *sont fréquents. Dans l'itération* 2, *après le comptage du support de l'itemset,* $\{a, c\}$ *ne peut pas être validé comme itemset fréquent. À l'étape* 3, *par la propriété d'anti-monotonie, l'algorithme* APRIORIGEN *dans l'étape d'élagage n'inclut pas les itemsets* $\{a, b, c\}$ *et* $\{a, c, d\}$ *car ils sont des sur-ensembles de l'itemset infréquent* $\{a, c\}$. *L'item set* $\{b, c, d\}$ *est déclaré infréquent après la validation de son support. Dans la dernière étape, le seul itemset de taille* 4 *est marqué infréquent car il est un sur-ensemble de trois itemsets de l'étape précédente.*

Entrées : $\mathcal{D}$ *//une base de données transactionnelle,*
σ *//un support minimal*
Sorties : $\mathcal{F}$, *//un ensemble de itemset fréquents*

```
1  début
2      C_1 ← {{i}|i ∈ I};
3      k ← 1;
4      tant que (C_k ≠ ∅) faire
5          // Calcule le support de tous les itemsets candidats;
6          pour chaque transaction (tid, I) ∈ D faire
7              pour chaque candidat C ∈ C_k faire
8                  si (C ⊆ I) alors
9                      C.support + +;
10         // Extraire tous les itemsets fréquents;
11         F_k ← {C|C.support ≥ σ};
12         // Nouveaux k-candidats;
13         C_{k+1} ← AprioriGen(F_k) ;
14         + + k;
15     retourner F = ⋃_k F_k;
16 fin
```

Algorithme 2 : Apriori

Entrées : $\mathcal{F}_k$ *//l'ensemble des k-itemsets fréquents*
Sorties : $\mathcal{C}_{k+1}$ *//l'ensemble des $k+1$-itemsets candidats*

```
1  début
2      pour chaque (X, Y ∈ F_k) faire
3          //Étape de jointure;
4          si (X[i] = Y[i], pour 1 ≤ i ≤ k − 1 and X[k] < Y[k]) alors
5              I ← X ∪ {Y[k]};
6              //Étape d'élagage;
7              si ∀J ⊂ I, |J| = k : J ∈ F_k alors
8                  C_{k+1} ← C_{k+1} ∪ I
9      retourner C_{k+1};
10 fin
```

Algorithme 3 : AprioriGen

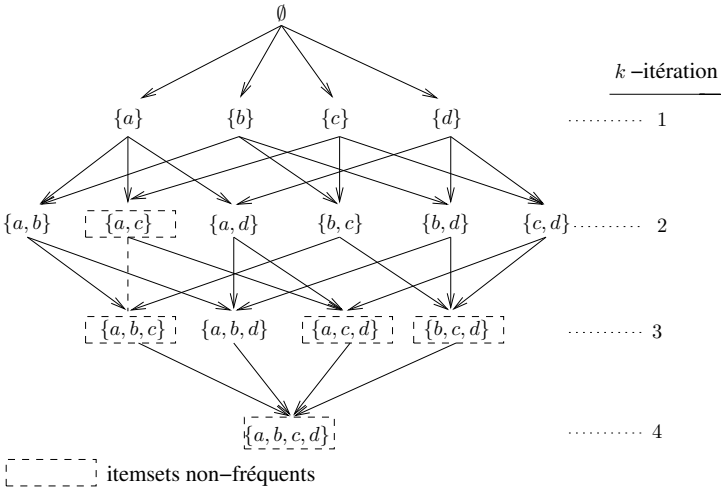


FIG. 2.1 – Treillis des itemsets générés et validés par étapes par l'algorithme APRIORI

2.2.3. Recherche de règles d'association

Situées dans le domaine des grandes surfaces, les règles d'association étudient le comportement des consommateurs en permettant de découvrir des corrélations entre divers items. La recherche de ces règles s'applique à des données de type *itemset* en permettant l'extraction de relations existantes à l'intérieur des transactions. Une règle d'association alors peut être vue comme une relation causale entre deux ensembles d'itemsets.

Une règle d'association est une implication de la forme $X \rightarrow Y$, où $X \subseteq \mathcal{I}$, $Y \subseteq \mathcal{I}$ et $X \cap Y = \emptyset$. Une telle règle exprime l'association suivante : si une transaction contient tous les items dans X alors cette transaction contient aussi tous les items dans Y. Le support d'une règle d'association $X \rightarrow Y$ dans $\mathcal{D}$ est le support de $X \cup Y$ dans $\mathcal{D}$. D'une manière similaire, la fréquence d'une règle correspond à la fréquence de $X \cup Y$. Une règle est dite fréquente si son support est supérieur à un seuil de support minimal établi par l'utilisateur. La *confiance* d'une règle d'association $X \rightarrow Y$ représente la probabilité conditionnelle d'avoir un itemset Y inclus dans une transaction qui contient un itemset X. Cette confiance est définie par : $conf(X \rightarrow Y, \mathcal{D}) = \frac{support(X \rightarrow Y, \mathcal{D})}{support(X, \mathcal{D})}$. Une règle est fiable si sa confiance dépasse un seuil ς de *confiance minimale* où $0 \leq \varsigma \leq 1$.

Problème 3 (Recherche de règles d'association) *Étant donnés un ensemble d'items $\mathcal{I}$, un base de données de transactions $\mathcal{D}$ sur $\mathcal{I}$, un support minimal σ et une confiance minimale ς, le problème consiste à extraire l'ensemble des règles fréquentes et fiables $\mathcal{R}(\mathcal{D}, \sigma, \varsigma)$ défini par : $\mathcal{R}(\mathcal{D}, \sigma, \varsigma) = \{X \rightarrow Y | X, Y \subseteq \mathcal{I}, X \cap Y = \emptyset, X \cup Y \in \mathcal{F}(\mathcal{D}, \sigma), conf(X \rightarrow Y, \mathcal{D}) \geq \varsigma\}$.*

Le processus d'extraction de règles d'association présenté dans [AIS93] se fait en deux

TAB. 2.3 – Ensemble d'itemsets de $\mathcal{F}$ et leurs supports ($\sigma \geq 1$) dans $\mathcal{D}$

Règles	Cover	Support	Fréquence	Confiance
$\{a\} \rightarrow \{b\}$	$\{1,2\}$	2	50%	100%
$\{b\} \rightarrow \{a\}$	$\{1,2\}$	2	50%	66%

étapes :

- Dans une première étape on cherche tous les itemsets fréquents. Ces itemsets correspondent aux règles du type $X \rightarrow \emptyset$.
- La deuxième étape consiste à générer des règles, à partir des itemsets fréquents de l'étape précédente, dont la confiance est supérieure à ς.

Exemple 4 *Pour illustrer l'extraction de règles d'association, considérons la bases de données $\mathcal{D}$ du tableau 2.1, un support minimal $\sigma = 2$ et une confiance minimale $\varsigma = 0.6$. Les seules règles extraites sont indiquées dans le tableau 2.3. Ces règles sont supportées par 2 transactions sur 4 dans $\mathcal{D}$. En revanche, il existe une différence entre la confiance des règles que peut être interprétée de la façon suivante : Pour la première $\{a\} \rightarrow \{b\}$ nous avons une confiance de 100% : un client qui achète l'item a va acheter l'item b en même temps. La règle $\{b\} \rightarrow \{a\}$ a un confiance de 66% : seulement 2 des 3 clients achetant l'item b vont acheter l'item a en même temps.*

2.3. Fouille de données semi-structurées

Une des caractéristiques principales des données semi-structurées est la flexibilité pour les données dont la structure peut : évoluer, contenir des informations multiples, des informations incomplètes ou bien appartenir à des types de données différents. Cette structure imprécise est implicitement définie dans les données elles-mêmes et elle est représentée par des arborescences (i.e. des arbres enracinés, ordonnés et étiquetés). L'absence d'une structure pré-définie ainsi que la structure arborescente des données semi-structurées rendent la manipulation de ces données difficile et complexe.

Le langage de balisage extensible XML[3] est l'une des façons de représenter les données semi-structurées et est maintenant devenu le format universel pour l'échange d'informations sur le Web. Comme son nom le suggère, XML est un langage balisé constitué d'éléments, d'attributs et de contenu. Les valeurs sont encadrées par deux balises. Les balises servent à décrire l'information et forment la structure de la donnée.

[3]de l'anglais *eXtensible Markup Language*

Les données semi-structurées sont représentées sous la forme de graphes ou d'arbres [Bun97]. Malgré la présence de cycles dans les données semi-structurées, nous nous référerons généralement à ces données en tant qu'arbres. Du fait de la nature arborescente des données, par la suite nous traitons les bases de données semi-structurées comme des bases de données constituées d'arborescences.

Dans la section suivante nous présentons des définitions et des propriétés utiles pour le traitement des arbres, qui seront employées tout au long de ce mémoire. Pour une référence complète on peut se reporter aux ouvrages [Knu73, Val02].

2.3.1. Définition et propriétés des arbres

Un *arbre* est un modèle abstrait d'une structure hiérarchique imposée par l'existence d'un chemin unique de n'importe quel nœud à la racine. Dans cette structure, la racine est placée en haut de l'arbre et les autres nœuds sont organisés par des niveaux conformément à leur distance par rapport à la racine [Val02]. Un *arbre libre* est un graphe non-orienté, connexe et sans cycle, noté $T = (V, E)$ où V est un ensemble de nœuds et E est un ensemble d'arcs ou d'arêtes. Si on permet à un nœud $u \in V$ d'être distingué et qu'on l'appelle la *racine* de T, alors on dira que T est une *arborescence*. Une arborescence T est définie par la triplet $T = (V, E, r)$ où :

- E décrit la relation binaire de *parenté* entre les nœuds de T, c'est-à-dire, chaque arête $(u, v) \in E$ est une relation *(père, fils)*,
- La racine ne possède pas de père, $\nexists (u, r) \in E$ tel que $u, r \in V$,
- Chaque nœud $u \in V \setminus \{r\}$ a un seul père, et
- Il existe un seul chemin $[r, \ldots, v]$ pour tout $v \in V$.

La racine d'une arborescence T est également notée $root(T)$. Étant donné un arbre T à k nœuds (on parle alors d'un k-arbre ou d'un arbre sur $[k]$) sa taille est donnée par $|T| = |V|$, également notée $size(T) = |V|$.

Exemple 5 *La figure 2.2 illustre un arbre libre transformé en arborescence (c.f. Figure 2.3) après avoir choisi le nœud* 1 *comme la racine de* T.

Si un chemin $[r, \ldots, u]$, de la racine à un nœud $u \in V$, est formé de k arcs, alors la *profondeur* de u est égale à k et elle notée par $depth(u) = k$. La profondeur d'un arbre T est donnée par le maximum des profondeurs de ses sommets. Formellement, $depth(T) = max\{depth(u) | u \in V\}$.

Relations de parenté et d'ancestralité

Étant donné un arbre $T = (V, E, r)$, on peut définir les relations suivantes en fonction du chemin unique $[r, \ldots, u]$, pour tout $u \in V$:

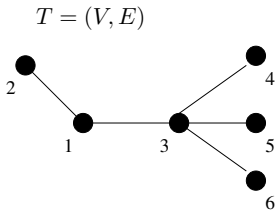


FIG. 2.2 – Arbre libre

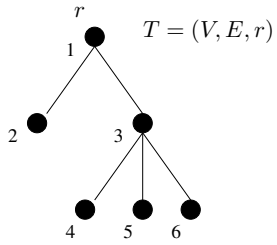


FIG. 2.3 – Arborescence

– Le *père* d'un nœud $u \in V \setminus \{r\}$, noté $parent(u)$, est le nœud adjacent à u dans le chemin $[r, \ldots, u]$ tel que $parent(u) = \{v \in V | (v, u) \in E\}$. Le père de la racine est indéfini, donc $parent(r) = \emptyset$.

– Si pour une arête $(v, u) \in E$, on dit que v est le père de u, alors u est le *fils* de v. L'ensemble de fils d'un nœud $v \in V$ est défini par $children(v) = \{u \in V | (v, u) \in E\}$. Si $children(u) = \emptyset$ alors le nœud u est nommé *feuille* sinon il est appelé *nœud interne*.

– Les relations d'ancestralité dans un arbre T sont encodées par la fermeture transitive E^+ de ses arêtes :

$$\begin{aligned} E^0 &= \{(u, u) | u \in V\}, \\ E^{n+1} &= \{(u, w) | \exists v \in V \text{ tel que } (u, v) \in E \wedge (v, w) \in E^n\}, \text{ pour } n \geq 0, \\ E^{\leq i} &= \bigcup_{n>0}^{i} E^n, \\ E^+ &= \bigcup_{n>0} E^n \end{aligned}$$

Les *ancêtres* d'un nœud $u \in V \setminus \{r\}$, sont tous les nœuds du chemin $[r, \ldots, u]$ sauf u. L'ensemble des ancêtres de u est défini par la fonction $ancestors(u) = \{v \in V | (v, u) \in E^+\}$.

– Si v est un ancêtre de u, alors on dit que u est un *descendant* de v. L'ensemble des descendants de v est défini par la fonction $descendants(v) = \{u \in V | (v, u) \in E^+\}$.

De la même façon nous pouvons définir les ancêtres (resp. les descendants) qui sont placés à une distance i d'un nœud par $ancestors^i(u) = \{v \in V | (v, u) \in E^i\}$ (resp. $descendants^i(v) = \{u \in V | (v, u) \in E^i\}$). Ou bien, si nous voulons récupérer l'ensemble de ancêtres (resp. descendants) qui se trouvent à une distance maximale i, on utilise $ancestors^{\leq i}(u) = \{v \in V | (v, u) \in E^{\leq i}\}$ (resp. $descendants^{\leq i}(v) = \{u \in V | (v, u) \in E^{\leq i}\}$).

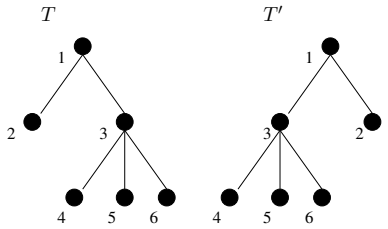


FIG. 2.4 – Des arbres ordonnés ou non-ordonnés

Relations d'ordre

Considérons un nœud u dans l'arbre $T = (V, E, r)$, alors on dit que le *sous-arbre* de T enraciné au nœud u est défini par l'arbre $T[u] = (V', E', u)$, tel que $V' = \{u\} \cup descendants(u)$ et $E' = V' \times V'$. Avec cette notion de sous-arbre, il est possible de donner une définition récursive d'un arbre plus en accord avec sa structure imbriquée. Alors un arbre est composé de :

- un nœud distingué et appelé racine $root(T)$
- une séquence finie et ordonnée d'arbres disjoints est appelée *forêt* et notée $< T_1, ..., T_n >$, où $n \geq 0$. Ces sous-arbres de $root(T)$ son appelés les sous-arbres *immédiats* de l'arbre T.

Un arbre est dit *ordonné* si on établit une relation d'ordre partielle $\preceq$ entre les fils d'un nœud $u \in V$ tels qu'ils peuvent être ordonnés du *premier* au *dernier* fils. Autrement dit, il existe un ordre relatif $\preceq$ entre les sous-arbres $T_1, ..., T_n$ composant un arbre T. Considérant cette relation d'ordre, un *arbre ordonné* est défini par $T = (V, E, r, \preceq)$.

Exemple 6 *Si on considère que T et T' dans la figure 2.4 sont des arbres ordonnés, alors $T \neq T'$. En revanche, si T et T' sont des arbres non-ordonnés alors $T = T'$.*

Relations de fratrie

Étant donné un arbre ordonné $T = (V, E, r, \preceq)$, les nœuds $w \in V$ et $u, v, y \in V \setminus \{r\}$, et les relations de parenté $(w, u), (w, v), (w, y) \in E$, il est possible de définir les fonctions suivantes pour établir une distinction entre les fils de w :

- Les nœuds $u, v \in V \setminus \{r\}$ sont *frères* si $parent(u) = parent(v)$. L'ensemble des frères d'un nœud, noté $siblings(u)$, est défini par $siblings(u) = \{v \in V | parent(u) = parent(v)\}$.
- Le fils le plus à gauche de w est défini par $first(w) = \{u | \nexists v, (w, v) \in E \wedge v \preceq u\}$.
- Le fils le plus à droite de w est défini par $last(w) = \{v | \nexists u, (w, u) \in E \wedge v \preceq u\}$.

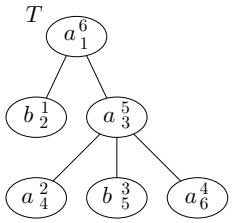


FIG. 2.5 – Arbre étiqueté T sur $\Sigma = \{a, b\}$

- Le *frère suivant* de u, est défini par $next(u) = \{v | (w, v) \in E \wedge u \preceq v \wedge \nexists y$ tel que $(w, y) \in E \wedge u \preceq y \preceq v\}$.
- Le nœud u est le *frère antérieur* de v, noté $previous(v)$, si $next(u) = \{v\}$.
- La *branche la plus à gauche* de T, notée $lmb(T)$, correspond au chemin $[v_1, v_2, \ldots, v_k]$, tel que $v_1 = r$, et $first(v_{i-1}) = \{v_i\}, \forall i, 2 \leq i \leq k$ et que $deg(v_k) = 1$.
- Le nœud terminal v_k dans $lmb(T)$ est appelé la *feuille la plus à gauche* et est noté $lml(T)$.
- La *branche la plus à droite* de T, notée $rmb(T)$, correspond au chemin $[v_1, v_2..., v_k]$, tel que $v_1 = r$ et que $last(v_{i-1}) = \{v_i\}, \forall i, 2 \leq i \leq k$ et $deg(v_k) = 1$.
- Le nœud terminal v_k dans $rmb(T)$ est appelé la *feuille la plus à droite* et est noté $rml(T)$.

Des étiquettes

On considère qu'un arbre T est étiqueté si tout nœud à l'intérieur de T est muni d'un étiquette appartenant à un ensemble fini d'étiquettes appelé *alphabet*. Nous autorisons deux nœuds à avoir la même étiquette. Les étiquettes (des nœuds) sont indiquées à l'intérieur de chaque nœud. Étant donné un alphabet $\Sigma = \{a, b, c, ...\}$, un arbre est dit *étiqueté* si à chaque nœud $u \in V$ correspond une étiquette $l \in \Sigma$. Il est défini par $T = (V, E, r, \lambda)$ où $\lambda : V \rightarrow \Sigma$ tel que $\lambda(u) = l$.

Représentation récursive des arbres

Un arbre peut être défini de façon récursive en fonction de ses sous-arbres composants. On dit qu'un arbre est un ensemble fini de nœuds si : 1) il existe un nœud distingué appelé *racine* et 2) les nœuds restants sont partitionnés en ensembles qui sont eux mêmes des arbres. La définition antérieure nous permet de faire la représentation d'un arbre à l'aide des parenthèses imbriquées. Considérons un arbre T avec ses sous-arbres immédiats $< T_1, T_2, \ldots, T_n > n \geq 0$ et l'étiquette de la racine $\lambda(root(T)) = l$, alors l'arbre T est représenté par $T = (l(T_1) \ldots (T_n))$. Pour simplifier la notation, les parenthèses sont enlevées après l lorsque $n = 0$.

Exemple 7 *La figure 2.5 illustre un exemple d'un arbre étiqueté qui est représenté par* $T = (a(b)(a(a)(b)(a)))$.

Parcours des arbres

Un processus pour visiter les nœuds dans un certain ordre est appelé une *traversée*. Une traversée qui inscrit chaque nœud dans l'arbre une fois exactement est une *énumération*. Les nœuds des arbres peuvent être énumérés en parcourant l'arbre en profondeur ou en largeur. La première méthode permet de parcourir l'arbre de sous-arbre en sous-arbre, tandis que la deuxième, parcourt l'arbre de niveau en niveau [A.S01b]. Le *parcours* d'un arbre est une façon d'ordonner ses nœuds afin de le parcourir. Cela consiste à visiter les nœuds d'un arbre en leur assignant un numéro afin de déterminer leur position dans l'arbre. On parcourt un arbre $T = (V, E, r)$ sur n nœuds en établissant une bijection d'ordre : $V \to \{1, ..., k\}$, de telle sorte que l'ordre pour le premier nœud u soit $ordre[u] = 1$, pour le deuxième nœud v, $ordre[v] = 2$, et on continue jusqu'au dernier nœud w avec un $ordre[w] = k$. Parmi les parcours en profondeur, on distingue à nouveau le parcours préfixe et le parcours suffixe.

Étant donné un arbre $T = (V, E, r)$ sur k nœuds, le *parcours préfixe* (également appelé en pré-ordre) est une bijection d'ordre $pre(T) : V \to \{1, \ldots, k\}$ telle que pour tout $u \in V$:

- $pre(r) = 1$
- si u n'est pas une feuille
  - $pre(first(u)) = pre(u) + 1$
- si u n'est pas le dernier fils
  - $pre(next(u)) = pre(u) + size(T[u])$

D'autre part, le *parcours postfixe* (parfois appelé en post-ordre) est une bijection d'ordre $post(T) : V \to \{1, \ldots, k\}$ telle que pour tout $u \in V$:

- $post(r) = k$
- si u n'est pas une feuille
  - $post(last(u)) = post(u) - 1$
- si u n'est pas le premier fils
  - $post(previous(u)) = post(u) - size(T[u])$

Exemple 8 *Reprenant l'arbre de la figure 2.5, les indices à l'intérieur des nœuds (resp. exposants) indiquent la numérotation des nœuds en parcourant* T *en pré-ordre (resp. post-ordre).*

Notation : Désormais nous utiliserons le mot *arbre ou arborescence* pour se référer à un arbre enraciné, ordonné, étiqueté et numéroté en profondeur d'abord. Afin de clarifier le contexte au moment de traiter plusieurs arbres, pour un arbre $T = (V, E, r, \lambda, \preceq)$ nous noterons V_T, E_T, r_T, λ_T et $\preceq_T$ chaque composant de T.

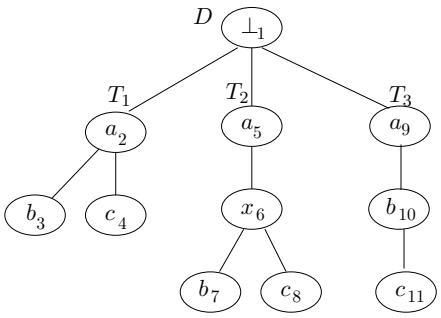


FIG. 2.6 – Une base de données arborescente $\mathcal{D}$

Base de données arborescentes

Nous définissons une *base de données arborescente* dorénavant appelée simplement base de données (c.f. Figure 2.6), comme une *forêt* composée d'un ensemble de $n > 1$ arbres $\mathcal{T} = \{T_1, ..., T_n\}$, constituant un graphe non orienté, sans cycle et non connexe. Cette forêt est groupée dans une structure notée $\mathcal{D}$ et définie par le triplet $\mathcal{D} = (D, \Sigma_\mathcal{D}, \lambda_\mathcal{D})$ où :

- D est un arbre ordonné et étiqueté sur un alphabet $\Sigma = \{a, b, c, ...\}$ et défini par $D = (V, E, r, \preceq, \lambda)$. La racine r_D est étiquetée avec le symbole $\perp$ qui n'appartient pas à Σ. Cet arbre est appelé *arbre de données*. Chaque sous-arbre enraciné au i-ème fils de r_D, représente un arbre T_i inclus dans la base de données.
- $\Sigma_\mathcal{D} = \{t_1, t_2, ..., t_n\}$ est un ensemble d'identifiants tels que pour tout $i \in [1, n]$ on associe l'arbre T_i à l'identifiant t_i.
- $\lambda_\mathcal{D} : V_D \setminus \{r_D\} \rightarrow \Sigma_\mathcal{D}$ est une fonction qui identifie l'arbre auquel appartient un nœud dans l'arbre de données (à exception de sa racine), c'est-à -dire qu'elle satisfait $\lambda_\mathcal{D}(u) = t_i$ pour tout $u \in V_{T_i}$.

La taille de la base de données est défini par $|\mathcal{D}| = |\Sigma_\mathcal{D}|$ et le nombre de nœuds à l'intérieur de la base est donné par $\| \mathcal{D} \| = |V_D| - 1 = \sum_{i=1}^{n} |V_{D_i}|$, pour $n \geq 1$.

2.3.2. L'algorithme APRIORI sur les arbres

Nous avons vu dans la section 2.2.2 que l'algorithme APRIORI est composé de deux phases principales : 1) La génération de candidats et 2) la validation des candidats. L'utilisation de cette approche pour l'extraction de motifs dans les bases de données arborescentes (de données complexes) implique l'adaptation de ces étapes pour remplacer les données transactionnelles par des données arborescentes. D'abord nous présentons les modifications faites à l'étape de génération de candidats.

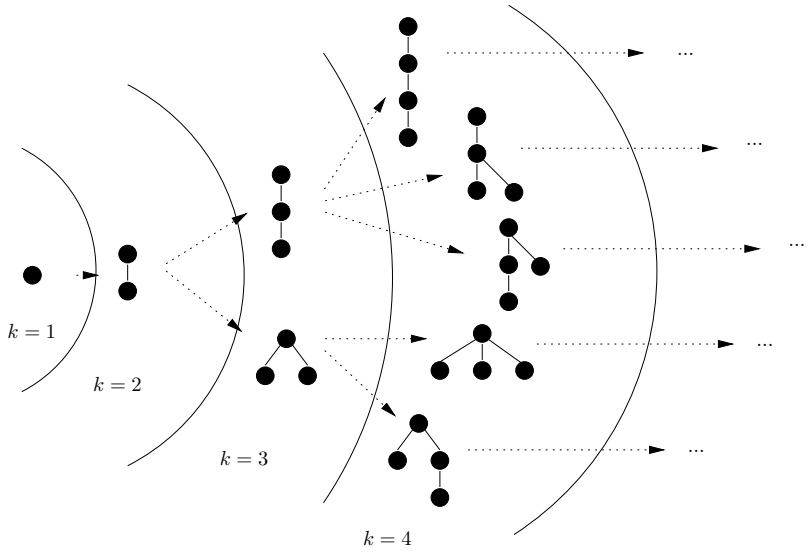


FIG. 2.7 – Énumération d'arbres enracinés, non-étiquetés et ordonnés ayant k nœuds

2.3.3. Génération d'arbres candidats

Dans cette étape, il faut d'abord penser à une méthode de génération d'arborescences selon un paramètre spécial, à savoir k le nombre de nœuds de l'arborescence. La méthode doit accomplir les conditions suivantes :

- Chaque arbre doit être obtenu de manière unique à partir d'un autre arbre de taille plus petite.
- Il doit exister un unique arbre de taille minimale.

Exemple 9 *Si nous considérons des arborescences ordonnées et non-étiquetées, la figure 2.7 illustre la génération d'arborescences ayant* $1, 2, 3$ *et* 4 *nœuds. Il est facile de constater que les deux conditions mentionnées ci-dessus sont respectées. Premièrement, nous avons que l'arbre à* 1 *nœud produit l'arbre à* 2 *nœuds, puis ce dernier produit les arbres à* 3 *nœuds, puis chaque arbre à* 3 *nœuds génère des arbres à* 4 *nœuds, etc. Deuxièmement, chaque arbre est généré à partir d'un seul arbre.*

Le nombre total d'arbres à k nœuds est donné par les nombres de Catalan[SF01] $C_n = \binom{2n}{n} - \binom{2n}{n-1}$. Les valeurs des premiers nombres de Catalan $k = 1, 2, 3, 4, 5, 6, 7, 8, 9, 10\ldots$ sont $1, 1, 2, 5, 14, 42, 132, 429, 1430, 4862, \ldots$, donc le nombre total d'arborescences à 5 nœuds générées par les arbres à 4 nœuds doit être égal à 14. Dans la suite, pour introduire l'énumération d'arborescences ayant k nœuds, nous nous appuyons sur la technique d'énumération d'objets combinatoires, appelée ECO [4] introduite par Barcucci dans [BLPP99].

[4] de l'anglais *Enumerating Combinatorial Objects*

Problème 4 (L'énumération d'arborescences) *Étant donné un ensemble $\mathcal{T}$ d'arbres et un paramètre q, appelé la* taille *d'un arbre, nous nous intéressons à l'ensemble d'arbres $\mathcal{T}_k$ pour lesquels la valeur du paramètre est égale à $k \in \mathbb{N}$, c'est-à-dire $\mathcal{T}_k = \{T \in \mathcal{T} : q(T) = k\}$. Le problème consiste à énumérer chaque arbre $T' \in \mathcal{T}_{k+1}$ (les arbres de taille $k+1$), à partir d'un arbre unique $T \in \mathcal{T}_k$.*

Considérons maintenant un opérateur ϑ tel que :

$$\vartheta : \mathcal{T} \to 2^{\mathcal{T}} \text{ tel que } \vartheta(\mathcal{T}_k) \subseteq 2^{\mathcal{T}_{k+1}}$$

où, pour un ensemble $\mathcal{T}$, $2^{\mathcal{T}}$ correspond à l'ensemble des parties de $\mathcal{T}$.

L'idée principale de la méthode ECO consiste à utiliser un opérateur ϑ pour construire chaque arbre $T' \in \mathcal{T}_{k+1}$ à partir d'un arbre unique $T \in \mathcal{T}_k$. Autrement dit, on construit chaque arbre $T' \in \mathcal{T}_{k+1}$ en faisant localement grandir un arbre $T \in \mathcal{T}_k$. L'opérateur ϑ doit accomplir les contraintes suivantes afin de pouvoir définir des objets d'une manière non-ambiguë :

Proposition 1 *Si pour $k \geq 0$, ϑ satisfait :*

1. *pour chaque $T' \in \mathcal{T}_{k+1}$, $\exists T \in \mathcal{T}_k$, tel que $T' \in \vartheta(T)$, et*
2. *pour chaque $T, T' \in \mathcal{T}_k, \vartheta(T) \cap \vartheta(T') = \emptyset$ chaque fois que $T \neq T'$,*

alors la famille d'ensembles $\Pi_{k+1} = \{\vartheta(T) : T \in \mathcal{T}_k\}$ est une partition de $\mathcal{T}_{k+1}$.

Pour définir l'opérateur $\vartheta : \mathcal{T} \to 2^{\mathcal{T}}$, nous considérons la méthode d'énumération d'arbres proposée par Nakano dans [Nak02] et utilisée par Asai dans [AAK+02]. Avec cette technique nous obtenons $rmb(T)$, la branche la plus à droite d'un arbre $T \in \mathcal{T}_k$, et à chaque nœud appartenant à cette branche, on ajoute un nouveau fils (placé à la position la plus à droite).

Exemple 10 *La figure 2.8 montre l'extension de ϑ sur un arbre T de taille 3. On ajoute un nouveau nœud à la feuille la plus à droite de l'arbre T générant l'arbre T'. On répète ensuite la même opération pour chaque membre de $rmb(T)$. Dans ce cas en particulier, nous obtenons l'arbre T''.*

Afin de réaliser l'insertion de nouveaux nœuds dans un arbre nous définissons un opérateur d'insertion suivante : Étant donnée une position p en pré-ordre d'un nœud $u \in V$ tel que $pre(u) = p$, et un arbre $T = (V, E, r, \preceq) \in \mathcal{T}_k$, on crée un arbre $T' = (V', E', r', \preceq') \in \mathcal{T}_{k+1}$ avec l'opérateur $\oplus$ de la manière suivante :

$$T' = T \oplus p$$

où p est la position du nœud auquel un nouveau nœud v sera ajouté (à la position la plus à droite). Pour l'arbre généré T' nous avons que $V' = V \cup \{v\}$, $E' = E \cup \{(u, v)\}$ et que la position en pré-ordre du nouveau nœud dans T' est $pre(v) = k + 1$.

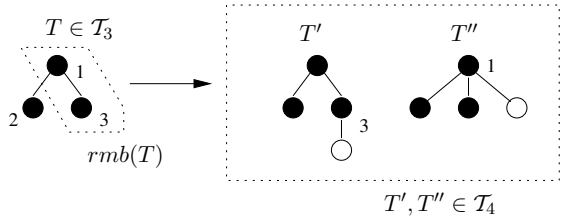


FIG. 2.8 – L'extension d'un arbre T de taille 3

Entrées : $\mathcal{T}_k$ *//l'ensemble d'arbres ordonnés à k nœuds*
Sorties : $\mathcal{T}_{k+1}$ *//l'ensemble d'arbres ordonnés à* $k+1$ *nœuds*

```
1 début
2 |  T_{k+1} ← ∅;
3 |  pour chaque arbre T ∈ T_k faire
4 |  |  pour chaque position p ∈ rmb(T) faire
5 |  |  |  T_{k+1} ← ⋃ T ⊕ p;
6 |  retourner T_{k+1};
7 fin
```

Algorithme 4 : ENUMTREES

Lemme 1 *Étant donné un arbre T, l'ensemble de positions valables p auxquelles on peut ajouter un nouveau nœud est donné par l'ensemble des positions des nœuds appartenant à la branche la plus à droite de T.*

Exemple 11 *La figure 2.8 illustre aussi la propriété précédente. Les arbres T' et T'' sont créés par l'addition d'un nœud aux positions* 3 *et* 1 *(des nœuds appartenant à $rmb(T)$).*

La méthode d'énumération proposée par Nakano nous permet de définir l'opérateur ϑ de la façon suivante :

Théorème 1 *L'image $\vartheta(T)$ d'un arbre $T \in \mathcal{T}_k$ est l'ensemble d'arbres dans $\mathcal{T}_{k+1}$ qui peut être obtenu en ajoutant un fils (le plus à droite) à chaque nœud appartenant à la branche la plus à droite de l'arbre T.*

Une fois défini l'opérateur ϑ, nous pouvons écrire l'algorithme ENUMTREES servant à l'énumération d'arborescences de taille $k+1$ à partir des arborescences de taille k.

Exemple 12 *Dans la figure 2.9 nous donnons un exemple de la génération d'arbres à* $1, 2, \ldots, 5$ *nœuds, Au sommet on ne trouve qu'un seul arbre composé d'un seul nœud. Si on donne comme entrée cet arbre initial à l'algorithme 4, on obtient* $(1)^5$ *seul arbre à* 2 *nœuds.*

[5] On met entre parenthèses le nombre d'arbres générés, pour le différencier du nombre de nœuds

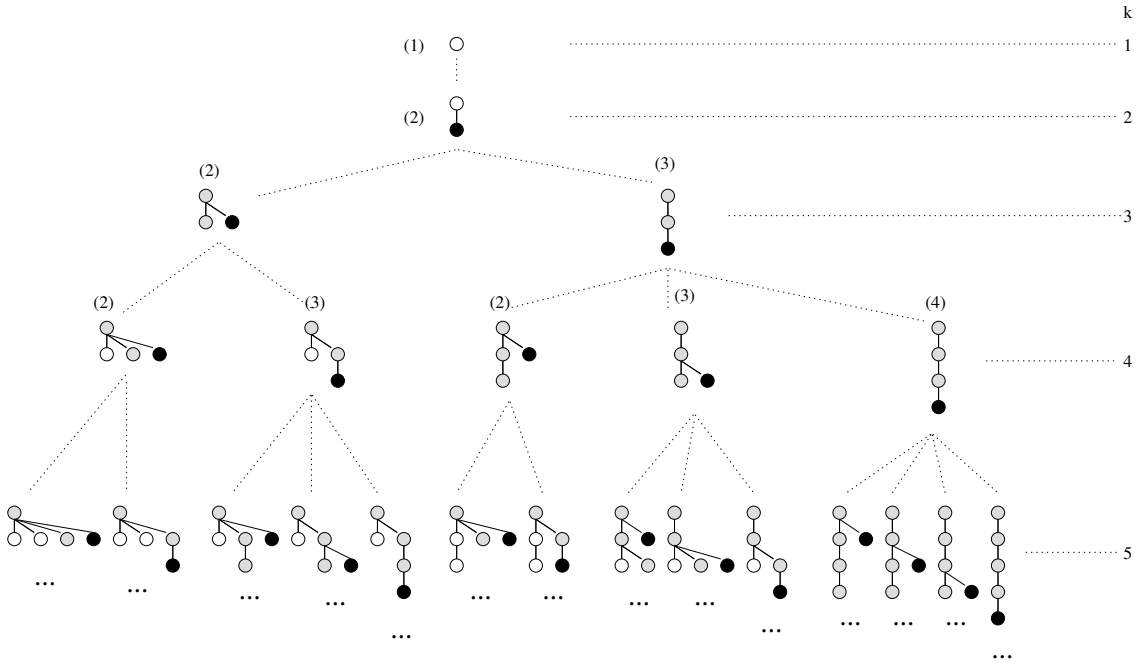


FIG. 2.9 – Énumération des arbres ayant k nœuds, par la méthode de la branche la plus à droite.

On fait grandir cet arbre en ajoutant un nouveau fils (les nœuds colorés noir) à la branche la plus à droite (les nœuds colorés gris) et on obtient (2) *nouveaux arbres. Au niveau* 3*, chaque arbre à son tour génère* (2) *et* (3) *arbres respectivement (par la branche la plus à droite de l'arbre de l'étape précédente)...*

Corollaire 1 *L'opérateur $\vartheta(T)$ énumère : 1) sans omission et 2) sans répétition, toutes les arborescences possibles $T' \in \mathcal{T}_{k+1}$.*

Pour la première contrainte *énumérer sans omission*, considérons les règles de succession introduites par [CGHK78] et utilisées par [Wes96] dans l'étude des permutations de Baxter. Une règle de succession Ω est un système $((a), \mathcal{P})$, composé par un axiome (a) et un ensemble $\mathcal{P}$ de productions définies sur un ensemble d'étiquettes $M \subset \mathbb{N}$ [Dea03].

$$\Omega = \begin{cases} (a), \text{ tel que } a \in \mathbb{N} \\ (k) \rightsquigarrow (e_1(k))(e_2(k)) \dots (e_k(k)), \text{ pour tout } k \in M \end{cases}$$

Ces règles de succession sont représentées convenablement par des *arbres de génération*. Un arbre de génération est un arbre enraciné et étiqueté qui possède la propriété suivante : les étiquettes d'un ensemble de fils d'un nœud u peuvent être déterminées à partir de

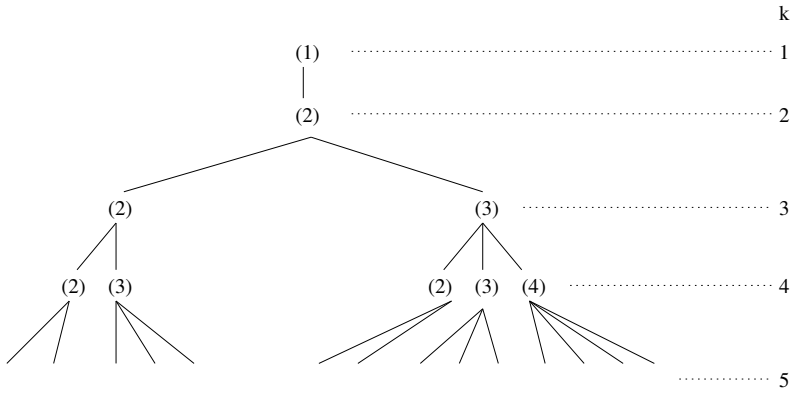


FIG. 2.10 – Arbre de génération décrivant la règle de succession Υ

l'étiquette du nœud u [Wes96]. Ces arbres sont construits de la façon suivante : la racine de l'arbre correspond à l'axiome (a), puis, récursivement, chaque nœud étiqueté k produit le niveau suivant de l'arbre composé par k nœuds étiquetés $(e_1(k))(e_2(k))\dots(e_k(k))$ respectivement.

Exemple 13 *Considérons par exemple la règle de succession Υ définissant les nombres de Catalan :*

$$\Upsilon = \begin{cases} (1) \\ (1) \rightsquigarrow (2) \\ (k) \rightsquigarrow (2)...(k)(k+1) \end{cases}$$

Cette règle est représentée par l'arbre de génération illustré par la figure 2.10. Au niveau 1, la racine de l'arbre est initialisée à 1. Puis au niveau suivant et selon la production $(1) \rightsquigarrow (2)$, le seul nœud du niveau 2 est étiqueté 2. Pour les niveaux suivants, chaque nœud étiqueté k, crée k fils étiquetés $2, \dots, k, k+1$.

Par induction, si on continue la génération d'arborescences par l'opérateur ϑ et la construction d'un arbre de génération on constate l'existence d'une bijection entre les deux arbres illustrés par les figures 2.9 et 2.10. La bijection montre que les nombres de fils générés par chaque nœud des arbres sont égaux, donc on peut établir que la génération d'arborescences par la branche la plus à droite est complète par rapport aux nombres de Catalan.

Dans la suite nous propageons la méthode d'énumération d'arbres décrite ci-dessus pour générer des arbres ordonnés et étiquetés. Étant donné un alphabet Σ et un arbre ordonné et étiqueté $T = (V, E, r, \lambda, \preceq) \in \mathcal{T}_k$, le but consiste à générer tous les arbres $T' \in \mathcal{T}_{k+1}$, en ajoutant un nouveau nœud étiqueté $l \in \Sigma$ à chaque membre de $rmb(T)$. Pour cela, nous définissons le couple (p, l), appelé (p, l)-extension [AAK+02], où p représente la position en pré-ordre dans l'arbre T où un nouveau nœud v (étiqueté l) sera ajouté. Ensuite, l'opérateur $\oplus$, utilisé pour obtenir un arbre T' à partir d'une (p, l)-extension d'un arbre T, est également

Entrées : $\mathcal{T}_k$ *//l'ensemble d'arbres ordonnés et étiquetés à k nœuds*,
Σ *//un alphabet*
Sorties : $\mathcal{T}_{k+1}$ *//l'ensemble d'arbres ordonnés et étiquetés à* $k+1$ *nœuds*

```
1 début
2     T_{k+1} ← ∅;
3     pour chaque arbre T ∈ T_k faire
4         pour chaque position p ∈ rmb(T) faire
5             pour chaque étiquette l ∈ Σ faire
6                 T_{k+1} ← ⋃ T ⊕ (p, l);
7     retourner T_{k+1};
8 fin
```

Algorithme 5 : GENCANDIDATS

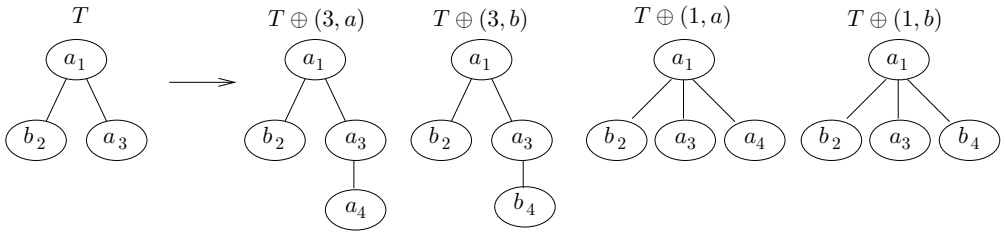


FIG. 2.11 – Génération d'arborescences ordonnées et étiquetées.

modifié de sorte que :

$$T' = T \oplus (p, l),$$

tel que $V' = V \cup \{v\}$, $E' = E \cup \{(u, v)\}$, $pre(v) = k+1$ et $\lambda(v) = l$.

Ainsi, en prenant en compte les arbres étiquetés et ordonnés, nous réécrivons l'algorithme 4 et nous présentons l'algorithme GENCANDIDATS (c.f. Algorithme 5), pour la génération de structures arborescentes candidats de taille $k+1$, à partir d'un ensemble d'arborescences de taille k.

Exemple 14 *Étant donné un alphabet* $\Sigma = \{a, b\}$, *la figure 2.11 illustre les* 4 *arbres générés par extension de la branche la plus à droite de l'arbre* T. *Nous faisons la combinaison des étiquettes* $l \in \Sigma$ *avec les positions* p *dans* $rmb(T) = \{3, 1\}$ *pour former tous les* (p, l)*-extensions possibles. Chaque extension représente un arbre candidat.*

2.3.4. Validation d'arborescences candidates

La deuxième étape de l'algorithme APRIORI consiste à valider le support des candidats afin de décider s'ils sont fréquents ou non. Dans le cas des données transactionnelles, il est

suffisant de vérifier si un itemset X est supporté par une transaction $T = (tid, I)$ de la base de données, c'est-à-dire on vérifie si $X \subseteq I$. En revanche, dans le cas des données semi-structurées nous devons vérifier si une structure arborescente est supportée par une base de données arborescente ce qui implique de tester si un arbre est inclus dans un autre arbre. Ce problème connu comme « inclusion d'arbres » a été initialemente introduit dans [Knu73], puis il a été reconnu comme une primitive importante pour l'interrogation des bases de données structurées dans [KM91]. Par rapport à la complexité de l'inclusion d'arbres, ce problème a été classé *NP-difficile* pour les arbres *non-ordonnés* [MT92, KM95]. Cependant pour le cas d'arbres *ordonnés* une première solution polynômiale a été proposée par [KM95] où, étant donnés deux arbres ordonnés P et T avec n_P et n_T représentant respectivement les nombres de nœuds de P et T, la validation se réalise en $O(n_P n_T)$ temps et espace. Il existe d'autres solutions en temps polynômial [Ric97, Che98, AS01a, BG05] mais plus récemment [BG05] offre une solution optimisée en $O(min(\frac{n_P n_T}{\log n_T}, l_P n_T, n_P l_T \log\log n_T))$ temps et $O(n_T + n_P)$ espace. Les valeurs de l_P et l_T représentent respectivement le nombre de feuilles des arbres P et T.

Dans la suite nous présentons la définition formelle de l'inclusion :

Problème 5 (Inclusion d'arbres, $\sqsubseteq$) *Étant donnés deux arbres ordonnés, un arbre motif P et un arbre cible T, le problème consiste à trouver le plus petit sous-arbre de T qui contient P. Si P est inclus dans T, on note $P \sqsubseteq T$.*

Cette inclusion est validée si on peut obtenir le motif en éliminant des nœuds de l'arbre cible. Pour cela, il faut définir un opérateur $T \ominus u$ qui permette d'enlever le nœud u de l'arbre T. Comme résultat de cette opération, toutes les arêtes incidentes à u seront enlevées et chaque fils de u sera lié au père de u par une nouvelle arête.

Formellement, on obtient un arbre P à partir d'un arbre T en éliminant des nœuds de T s'il existe une séquence de nœuds $u_1, u_2, ..., u_k$ telle que :

$$\begin{cases} T_0 = T, \\ T_{i+1} = T \ominus u_{i+1}, \text{ pour } i = 0, 1, ..., k-1 \end{cases}$$

et si $T_k \cong P$ alors, P est inclus dans T.

Exemple 15 *Pour donner un exemple, considérons les arbres P et T illustrés par la Figure 2.12. Il est évident que l'arbre P est inclus dans T. Pour vérifier cette affirmation, on applique une suite de transformations sur l'arbre $T_0 = T$ de la Figure 2.13, en éliminant les nœuds 2 et 3 jusqu'à obtenir l'arbre $T_2 \cong P$ et en conséquence valider l'inclusion $P \sqsubseteq T$.*

Nous décrivons maintenant les différentes notions d'inclusion traditionnelles employées au long de ce mémoire (voir le mémoire de Kilpeläinen [Kil92] pour un exposé plus détaillé des différents types d'inclusion).

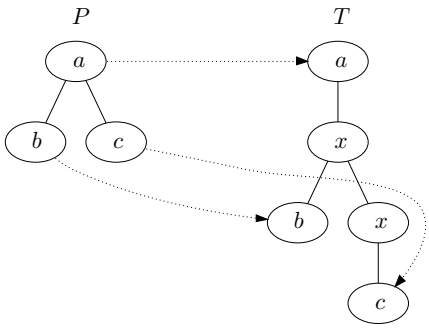


FIG. 2.12 – L'arbre motif P est-il inclus dans l'arbre cible T ?

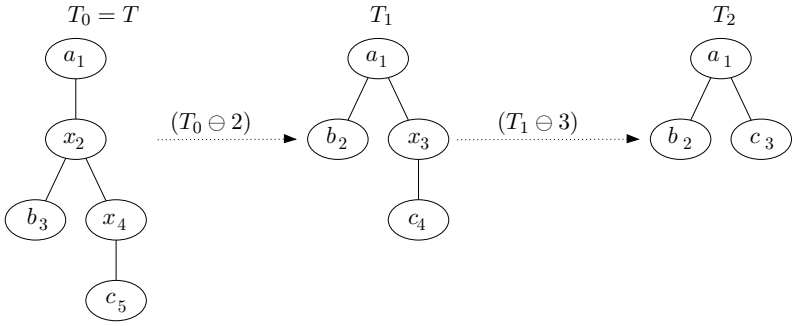
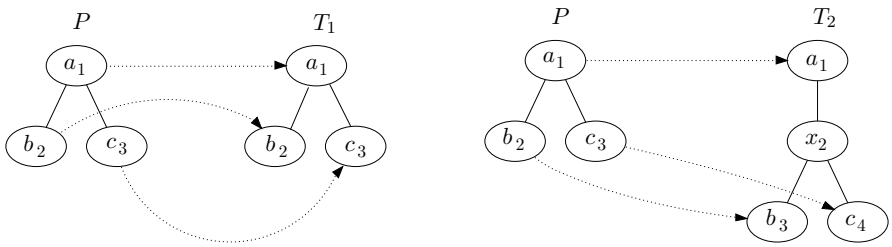


FIG. 2.13 – Transformation de l'arbre T (c.f. 2.12) dans l'arbre P pour valider $P \sqsubseteq T$

Intuitivement, nous dirons que T possède une occurrence de P, s'il existe une fonction qui établit une correspondance entre les nœuds de P et ceux de T, en préservant les étiquettes et la structure de l'arbre P. Nous appellerons cette fonction *mapping*. Par exemple, les lignes pointillées dans la Figure 2.12 représentent un mapping de P dans T. La notion de mapping est traduite formellement de la façon suivante :

Soient P et T deux arbres, nous disons que P est inclus dans un arbre T, noté $P \sqsubseteq T$, s'il existe une fonction (*ou mapping*) pour établir une correspondance des nœuds de P aux nœuds de T, définie par ϕ: $V_P \rightarrow V_T$, telle que pour tout $u, v \in V_P$

1. ϕ est une fonction *injective* (i.e. $u \neq v$ implique $\phi(u) \neq \phi(v)$).
   Si ϕ est bijective, alors P et T sont isomorphes.
2. ϕ préserve les étiquettes : $\lambda_P(u) = \lambda_T(\phi(u))$.
3. ϕ préserve les relations de *père-fils* : $(u, v) \in E_P \iff (\phi(u), \phi(v)) \in E_T$.
4. ϕ doit préserver la relation d'ordre entre frères, c'est-à -dire, si $u \preceq_P v$, alors $\phi(u) \preceq_T \phi(v)$.



(a) *Induite*, $P \sqsubseteq_i T_1$ (b) *Incrustée*, $P \sqsubseteq_e T_2$

FIG. 2.14 – Inclusion d'arbres

Inclusion induite, $\sqsubseteq_i$

On dit que la définition précédente est la plus restrictive car elle prend en compte toutes les contraintes pour établir un mapping (i.e. injectivité des nœuds, préservation des étiquettes des nœuds, relations directes « père-fils » entre nœuds, et ordre entre nœuds frères). Nous appellerons ce type d'inclusion *induite*[6], notée $\sqsubseteq_i$.

Nous pouvons formuler différentes variantes de la définition d'inclusion en relaxant une ou plusieurs de ces contraintes [Kil92].

Inclusion incrustée, $\sqsubseteq_e$

Une première modification consiste à considérer les relations de parenté indirectes entre nœuds (i.e. des relations *ancêtre-descendant*). Ce type d'inclusion est appelé *inclusion incrustée*[7], notée $\sqsubseteq_e$ et dan ce cas, la fonction de correspondance ϕ doit vérifier les conditions suivantes pour tout nœud $u, v \in V_P$:

1. ϕ est une fonction *injective* : $u \neq v$ implique $\phi(u) \neq \phi(v)$.
2. ϕ préserve les étiquettes : $\lambda_P(u) = \lambda_T(\phi(u))$.
3. ϕ préserve les relations *ancêtre-descendant* : $(u, v) \in E_P \iff (\phi(u), \phi(v)) \in E_T^+$.
4. ϕ préserve la relation d'ordre entre frères : si $u \preceq_P v$, alors $\phi(u) \preceq_T \phi(v)$.

Exemple 16 *La figure 2.14(a) illustre l'inclusion* induite *de l'arbre P dans l'arbre T_1 tout en respectant les relations* père-fils *entre nœuds. Par ailleurs, dans la figure 2.14(b) nous trouvons une inclusion* incrustée *de P dans T_2 car celle-ci valide les relations* ancêtre-descendant *des nœuds.*

[6] de l'anglais *induced*
[7] de l'anglais *embedded*

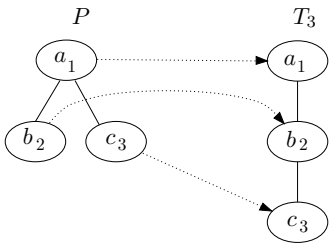


FIG. 2.15 – Inclusion par *subsomption* de P dans T, $P \sqsubseteq_s T_3$

Inclusion par subsomption, $\sqsubseteq_s$

Termier et al. présentent dans [TRS02], une notion d'inclusion encore plus relaxée. Cette définition propose deux modifications principales à la définition d'inclusion incrustée :

- La contrainte 3 est modifiée de telle sorte que on n'a plus l'équivalence entre les arêtes de P et les arêtes de T, mais une seule implication.
- La contrainte 4 indiquant la relation d'ordre entre nœuds frères est supprimée.

Cette inclusion est connue sous le nom d'*inclusion par subsomption*, notée $\sqsubseteq_s$. Dans ce cas le mapping ϕ doit vérifier pour tout nœud $u, v \in V_P$:

1. ϕ est une fonction *injective* : $u \neq v$ implique $\phi(u) \neq \phi(v)$.
2. ϕ préserve les étiquettes : $\lambda_P(u) = \lambda_T(\phi(u))$.
3. ϕ préserve les relations *ancêtre-descendant* : $(u, v) \in E_P \Longrightarrow (\phi(u), \phi(v)) \in E_T^+$.

Exemple 17 *La figure 2.15 illustre l'inclusion par subsomption de l'arbre P dans l'arbre T_1. Celle-ci est injective par rapport aux nœuds, elle respecte les étiquettes des nœuds, mais par contre elle ne préserve pas l'équivalence des relations* ancêtre-descendant *entre l'arbre P et l'arbre T_3.*

2.3.5. Occurrences d'un arbre motif dans une base de données

Dans cette section, nous présentons les différentes définitions d'*occurrence* introduites par Asai dans [AAK+02]. Ces notions seront prises en compte au moment de faire le calcul du support d'un arbre candidat dans une base de données.

Étant donné un type d'inclusion $\sqsubseteq$ (induite, incrustée, par subsomption, etc.), un k-arbre P et une base de données $\mathcal{D} = (D, \Sigma_\mathcal{D}, \lambda_\mathcal{D})$ telle que $P \sqsubseteq D$, une occurrence de P dans T est définie par l'ensemble $\phi = \{\phi(1), ..., \phi(k)\} \subseteq V_D$. L'ensemble d'occurrences est alors défini par $OCC(P, \mathcal{D}) = \{\phi \colon V_P \rightarrow V_D\}$.

Exemple 18 *Si nous prenons par exemple la base de données illustrée par la figure 2.6, l'arbre P de la figure 2.14 et une inclusion de type* incrustée, *nous pouvons identifier les*

occurrences suivantes de P dans la base $\mathcal{D}$: $\phi_1 = \{\phi(1)_1, \phi(2)_1, \phi(3)_1\} = \{2, 3, 4\}$ *et* $\phi_2 = \{\phi(1)_2, \phi(2)_2, \phi(3)_2\} = \{5, 7, 8\}$.

Via OCC, nous pouvons identifier d'autres types d'occurrences :

L'occurrence de la feuille la plus à droite, définie par $moc(\phi) = \phi(k) \in V_D$ où l'ensemble des occurrences de la feuille la plus à droite est donné par $MOC(P, \mathcal{D}) = \{\phi(k) | \phi : V_P \rightarrow V_D\}$.

L'occurrence de la racine définie par $roc(\phi) = \phi(1) \in V_D$ où l'ensemble des occurrences de la racine est défini par $ROC(P, \mathcal{D}) = \{\phi(1) | \phi : V_P \rightarrow V_D\}$

L'occurrence dans un document définie par $doc(\phi) = t \in \Sigma_{\mathcal{D}}$ où l'ensemble d'occurrences dans un document est défini par $DOC(P, \mathcal{D}) = \{\lambda_{\mathcal{D}}(u) | u \in \phi\}$.

Exemple 19 *D'après l'exemple 18, nous avons deux occurrences de la feuille la plus à droite de P dans $\mathcal{D}$, $moc(\phi_1) = 4$ et $moc(\phi_2) = 8$, alors $MOC(P, \mathcal{D}) = \{4, 8\}$. Les occurrences de la racine sont représentées par $roc(\phi_1) = 2$ et $roc(\phi_2) = 5$, elles sont groupées dans l'ensemble $ROC(P, \mathcal{D}) = \{2, 5\}$. Finalement, nous trouvons des occurrences dans les documents définies par $doc(\phi_1) = \{t_1\}$ et $doc(\phi_2) = \{t_1\}$ groupées dans l'ensemble $DOC(P, \mathcal{D}) = \{t_1, t_2\}$.*

2.4. Énoncé du problème

Étant donné un arbre P, une base de données $\mathcal{D}$, et un type d'occurrence $\tau \in \{occ, moc, roc, doc\}$, le *support* de P est défini de la façon suivante :

- Si $\tau = doc$ alors $support(P, \mathcal{D}) = \frac{|\tau(P, \mathcal{D})|}{|\mathcal{D}|}$ où $|\tau(P, \mathcal{D})|$ est le nombre d'occurrences de P dans la base de données. Le support correspond au ratio entre le nombre d'arbres dans $\mathcal{D}$ qui contiennent au moins une occurrence de P et le nombre total d'arbres qui font partie de la base $\mathcal{D}$.
- Si $\tau \in \{occ, moc, roc\}$ le support de P est appelé *support pondéré* noté $support_w(P, \mathcal{D})$ et défini par $support_w(P, \mathcal{D}) = \frac{|\tau(P, \mathcal{D})|}{||\mathcal{D}||}$

Pour chaque entier non-négatif k, l'ensemble de tous les k-sous-arbres fréquents est noté par $\mathcal{F}_k$ et l'ensemble de tous les sous-arbres fréquents par $\mathcal{F} = \cup_k \mathcal{F}_k$.

Le problème de la recherche de structures arborescentes fréquentes est décrit formellement de la façon suivante :

Problème 6 (Recherche de structures arborescentes fréquents) *Étant donnés une base de données arborescente $\mathcal{D}$, et un support minimal σ, le problème consiste à extraire l'ensemble d'arbres fréquents $\mathcal{F}(\mathcal{D}, \sigma)$, noté simplement $\mathcal{F}$, tel que $\mathcal{F} = \{F | support(F, \mathcal{D}) \geq \sigma\}$ ou bien $\mathcal{F} = \{F | support_w(F, \mathcal{D}) \geq \sigma\}$ en considérant un support pondéré.*

2.5. Conclusion

Dans ce chapitre d'état de l'art nous avons tout d'abord étudié l'application de l'approche Apriori [AS94] aux données transactionnelles pour l'extraction des itemsets fréquents. Deux étapes principales ont été distinguées à l'intérieur de la méthode Apriori : 1) la génération de candidats et 2) la validation des candidats. Ensuite nous avons montré que cette approche pouvait être aussi appliquée à un autre type de données plus complexe, plus particulièrement des données semi-structurées, afin d'extraire des structures arborescentes fréquents. Cependant la nature arborescente des données semi-structurées implique la modification de la méthode Apriori présentée dans la section 2.3.2. Concernant la génération de candidats, la section 2.3.3 a introduit la méthode de génération d'arborescences, appelée génération *par la branche la plus à droite*, proposée initialement par Nakano [Nak02] et utilisée postérieurement par Asai et Zaki [AAK+02, Zak02]. Pour l'étape de validation de candidats, la section 2.3.4 a présenté le problème de ***l'inclusion d'arbres*** [Knu73, KM91, Ric97, Che98, AS01a, BG05] où sont définies les contraintes à prendre en compte au moment de valider l'inclusion d'un arbre motif dans un arbre cible. Ensuite nous avons présenté les différents types d'*occurrence* selon lesquels nous faisons le calcul du support des arbres candidats. Finalement dans ce chapitre nous avons présenté de façon formelle la problématique appelée ***recherche de structures arborescentes fréquentes***, traitée dans cette thèse.

Le chapitre suivant est destiné à présenter quelques unes des principales approches proposées dans la littérature pour l'extraction de sous-arbres fréquents.

Chapitre 3

Approches existantes

Dans ce chapitre nous détaillons les approches principales de la littérature pour l'extraction de structures arborescentes fréquentes. Tout d'abord, nous présentons une révision de l'approche proposée par Asai dans [AAK+02] pour l'extraction d'arbres ordonnés fréquents en prenant en compte un type d'inclusion induite. Nous poursuivons par la présentation de l'algorithme TreeMiner proposé par Zaki dans [Zak02] et qui a par but la recherche d'arbres fréquents ordonnés en considérant une inclusion incrustée. Nous faisons ensuite le point sur l'algorithme Tides présente par Tatikonda dans [TPK06] pour l'extraction de structures arborescentes fréquentes employant l'approche *Pattern-growth*. Finalement nous abordons le travail mené par Termier dans [TRS02] également destiné à la recherche d'arbres fréquents mais en considérant l'inclusion par subsomption.

Sommaire

3.1. Introduction **38**
3.2. L'algorithme Freqt, une approche Apriori en largeur **39**
3.3. L'algorithme TreeMiner, une approche Apriori en profondeur **44**
3.4. L'algorithme Tides, une approche Pattern-Growth **52**
3.5. L'algorithme TreeFinder, une méthode logique inductive . . . **57**
3.6. Conclusion **61**

3.1. Introduction

Dans le chapitre précédent, nous avons introduit les notions fondamentales pour effectuer la recherche de motifs fréquents (*couverture, support, fréquence*). Nous avons vu aussi que les documents semi-structurés sont représentés par des arbres. Pour cette raison, nous avons présenté les concepts nécessaires au traitement des arbres (*racine, profondeur, parent, descendants, ancêtres, frères, etc.*). Puis nous avons étudié comment la notion d'inclusion d'arbres peut être modifiée en changeant quelques unes des propriétés du mapping (*injectivité, étiquetage, relation ancêtre-descendant, ordre*). Finalement nous avons formalisé le problème de la recherche de structures arborescentes qui consiste, à partir d'une base de données d'arbres, à extraire les sous-arbres qui apparaissent fréquemment dans la base, (i.e. les sous-arbres dont le nombre d'occurrences est supérieur à un support minimal σ spécifié par l'utilisateur, c.f. Problème 6).

Par ailleurs, nous soulignons que ces dernières années, de nouvelles approches pour la fouille de données complexes ont été proposées car les approches traditionnelles de fouille de données (i.e. recherche d'itemsets, de motifs séquentiels,...) ne sont pas adaptées pour manipuler la complexité de ces données.

Dans ce chapitre, nous décrivons différentes techniques de fouille d'arborescences fréquentes en nous focalisant sur chacune de leurs différences. En particulier, nous considérons les quatre méthodes suivantes : 1) FREQT [AAK+02], 2) TREEMINER [Zak02], 3) TIDES [TPK06] et 4) TREEFINDER [TRS02]. Les deux premières sont des approches basées sur le principe APRIORI [AIS93] car les arbres possèdent la propriété d'anti-monotonie (c.f. Propriété 1) : « *Si un arbre n'est pas fréquent, alors tout sur-arbre le contenant sera également non fréquent* ». Ces techniques sont des méthodes de type *générer-élaguer* qui extraient les sous-arbres par l'intermédiaire d'algorithmes en largeur (appelés aussi par niveau) ou par des algorithmes en profondeur, avec lesquels on distingue deux phases principales :

La génération des arbres candidats, une phase dans laquelle les candidats de taille $k+1$ sont construits à partir des arbres fréquents de taille k.

La validation du support des candidats, une phase qui vérifie les candidats sur la base de données pour rechercher les sous arbres fréquents, et élague les candidats non fréquents.

Le troisième algorithme, TIDES, est par contre basé sur l'approche appelée PATTERN-GROWTH [HPY00, PH02], une technique dite *sans génération de candidats*. Enfin, nous exposons l'algorithme TREEFINDER, une méthode intéressante mais incomplète, reposant sur les techniques de programmation logique inductive qui permettent l'extraction d'arbres motifs, relaxant les relations de parenté et d'ordre entre nœuds.

Comme nous avons vu dans la section 2.3.1, nous considérons une base de données notée $\mathcal{D}$, représentant une collection de documents semi-structurés. Rappelons que l'ensemble des arbres d'une base de données arborescente $\mathcal{D}$ peut être regroupé en un même arbre (i.e.

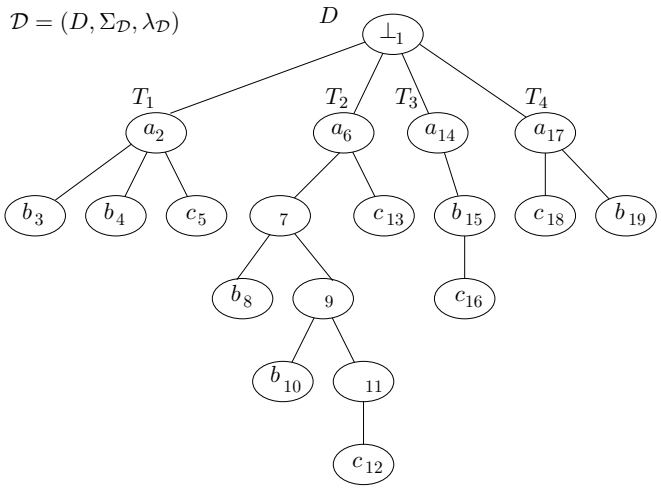


FIG. 3.1 – Une base de données arborescente

l'arbre de données D) par l'ajout d'une racine comme illustré par l'exemple ci-dessous. Afin de simplifier la notation nous appellerons aussi base de données l'arbre de données D.

Exemple 20 *La figure 3.1 illustre l'arbre de données D appartenant à la base de données $\mathcal{D}$. Cet arbre sert à grouper les sous-arbres T_1, T_2, T_3, T_4 représentant des documents semi-structurés. Nous nous servirons de cette figure pour détailler les exemples dans chacune des approches exposées dans ce chapitre. Les étiquettes des nœuds* $7, 9$ *et* 11 *ont été omises, ces nœuds servant à illustrer les relations indirectes* ancêtre-descendant.

Sauf mention contraire, tous les arbres considérés ici sont des arbres enracinés, étiquetés et ordonnés et nous omettons donc de le préciser. Nous rappelons que, étant donné un arbre $T = (V, E, r, \lambda, \preceq)$ et un nœud $u \in V$, la notation $u \in T$ indique que u appartient à T. Si T est un arbre à k nœuds (on parle alors d'un k-arbre ou d'un arbre sur $[k]$), les nœuds sont énumérés avec des entiers de 1 à k selon un parcours en profondeur d'abord de T, de telle sorte que $root(T) = 1$ et $rml(T) = k$.

3.2. L'algorithme FREQT, une approche APRIORI en largeur

L'algorithme FREQT (c.f. Algorithme 6) a été proposé par Asai dans [AAK+02], comme une solution à la recherche de sous-arbres fréquents dont l'inclusion est de type *induite* c'est-à-dire que les mappings ϕ des candidats dans une base de données doivent :

1. être injectives,
2. préserver les étiquettes,
3. préserver les relations *père-fils* entre nœuds, et

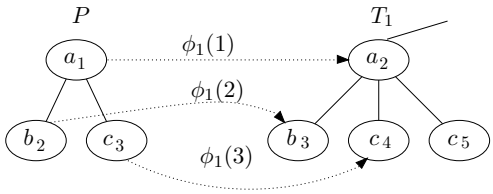


FIG. 3.2 – Inclusion induite $P \sqsubseteq_i T_1$

4. préserver l'ordre entre nœuds frères

Exemple 21 *Considérons l'arbre P et le sous-arbre T_1 (inclus dans notre base de données) illustrés par la figure 3.2. On trouve que l'arbre P a deux mappings ϕ_1 et ϕ_2 respectant les conditions d'une inclusion induite tout en établissant une correspondance entre les nœuds de P vers les nœuds de T_1 tel que $\phi_1 : \{\phi_1(1), \phi_1(2), \phi_1(3)\} = \{2, 3, 4\}$ et $\phi_2 : \{\phi_2(1), \phi_2(2), \phi_2(3)\} = \{2, 3, 5\}$ (i.e. le nœud 1 de P correspond au nœud 2 de T_1, le nœud 2 de P au nœud 3 de T_1, et le nœud 3 de P au nœud 4 de T_1). À partir de la figure, il est évident qu'il existe un autre mapping $\phi_2 : \{2, 3, 5\}$. En conséquence, l'ensemble d'occurrences induites de P dans la base de données D est donné par la liste $OCC(P, D) = \{\{2, 3, 4\}, \{2, 3, 5\}\}$.*

L'approche FREQT est dite de type APRIORI, car elle adopte une recherche *par niveau* ou *en largeur d'abord* pour trouver tous les arbres fréquents. Ainsi, à chaque niveau de l'algorithme, des arbres candidats sont générés à partir des arbres trouvés fréquents au niveau précédent.

Le niveau k fait référence à l'étape de l'algorithme où sont traités les arbres de taille k. Ces arbres sont ensuite vérifiés sur la base de données afin d'établir leur support. Dans ces approches, la génération des arbres candidats est donc cruciale, l'enjeu étant de générer tout arbre fréquent comme un candidat tout en minimisant le nombre de candidats possibles pour être efficace. Pour restreindre ce nombre, les approches traditionnelles s'appuient sur la propriété d'anti-monotonie [PH02].

La proposition d'Asai utilise la technique de la programmation dynamique pour optimiser la validation du support des candidats. Cette technique est un principe général d'optimisation algorithmique qui dans ce contexte, consiste à stocker des occurrences d'un sous-arbre de taille k, évitant de ce fait le recalcul des occurrences des sous-arbres de taille $k + 1$. Le stockage des occurrences est maintenu par des listes d'occurrences de la feuille la plus à droite (obtenues à partir de $\phi(k)$), notées MOC. Ces listes sont obtenues à partir des listes d'occurrence OCC (e.g. $MOC = \{\phi_1(3), \phi_2(3)\} = \{4, 5\}$).

La première étape dans l'algorithme 6 (ligne 2) est la découverte des sous-arbres fréquents de taille 1. Ici, on propose simplement chaque étiquette l incluse dans l'alphabet Σ comme candidat de taille 1 et on parcourt ensuite la base de données pour calculer le support de chaque candidat et déterminer s'il est fréquent ou non. L'ensemble des sous arbres fréquents

```
   Entrées : Σ //un alphabet,
             D //une base de données sur Σ,
             σ //un support minimal
   Sorties : F //l'ensemble de tous les sous-arbres fréquents dans D
 1 début
 2     ⟨F_1, MOC_1⟩ ← L'ensemble d'arbres fréquents de taille 1;
 3     ⟨F_2, MOC_2⟩ ← L'ensemble d'arbres fréquents de taille 2;
 4     k ← 2;
 5     tant que F_k ≠ ∅ faire
 6         //Génération des k-candidats;
 7         ⟨C_{k+1}, MOC_{k+1}⟩ ← EXPAND-TREES(F_k, MOC_k, F_1);
 8         pour chaque candidat C ∈ C_{k+1} faire
 9             //Valide le support de C à partir de MOC_{k+1}(C);
10             si support(C, D) ≥ σ alors
11                 F_{k+1} = ∪{C};
12         k ← k + 1;
13     retourner F = F_1 ∪ F_2 ∪ ... ∪ F_{k−1};
14 fin
```

Algorithme 6 : FREQT

à 1 nœuds et les occurrences de chacun au sein de la base de données sont gardés dans $\mathcal{F}_1$ et MOC_1 respectivement.

La deuxième étape destinée à la découverte des sous-arbres fréquents de taille 2 (ligne 3), est similaire au pas décrit ci-dessus. La seule différence est que, ici, on propose des 2-candidats (des arbres composés par un nœud père et un nœud fils) formés par des couples $(l_1, l_2) \in \Sigma^*$. L'étiquette l_1 correspond au père et l'étiquette l_2 est pour le fils. Les résultats obtenus à ce niveau sont stockés dans $\mathcal{F}_2$ et MOC_2.

Le bloc suivant (lignes 4-12) commence par une boucle à l'étape 2 (ligne 4), Après, la fonction EXPAND-TREES (ligne 7) est chargée de construire les arbres candidats de taille $k+1$ à partir des arbres fréquents de taille k. Cette fonction actualise aussi les listes d'occurrences. Une fois les candidats générés, le support de chaque candidat par rapport à la base de données est calculé à partir des listes MOC (ligne 8). Si le support d'un candidat dépasse le seuil σ établi pour l'utilisateur (ligne 10), alors ce candidat est ajouté à l'ensemble d'arbres fréquents (ligne 11). Le compteur des étapes k est augmenté à la ligne 12 pour relancer la boucle à la ligne 5. Ce processus itératif s'arrête lorsqu'il n'y a plus d'arbres fréquents de taille k. L'algorithme FREQT finalise en rassemblant tous les arbres qui ont été trouvés jusqu'à la $(k-1)^{\text{ième}}$ étape.

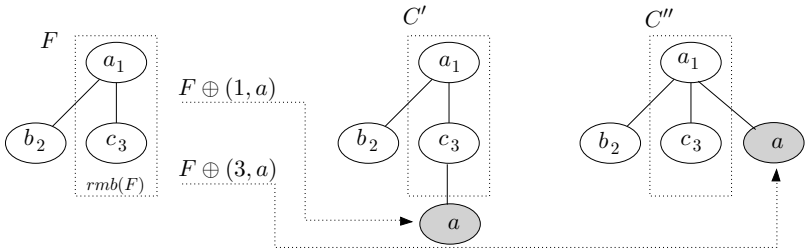


FIG. 3.3 – Des extensions (p, l) de l'arbre F

3.2.1. Génération de candidats

La génération de candidats se fait en ajoutant un nouveau nœud à un arbre fréquent F de taille k, de telle sorte qu'un candidat C de taille $k + 1$ soit construit. Comme cela est détaillé dans la section 2.3.3, un nouveau nœud peut être ajouté seulement aux nœuds appartenant à la branche la plus à droite d'un arbre fréquent.

Exemple 22 *La figure 3.3 illustre l'arbre F de taille 3 combiné avec deux (p, l)-extensions, $(1, a)$ et $(3, a)$ donnant comme résultat les arbres candidats C' et C'' de taille 4. Nous rappelons que le premier élément p dans une (p, l)-extension représente la position d'un nœud appartenant à la branche la plus à droite d'un arbre, où on va ajouter un nouveau nœud étiqueté l.*

L'algorithme EXPAND-TREES (c.f. Algorithme 7) est chargé de la génération d'arbres candidats. L'algorithme parcourt tous les arbres k-fréquents (ligne 3). Chaque nœud u appartenant à la branche la plus à droite d'un arbre fréquent représente une opportunité pour la création d'un nouveau candidat (ligne 4). Pour générer les candidats, Asai propose une optimisation cherchant à réduire le nombre de candidats, appelée *edge-skip*. Celle-ci prend en compte les arbres fréquents de taille 2 (ligne 5), à la différence de l'algorithme GENCANDIDATS (c.f. Algorithme 5) qui utilise toutes les étiquettes dans l'alphabet Σ sur lequel est construite la base de données. Alors, chaque nœud u est combiné avec chaque arbre dans $\mathcal{F}_2$ décrit par sa représentation par des parenthèses où l_1 est l'étiquette de la racine et l_2 est l'étiquette du fils de la racine. Si l'étiquette du nœud u et l'étiquette l_1 sont égales (ligne 6), alors la création du nouveau candidat se fait par extension d'un arbre fréquent F (ligne 8). Le pas suivant consiste à la mise à jour des listes MOC du candidat généré (ligne 9,10). Finalement l'algorithme retourne l'ensemble de candidats chacun muni de sa liste d'occurrences MOC (ligne 11).

3.2.2. Validation du support

Les listes d'occurrences sont un moyen efficace pour stocker les mappings d'un sous-arbre dans la base de données. Au lieu de stocker toute l'information d'une occurrence ϕ,

Entrées : $\mathcal{F}_k$ *//des sous-arbres fréquents de taille* k,
MOC_k *//listes des occurrences*,
$\mathcal{F}_1$ *//sous-arbres fréquents de taille 1*
Sorties : $\langle \mathcal{C}_{k+1}, MOC_{k+1} \rangle$ *//une liste de candidats avec sa liste d'occurrences*

```
1  début
2      ⟨C_{k+1}, MOC_{k+1}⟩ ← ∅;
3      pour chaque arbre F ∈ F_k faire
4          pour chaque nœud u ∈ rmb(F) faire
5              pour chaque arbre (l1(l2)) ∈ F_2 faire
6                  si λ(u) = l1 alors
7                      p ← pre(u);
8                      C ← F ⊕ (p, l2);
9                      MOC(C) ← UPDATE-MOC(MOC(F), (p, l2));
10                     ⟨C_{k+1}, MOC_{k+1}⟩ ← ⋃⟨C, MOC(C)⟩;
11     retourner ⟨C_{k+1}, MOC_{k+1}⟩;
12 fin
```

Algorithme 7 : EXPAND-TREES

l'algorithme garde seulement les occurrences des feuilles les plus à droite $moc(\phi)$.

L'algorithme 7 reçoit l'ensemble d'occurrences $moc(\phi)$ d'un arbre fréquent F de taille k, noté MOC_F, et un couple (p, l). Avec ces deux paramètres UPDATE-MOC crée une nouvelle liste d'occurrences pour la (p, l)-extension de F, notée MOC_C. La construction de MOC_C commence par une recherche à partir de chaque position x enregistrée dans MOC_F. Puis, on valide la position p à laquelle un nouveau nœud a été ajouté dans F. Si $p = rml(F)$, cela signifie qu'il faut chercher un nœud dans les fils de x qui ait une étiquette égale à l. Au contraire, si $p \neq rml(F)$, alors on commence la recherche par les nœuds frères du nœud à la position p. La recherche des nœuds étiquetés l dans la base de données s'arrête lorsqu'il n'y a plus de nœuds fils ou frères de x. Afin d'éviter la duplication d'information dans MOC_C, UPDATE-MOC inclut la variable *check* pour valider si les frères d'un $p-1$ ancêtre de x ont déjà visités été.

Exemple 23 *Prenons par exemple la base de données illustrée par la figure 3.1 et un support minimal* $\sigma = 1$ *nous indiquant que pour considérer un sous-arbre fréquent, il suffit qu'il soit inclus dans un seul des sous-arbres* T_1, T_2, T_3 *ou* T_4 *de l'arbre de données* D. *Dans la figure 3.4 nous présentons un exemple illustrant la découverte de sous-arbres fréquents selon l'algorithme* FREQT. *La première colonne indique l'étape ou niveau de recherche. La deuxième colonne énumère les k-arbres plats, et sa représentation par des parenthèses, valables pour générer les candidats. Les instances fréquentes de chaque k-étape combinées avec*

Entrées : MOC_F //*la liste d'occurrences d'un arbre fréquent de taille*,
(p, l) //*une extension sur l'arbre* F
Sorties : MOC_C //*la liste d'occurrences d'un arbre candidat*

```
1  début
2      MOC_C ← ∅;
3      check ← ∅;
4      pour chaque élément x ∈ MOC_F faire
5          //x, y des nœuds dans l'arbre de données D;
6          si p = rml(F) alors
7              y ← first(x);
8          sinon
9              si check = x alors
10                 continuer au suivant x ∈ MOC_F        //détection de doublons;
11             sinon
12                 y ← next(anc^(p−1)(x));
13                 check ← x;
14         tant que y ≠ NIL faire
15             si λ(y) = l alors
16                 MOC_C ∪ {y}
17             y ← next(y);
18 fin
```

Algorithme 8 : UPDATE-MOC

les arbres fréquents $\mathcal{F}_2$ sont employées pour générer les candidats de taille supérieure $k+1$. Le processus s'arrête à l'étape 5, lorsqu'on ne trouve plus d'arbres fréquents.

3.3. L'algorithme TreeMiner, une approche Apriori en profondeur

Zaki propose dans [Zak02] les algorithmes : *hTreeMiner* et *vTreeMiner*, pour la recherche de sous-arbres fréquents dans des bases de données. Le premier utilise une recherche itérative en largeur d'abord, pour trouver tous les sous-arbres fréquents. En revanche, le deuxième emploie une recherche récursive en profondeur d'abord et une représentation verticale des arbres pour une validation rapide du support des candidats. Le travail proposé par Zaki considère une inclusion entre arbres de type *incrustée*. Cette définition permet de généraliser la définition traditionnelle d'un sous-arbre en considérant les relations *ancêtre-descendant* au lieu des relations *père-fils*. Les occurrences ϕ des candidats à l'intérieur d'une base de

Étape k	Énumération	Candidats $\mathcal{C}_k$	Fréquents $\mathcal{F}_k$
1	()	$\{(a), (b), (c), \ldots\}$	$\{(a), (b), (c)\}$
2	 (())	$\{(a(a)), (a(b)), (a(c))$ $, (b(a)), (b(b)), (b(c))$ $, (c(a)), (c(b)), (c(c)), \ldots\}$	$\{(a(b)), (a(c)), (b(c))\}$
3	 ((()))	$\{(a(b(c)))$	$\{(a(b(c)))\}$
	 (()())	$, (a(b)(b)), (a(b)(c))$ $, (a(c)(b)), (a(c)(c)), (b(c)(c)), \ldots\}$	$\{(a(b)(b)), (a(b)(c))$ $, (a(c)(b))\}$
4	 ((())())	$\{(a(b(c))(b)), (a(b(c))(c))$	
	 (((())))	$, (a(b(c)(c)))$	$\{(a(b)(b)(c))\}$
	 (()()())	$, (a(b)(b)(b)), (a(b)(b)(c)), (a(b)(c)(b))$ $, (a(b)(c)(c)), (a(c)(b)(b)), (a(c)(b)(c)), \ldots\}$	
5	...	...	$\emptyset$

Fig. 3.4 – Processus d'extraction d'arbres fréquents selon l'algorithme Freqt

données doivent :

1. être injectives
2. préserver les étiquettes
3. préserver les relations *ancêtre-descendant* entre nœuds
4. préserver l'ordre entre nœuds frères

Exemple 24 *Dans la figure 3.5 on constate que l'arbre candidat C a une occurrence incrustée $\phi_1 = \{6, 8, 12\}$ de ses nœuds aux nœuds de l'arbre T_2 et qui respecte les conditions spécifiées ci-dessus. Chaque occurrence d'un arbre motif C dans une cible T est identifiée par un ensemble de positions nommé* étiquettes de correspondance. *Considérant la base de données illustre par la figure 3.1, le sous-arbre C apparaît deux fois dans D aux positions indiquées par les étiquettes de correspondance et représentées par la liste $OCC(C, D) = \{\{2, 3, 5\}, \{2, 4, 5\}, \{6, 8, 12\}, \{6, 8, 13\}, \{6, 10, 12\}, \{6, 10, 13\}\}$. Toutefois, le sous-arbre C n'a que deux* occurrences de document *à l'intérieur de la base de données, c'est-à-dire $DOC(C, D) = \{T_1, T_2\}$.*

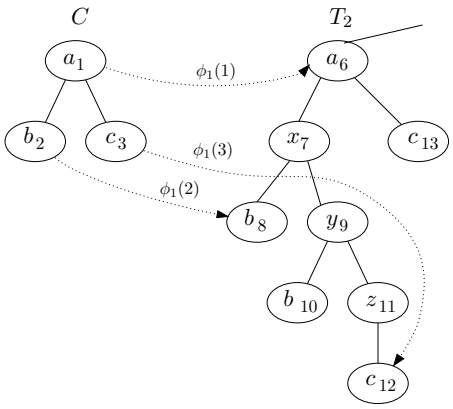


FIG. 3.5 – Inclusion incrustée $C \sqsubseteq_e T_2$

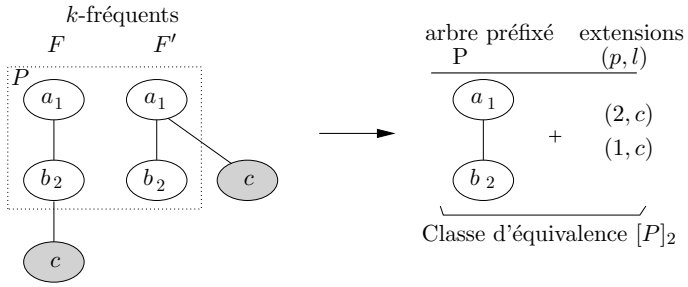


FIG. 3.6 – Une classe d'équivalence

3.3.1. Génération de candidats

La génération de candidats se fait par l'extension des éléments fréquents. Á chaque étape, on ajoute un nouveau nœud. Autrement dit, les candidats de taille $k+1$ sont créés à partir des éléments fréquents de taille k.

Classes d'équivalence

Tous les arbres fréquents de taille k (k-arbre) avec un sous-arbre préfixé P de taille $k-1$ en commun sont groupés dans des classes d'équivalence. Ces dernières sont utilisées par Zaki dans l'algorithme ECLAT [ZPOL97]. Les classes d'équivalence notées $[P]_{k-1}$, sont composées par le sous-arbre P et une liste de couples (p, l) (similaires aux (p, l)-extensions définies par Asai) qui représentent les feuilles les plus à droite attachées à chaque k-arbre fréquent. Les éléments du couple (p, l) correspondent respectivement à la position dans P et à l'étiquette où la feuille la plus à droite est attachée.

Exemple 25 *La figure 3.6 montre la classe d'équivalence $[P]_2$ formée à partir des arbres fréquents F et F' de taille 3. La classe $[P]_2$ est composée de l'arbre préfixé P, l'arbre*

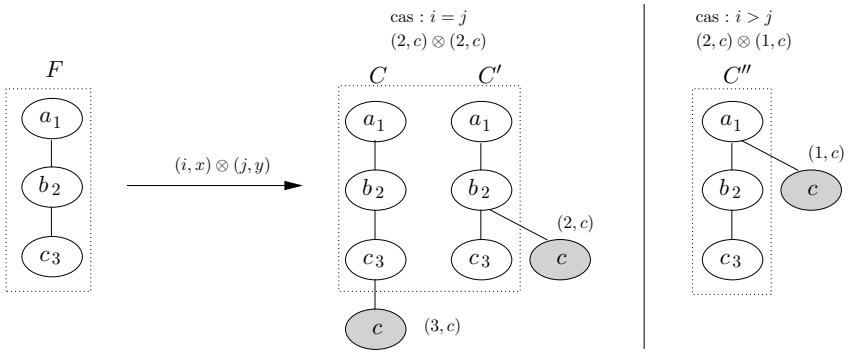


FIG. 3.7 – Des candidats avec un arbre préfixé P égal à F l'arbre fréquent de la figure 3.6.

commun à F et F' jusqu'à la position $k-1$, et les couples $(2,c)$ et $(1,c)$. Dans le premier couple $(2,c)$ par exemple, c est l'étiquette de la feuille la plus à droite de F, attachée à la position 2 de l'arbre P.

Une fois les classes d'équivalence formées, la génération de candidats lance un processus de construction de nouvelles classes d'équivalence à partir de la *jointure* des éléments appartenant à des classes d'équivalence déjà créées. De manière similaire à la méthode FREQT, l'ensemble des positions valables dans P auxquelles un nouveau nœud u peut être ajouté, dans cette approche, est donnée par la branche la plus à droite de P. L'algorithme TREEMINER utilise la procédure définie par le théorème 1, où l'idée principale est la construction d'une nouvelle classe d'équivalence à partir de la jointure des éléments appartenant à une classe d'équivalence déjà créée.

Définition 1 (Jointure des classes d'équivalence) *Étant donné un arbre préfixé P et deux couples quelconques ((i,x) et (j,y)) appartenant à une classe d'équivalence $[P]_{k-1}$ et considérant P_x, le sous-arbre contenant l'élément x, un opérateur de jointure, noté $\otimes$, est défini sur les éléments (i,x) et (j,y) tel que :*

1. *Si $(i = j)$:*
   a) *Si $P = \emptyset$ alors, ajouter l'élément $(j+1,y)$ à la classe $[P_x]_k$*
   b) *Si $P \neq \emptyset$ alors, ajouter les éléments (j,y) et $(j+1,y)$ à la classe $[P_x]_k$*
2. *Si $(i > j)$: alors, ajouter l'élément (j,y) à la classe $[P_x]_k$*
3. *Si $(i < j)$: la création de nouveaux candidats n'est pas possible.*

Exemple 26 *Considérons la classe d'équivalence $[P]_2$, illustrée par la figure 3.6. Elle est composée par l'arbre préfixé P et les éléments $(2,c)$ et $(1,c)$. D'abord on réalise l'extension des classes en combinant les éléments $(2,c)$ et $(1,c)$ (c.f. Figure 3.7). La jointure du premier*

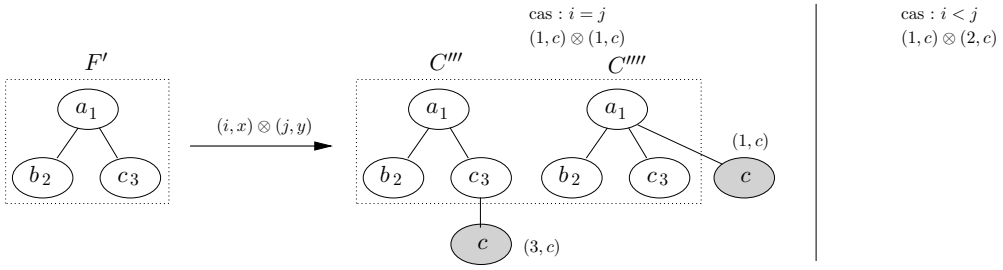


FIG. 3.8 – Des candidats avec un arbre préfixé P égal à F' l'arbre fréquent de la figure 3.6.

élément avec lui même $(2, c) \otimes (2, c)$ *satisfait la première condition de la définition précédente* $(2 = 2)$ *donnant comme résultat les arbres candidats* C *et* C'. *La combinaison* $(2, c) \otimes (1, c)$ *vérifie la deuxième condition de la définition car* $2 > 1$ *et génère l'arbre candidat* C''. *Les trois candidats générés* C, C', C'' *à son tour, font partie d'une nouvelle classe d'équivalence* $[F]_3$ *composée par* F *comme l'arbre préfixé et les éléments* $(3, c)$, $(2, c)$ *et* $(1, c)$. *Lorsqu'on fait la jointure des éléments* $(1, c) \otimes (1, c)$ *(c.f. Figure 3.8), on obtient les candidats* C''' *et* C'''' *(une nouvelle fois la première condition du théorème est accomplie car les positions sont égales i.e.* $1 = 1$*). Finalement, selon la troisième condition du théorème, la combinaison* $(1, c) \otimes (2, c)$ *ne génère aucune candidat car* $1 < 2$. *Une autre classe d'équivalence* $[F']_3$ *est créée par les arbres* C''' *et* C''''. *Elle est formée du préfixe* F', *et des éléments* $(3, c)$ *et* $(1, c)$.

3.3.2. Validation du support

L'*étendue* d'un nœud $u \in T$, notée $s = [u, v]$, est un concept utilisé par Zaki, pour réaliser la validation du support d'un candidat de façon efficace. Pour chaque sous-arbre $T(u), u \in V$ de T, la limite inférieure de l'étendue est donnée par la position en pré-ordre du nœud u, et la limite supérieure v correspond à la position de la feuille la plus à droite de $T(u)$, telle que $v = rml(T(u))$.

Exemple 27 *Nous illustrons dans la figure 3.9, l'étendue* $s = [7, 12]$ *du sous-arbre enraciné au nœud* 7 *de l'arbre* T. *Cette intervalle correspond aux positions de tous les nœuds appartenant au sous-arbre* $T_2(7)$.

hTreeMiner

Initialement proposé par Zaki, l'algorithme *hTreeMiner* (c.f. Algorithme 9), est ainsi nommé car il emploie toujours une représentation horizontale des arbres (i.e. chaîne de caractères), autant pour la génération de candidats que pour la validation de son support. *hTreeMiner* utilise une recherche de structures fréquentes en largeur d'abord dans un processus itératif similaire à l'algorithme *Apriori*[AMS+96]. Premièrement dans une étape k,

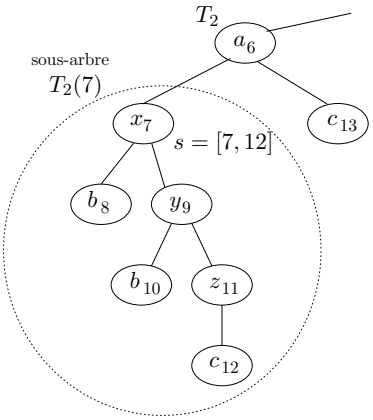


Fig. 3.9 – L'étendue d'un nœud.

il propose toutes les structures candidates de taille k et après il passe à la validation du support de chaque candidat pour déterminer si ce candidat est fréquent ou non. Cette approche a pour inconvénient l'utilisation d'un grand espace de mémoire afin de manipuler tous les candidats possibles de taille k en même temps. En revanche, on a l'avantage d'une grande capacité d'élagage de candidats.

Entrées : Σ *//un alphabet*,
$\mathcal{D}$ *//une base de données sur* Σ,
$0 < \sigma \leq 1$ *//un support minimal*
Sorties : $\mathcal{F}_k$ *//l'ensemble des k-sous-arbres fréquents dans D*

```
1 F_1 ← L'ensemble de sous-arbres fréquents de taille 1;
2 F_2 ← L'ensemble de sous-arbre fréquents de taille 2;
3 pour (k = 3; F_{k-1} ≠ ∅; k = k + 1) faire
4     C_k =Des classes [P]_{k-1} avec les k-candidats;
5     pour chaque arbre T ∈ D faire
6         //Incrémenter le comptage de tous les candidatsC ⊑_e T, C ∈ [P]_{k-1};
7         si support(C, D) ≥ σ alors
8             F_k = ∪{C};
9     F_k = classes des k-sous-arbres fréquents;
```

Algorithme 9 : hTreeMiner

vTreeMiner

vTreeMiner (c.f. Algorithme 10) doit son nom à la représentation verticale des arbres introduite par Zaki pour le comptage de la fréquence des arbres candidats. Cette représentation est basée sur les listes d'occurrence définies par la suite. Soit X un k-sous-arbre d'un arbre

T. Soit x_k la feuille la plus à droite de X. On note par $\mathcal{L}(X) = (t, e, s)$ la liste d'étendue de X où t est l'identifiant de l'arbre T, e l'étiquette de correspondance de X dans T, et s l'étendue de x_k.

Entrées : Σ *//un alphabet,*
$\mathcal{D}$ *//une base de données sur* Σ,
$0 < \sigma \leq 1$ *//un support minimal*
Sorties : $\mathcal{F}_k$ *//l'ensemble des k-sous-arbres fréquents dans D*

```
1 F_1 ← L'ensemble de sous-arbres fréquents de taille 1;
2 F_2 ← L'ensemble de sous-arbre fréquents de taille 2;
3 pour chaque classe d'équivalence [P]_1 ∈ E faire
4     EnumerateFrequentSubtrees([P]_1)
```

Algorithme 10 : vTreeMiner

Entrées : $[P]$ *//une classe d'équivalence*
Sorties : $\mathcal{F}_k$ *//l'ensemble des sous-arbres fréquents dans D générés à partir de* $[P]$

```
1 pour chaque élément (i, x) ∈ [P] faire
2     [P_x] = 0;
3     pour chaque élément (j, y) ∈ [P] faire
4         C = (i, x) ⊗ (j, y)              //génération de candidats;
5         L(C) = L(x) ∩_⊗ L(y)             //validation du support;
6         si un candidat C ∈ C, R est fréquent alors
7             [P_x] = [P_x] ∪ C;
8     EnumerateFrequentSubtrees([P_x])
```

Algorithme 11 : EnumerateFrequentSubtrees

vTreeMiner effectue une recherche des structures fréquentes en profondeur d'abord (i.e. recursive). L'algorithme génère tous les arbres fréquents de taille 1 ($\mathcal{F}_1$) puis recherche les fréquents de taille 2 ($\mathcal{F}_2$) et à ce moment il crée le format vertical pour chaque arbre fréquent dans $\mathcal{F}_1$. Pour découvrir les arbres fréquents $\mathcal{F}_k, k \geq 3$ d'une manière récursive, l'algorithme utilise les classes d'équivalence formées lors des étapes $k-1$. La jointure des classes $\mathcal{C} = (i, x) \otimes (j, y)$ est utilisée pour la génération des candidats et la jointure des listes d'étendue notée $\mathcal{L}(\mathcal{C}) = \mathcal{L}(x) \cap_{\otimes} \mathcal{L}(y)$ est utilisée pour décider si un candidat est fréquent. Pour la jointure des listes d'étendue est considérée l'algèbre d'intervalles. Soient $s_x = [l_x, u_x]$ et $s_y = [l_y, u_y]$ l'étendue du nœud x et y respectivement, alors s_x est moindre que s_y, noté $s_x < s_y$, si et seulement si $u_x < l_y$. De plus, s_x est contenu par s_y, noté $s_y \supset s_x$, si et seulement si $l_y \leq l_x$ et $u_y \geq u_x$.

La génération de candidats se fait en ajoutant (j, y) et $(j+1, y)$ à la classe $[P_x]_k$. Si

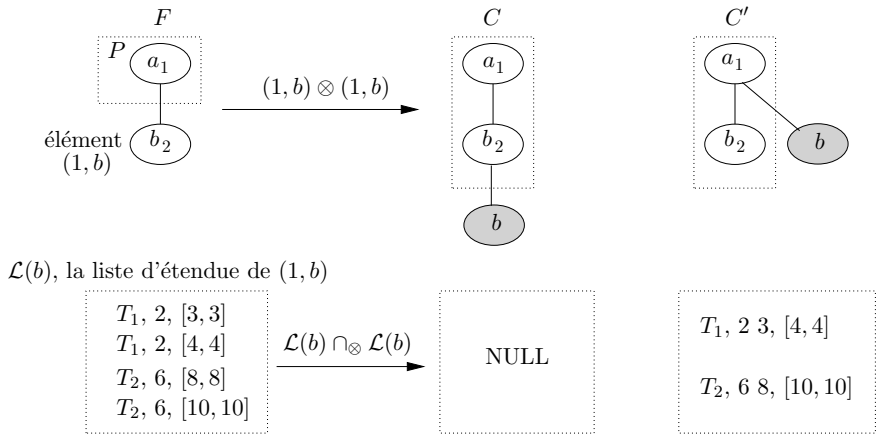


Fig. 3.10 – Validation du support par des listes d'étendue.

on utilise $(j + 1, y)$, cela implique qu'on ajoute un nœud y comme descendant de x, et pour savoir si ce sous-arbre existe dans un arbre T, il faut chercher s'il y a des triplets $(t_y, s_y, m_y) \in \mathcal{L}(y)$ et $(t_x, s_x, m_x) \in \mathcal{L}(x)$ tels que :

1. les deux triplets existent dans le même arbre t $(t_y = t_x = t)$,
2. les nœuds x et y sont extensions de la même occurrence préfixée, avec une même étiquette de correspondance m $(m_y = m_x = m)$, et
3. y est inclus dans l'étendue de x (i.e. $s_y \subset s_x$).

Si ces conditions sont satisfaites, alors nous avons trouvé une structure où y est un descendant de x dans un sous-arbre T. Le candidat (y, j) représente le cas quand y est un frère de x. Pour savoir s'il existe (y, j) dans T, on doit vérifier qu'il existe des triplets $(t_y, s_y, m_y) \in \mathcal{L}(y)$ et $(t_x, s_x, m_x) \in \mathcal{L}(x)$ telles que :

1. t $(t_y = t_x = t)$,
2. m $(m_y = m_x = m)$, et
3. x arrive avant y en considérant un ordre en profondeur d'abord $(s_y < s_x)$.

Exemple 28 *Considérons une classe d'équivalence* $[P]$ *composée par le sous-arbre* P *inclus dans l'arbre fréquent* F *illustré dans la figure 3.10 et l'élément* $(1, b)$*. La jointure* $(1, b) \otimes (1, b)$ *produit les deux arbres candidats* C *et* C'*. La liste d'étendue* $\mathcal{L}(b)$ *de l'élément* $(1, b)$*, représente les occurrences de l'arbre* F *dans la base de données* D *de la figure 3.1 où chaque ligne est interprétée de la manière suivante : le premier élément nous indique la présence de* F *dans le sous-arbre* T_1 *de* D*. La position* 2 *correspond à la position dans* D *du sous-arbre* P*. Finalement, l'étendue* $[3, 3]$ *correspond à la portée de la feuille la plus à droite de* F *(l'élément* $(1, b)$*). Ces listes d'étendue représentent un moyen efficace pour valider le support*

d'un arbre candidat. Pour le premier candidat C construit par la jointure $(1, b) \otimes (1, b)$, la figure montre que ce n'est pas possible de réaliser la jointure des listes $\mathcal{L}(b) \cap_{\otimes} \mathcal{L}(b)$. Dans ce cas le nouveau nœud b attaché à la position 2 doit être un fils de l'élément $(1, b)$. Pour vérifier cette condition, considérons les triplets $(T_1, 2, [3, 3])$ et $(T_1, 2, [4, 4])$ où nous vérifions la présence de deux occurrences du sous-arbre P de F dans l'arbre $T_1 \in D$ avec une même étiquette de correspondance 2. Par contre il n'existe pas de nœud dans T_1 dont la portée est incluse dans la portée de l'élément $(1, b)$, (i.e. $[4, 4] \not\subset [3, 3]$). Ce scénario est répété pour les triplets $(T_2, 6, [8, 8])$ et $(T_2, 6, [10, 10])$. Maintenant considérons le candidat C' où le nouveau nœud est attaché à la position 1, comme un frère de l'élément $(1, b)$. Dans ce cas deux nouvelles listes sont formées de la façon suivante : la première correspond à la jointure des listes dans l'arbre T_1 où on a ajouté la position de l'élément $(1, b)$ à l'étiquette de correspondance de sorte que nouvelle étiquette est 23, et on a ajouté l'étendue du nouveau nœud. La seconde liste est construite de façon similaire. Ces triplets indiquent la présence de deux occurrences de C' dans la base de données, aux positions $2, 3$ et 4, et aux positions $6, 8$ et 10.

3.4. L'algorithme TIDES, une approche PATTERN-GROWTH

Les algorithmes TRIPS[1] et TIDES[2], présentés par Tatikonta dans [TPK06], proposent une technique de recherche d'arbres fréquents basée sur l'approche « Pattern-Growth » [HPY00]. La principale différence entre ces deux algorithmes se trouve dans la façon de coder les arbres pour leur traitement dans le processus de la fouille d'arbres. L'algorithme TRIPS utilise une représentation des arbres basée sur des séquences appelées *codage Prüfer*. Ces séquences ont été introduites par Heinz Prüfer dans [Prü18] où il montre que tout arbre peut être codé par un vecteur (i.e. un $(n-1)$-uplet ordonné d'étiquettes choisies sur un alphabet $[n]$). Dans le codage Prüfer, les arbres sont codés en parcourant les arbres en post-ordre. Par ailleurs, l'algorithme TIDES emploie une représentation des arborescences basée sur un *tableau de pères* similaire au codage proposé indépendamment par Del Razo et al. dans [LLT05, LLT06]. Ce codage se construit en parcourant les arbres en pré-ordre. Dans cette section nous présentons seulement l'algorithme TIDES car cette approche réalise l'extension des arbres par la branche la plus à droite, une méthode formalisée dans la section 2.3.3 et utilisée par Asai et Zaki dans les algorithmes FREQT et TREEMINER respectivement.

TIDES adopte une stratégie qui fait *grandir* les motifs fréquents d'une taille k à une taille $k+1$, cela pour éviter une étape explicite de *génération de candidats*. Cette méthode cherche à ne faire grandir que les arbres qui sont présents dans la base de données, tout en évitant la génération de faux candidats.

[1] de l'anglais *TRee mIning algorithm using Prüfer Sequences*
[2] de l'anglais *Tree mIning algorithm using DEpth first order Séquences*

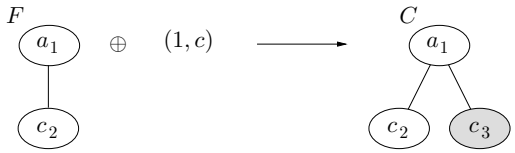


FIG. 3.11 – Un faux candidat C car il n'est pas présent dans la base de données

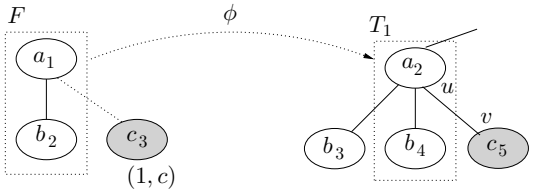


FIG. 3.12 – $(1, c)$, une $(p, l) - extension$ valable de l'arbre fréquent F dans T_1.

Exemple 29 *L'exemple d'un faux candidat est illustré par la figure 3.11 où le candidat C est produit en ajoutant l'extension $(1, c)$ à l'arbre F. Nous pouvons constater, d'après la figure 3.1 qui montre notre base de données exemple, que l'arbre C n'est pas supporté par la base D, $C \not\sqsubseteq D$.*

3.4.1. Algorithme

TIDES est un algorithme récursif où chaque arbre fréquent trouvé à un certain niveau est agrandi en ajoutant un nouveau nœud. Les positions valables auxquelles ce nœud sera ajouté correspondent au chemin qui va de la racine à la feuille la plus à droite de l'arbre fréquent. Cela veut dire qu'on utilise la méthode classique « de la branche la plus à droite » pour la génération d'arbres de taille augmentée de 1.

La méthode d'extraction d'arbres fréquents dans l'approche « Pattern-Growth » est basée sur la technique *diviser pour régner*. Pour compatibilité avec la notation employée dans ce mémoire, nous faisons référence aux (l, p)-*extension points* définis dans [TPK06], comme des $(p, l) - extensions$ définies dans la section 2.3.3. Une $(p, l) - extension$ est dite *valable* si : étant donné une arbre fréquent F avec un mapping ϕ dans un arbre T appartenant à la base de données, il existe un nœud $v \in T$ placé à la droite de ϕ et connecté à un nœud $u \in \phi$.

Exemple 30 *La figure 3.12 illustre une (p, l)-extension valable pour l'arbre F. Le mapping ϕ indique une occurrence de F dans T_1 et nous observons la présence du nœud v connecté et placé à la droite de ϕ. Le nœud v représente une possibilité valable de mapping avec l'extension $(1, c)$ de F.*

En résumé, l'algorithme TIDES travaille de la façon suivante : au début il démarre en parcourant la base pour trouver l'ensemble d'arbres fréquents $\mathcal{F}_1$ (des arbres composés d'un

Entrées : D *//une base de données sur* Σ,
$0 < \sigma \leq 1$ *//un support minimal*
Sorties : $\mathcal{F}$ *//l'ensemble de tous les sous-arbres fréquents dans* D

1 $\langle \mathcal{F}_1, tidList \rangle \leftarrow$ des fréquents de taille 1 et la liste d'arbres dans D supportant $\mathcal{F}_1$;
2 **pour chaque** *nœud* $v \in F_1$ **faire**
3 MINETREES$(\emptyset, (-1, v), tidList)$;

Algorithme 12 : TIDES

seul nœud) et au même temps construit une liste avec les identifiants des arbres dans la base de données supportant l'ensemble $\mathcal{F}_1$. Après, chaque k-arbre fréquent est étendu avec les arbres fréquents projetés sur la base de données pour construire les arbres de taille $k+1$. Ce processus réalise un parcours de l'espace de recherche en profondeur d'abord en employant la fonction recursive MINETREES (c.f. Algorithme 13) et qui reçoit les paramètres suivants : 1) un arbre fréquent F, 2) une (p, l)-extension, et 3) une liste d'arbres dans la base de données supportant l'arbre F.

Entrées : F *//un arbre fréquent de taille* $k-1$,
(p, l) *//une* (p, l)*-extension valable sur* F,
$tidList$ *//la liste d'arbres dans la base supportant* $F \oplus (p, l)$

1 $F' \leftarrow F \oplus (p, l)$;
2 $nwTidList \leftarrow \emptyset$;
3 **pour chaque** *arbre* $T \in tidList$ **faire**
4 **si** (p, l) *est point d'extension valable de* F *dans* T **alors**
5 mis à jour de la liste d'occurrences de T;
6 $nwTidList \leftarrow \bigcup T$;
7 $H \leftarrow \emptyset$;
8 **pour chaque** *arbre* $T \in newTidList$ **faire**
9 Parcourir T, en cherchant des possibles (p, l)-extensions valables;
10 Ajouter les (p, l)-extensions générés à H;
11 **pour chaque** $h \in H$ **faire**
12 **si** $h.support \geq \sigma$ **alors**
13 MINETREES$(F', (h.p, h.l), nwTidList)$;

Algorithme 13 : MINETREES

À l'intérieur de la fonction MINETREES nous pouvons identifier l'initialisation de variables (lignes 1 et 2). Ici l'algorithme fait grandir l'arbre F par la droite en ajoutant un nouveau nœud v représenté par la (p, l)-extension. Une nouvelle variable $nwTidList$ est employée pour enregistrer la liste d'arbres dans D supportant l'arbre étendu F'.

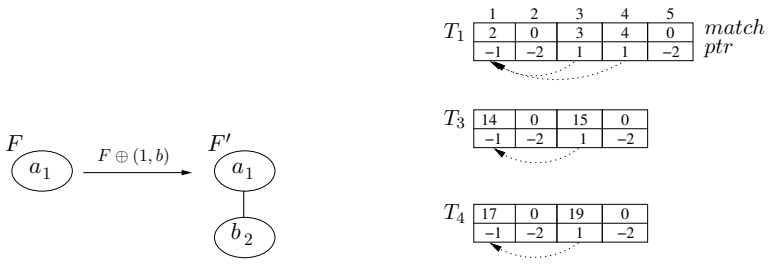
Puis la base de données est balayée (lignes 3-6), pour chercher les (p, l)-extensions valables de F dans D. Si la position p où le nouveau nœud a été ajouté est égal à -1, cela signifie que l'on cherche les occurrences d'un arbre motif F de taille 1 donc l'arbre $F = \emptyset$. En revanche, si la valeur de p est différente de -1 cela veut dire qu'un nœud v a été ajouté à un nœud appartenant à la branche la plus à droite de F. Donc, la recherche des mappings du nœud v dans la base de données doit se réaliser sur tous les nœuds appartenant aux arbres dans la base de données dont la numération en *pré-ordre* est plus grande que v (des nœuds à droite de v). Chaque arbre de la base est parcouru en cherchant des nœuds étiquetés l. Ces ancrages sont stockés dans des vecteurs appelés *listes d'occurrence*[3], des structures de données déjà utilisées dans [NK04, TDH+06b], servant à sauvegarder l'information concernant toutes les occurrences d'un arbre motif F dans un arbre T dans la base de données. Une liste d'occurrence est un vecteur de couples $(match, ptr)$, où $match$ est la position d'un nœud d'un arbre T avec lequel on a établi une correspondance avec un nœud de F et ptr est un pointeur vers le nœud père dans le mapping. Le couple $(0, -2)$ est employé pour séparer les mappings des différents nœuds de F.

Exemple 31 *Pour illustrer le fonctionnement de la fonction* MINETREES, *supposons qu'on ait reçu comme paramètres un arbre* $F \in \mathcal{F}_2$, *un couple extension* $(1, c)$ *et une liste des arbres* $tidList = \{T_1, T_3, T_4\}$ *dans la base de données* D *qui contiennent au moins une occurrence de l'arbre* F. *L'inclusion entre arbres considérée dans cette section est de type* induite.

D'abord cette fonction réalise l'extension de F *vers* $F' = F \oplus (1, c)$ *(c.f. Figure 3.13(a)). Puis les listes d'occurrences du motif* F' *dans les arbres énumérés par la liste tidList, sont mises à jour. La figure 3.13(b) montre les listes d'occurrences de* F' *dans chaque arbre. Pour la racine de* F', *il existe un mapping vers le nœud* 2 *de l'arbre* T_1. *Pour conserver la topologie d'un mapping, l'élément ptr des occurrences est adressé vers la position du père de chaque élément. Dans le cas du mapping de la racine de* F', *ptr est mis à* -1 *car, la racine n'a pas de père. Le couple* $(0, -2)$ *indique le passage aux mappings d'un autre nœud de* F' *donc, il existe une seule occurrence de la racine de* F' *vers l'arbre* T_1. *Ensuite, pour le nœud* 2 *de* F', *il existe deux mappings vers les nœuds* 3 *et* 4 *de* T_1. *L'élément ptr de ces mappings est adressé vers la position* 1 *du vecteur (le père des nœuds* 3 *et* 4*). Les listes d'occurrences des arbres* T_3 *et* T_4 *révèlent la présence d'un mapping des nœuds de* F' *vers les nœuds* $14, 15$ *et* $17, 19$ *respectivement. La nouvelle liste d'occurrences est définie par* $newTidList = \{T_1, T_3, T_4\}$, *les arbres contenant* F'.

Dans une autre étape, la fonction MINETREES parcourt chaque arbre inclu dans *newTidList* (lignes 7-10), en cherchant des nouvelles extensions valables pour la couple (p, l). Ces extensions sont cherchées à partir de la dernière position du mapping correspondant à la feuille la plus à droite de F. La recherche des possibles extensions valables démarre à partir

[3] de l'anglais *embedding lists*.



(a) F', une (p, l)-extension de F (b) Liste d'occurrences de F'

FIG. 3.13 – Des mappings du motif F' dans la base de données illustrée par la figure 3.1

de la dernière position enregistrée dans les listes d'occurrences. Une fois qu'une extension est identifiée, elle est stockée dans une table de hachage, notée H, et un compteur qui représente son support est augmenté seulement si elle n'a pas été trouvée dans un même arbre (lignes 8-10). Finalement si le support d'un élément de la table table de hachage est supérieur au seuil σ établi par l'utilisateur, alors on fait un appel récursif de la fonction MINETREES avec F', le couple (p, l) stocké dans H dont *support* $\geq \sigma$ et la liste d'arbres contenant F'.

Exemple 32 *Pour illustrer la recherche de nouvelles extensions de l'arbre F', considérons ses occurrences dans l'arbre T_1 et T_3 illustrées par la figure 3.14. D'après la liste des occurrences de T_1 (cf. Figure 3.13(b)), nous constatons les occurrences de la feuille la plus à droite de F' aux positions 3 et 4 de T_1. Premièrement, nous parcourons tous les nœuds aux positions pos de T_1 dont pos > 3. Pour chaque nœud parcouru, on vérifie que ce nœud soit un fils d'un nœud v appartenant à la branche la plus à droite d'une occurrence de F' dans T_1. Cette dernière condition est remplie par le nœud 4, alors une nouvelle extension $(1, b)$ est ajoutée à la table H signifiant qu'il est possible de trouver le motif $F' \oplus (1, b)$ dans T_1. Le support de cette extension est initialisé à 1. Puis on vérifie le nœud 5 et on ajoute l'extension $(1, c)$ avec un support 1. Ce parcours est ensuite relancé pour les positions pos > 4 et on trouve une nouvelle fois l'extension $(1, c)$ qui est déjà présente dans H dont le support reste invariable car cette extension est générée par l'arbre T_1. Cela veut dire que le support d'une extension est augmenté sauf si cette extension est générée par des arbres différents. En parcourant l'arbre T_3 à partir des positions pos > 16, on trouve une extension possible $(2, c)$ de F'. La liste d'occurrences de l'arbre T_4 ne produit pas de couples (p, l), car il n'existe pas de nœud placé à droite du nœud 19.*

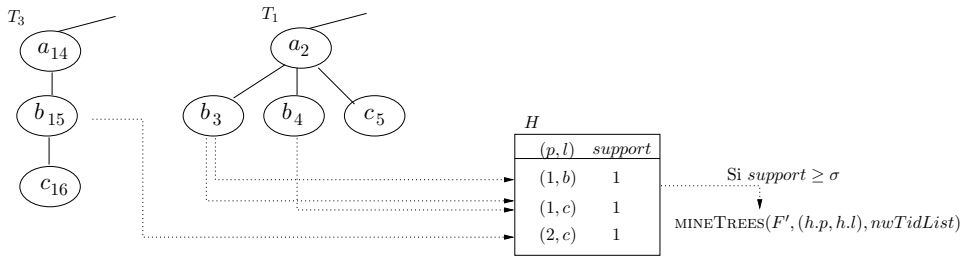


FIG. 3.14 – Des possibles (p,l)-extensions pour F'.

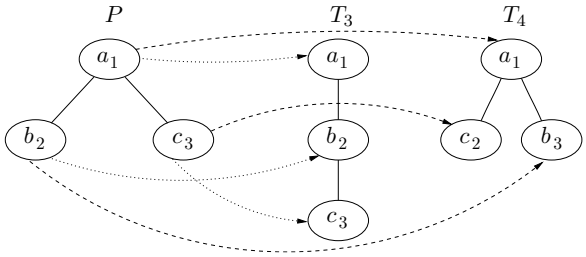


FIG. 3.15 – Inclusion par subsomption $P \sqsubseteq_s T_3$ et $P \sqsubseteq_s T_4$

3.5. L'algorithme TreeFinder, une méthode logique inductive

Dans [TRS02], Termier propose l'algorithme TreeFinder, une technique basée sur la « programmation logique inductive » [FFDB02, Dze03] pour l'extraction d'arbres fréquents. Sa méthode repose sur une définition d'inclusion appelée *inclusion par subsomption*, notée $\sqsubseteq_s$. C'est une définition relaxée qui permet les relations d'ancestralité indirectes. De plus, la fonction de correspondance entre un arbre et un sous-arbre n'est pas nécessairement injective par rapport aux relations d'ancestralité (les arcs). Dans l'inclusion par subsomption, la fonction de correspondance ϕ doit respecter les conditions suivantes :

1. ϕ est injective sur les nœuds,
2. ϕ préserve les étiquettes,
3. ϕ préserve les relations d'*ancestralité* : $\forall u, v \in V_P$, $(u,v) \in E_P \Longrightarrow (\phi(u), \phi(v)) \in E_T^+$.

Exemple 33 *Cette inclusion n'est pas injective sur les arcs (la figure 3.15 illustre cette remarque). Le sous-arbre P est inclus par subsomption dans T_3 ($P \sqsubseteq_s T_3$), considérant des relations indirectes* ancêtre-descendant *entre nœuds et malgré l'inexistence de l'arête $(b,c) \in E_{T_3}$ dans l'arbre P. L'inclusion $P \sqsubseteq_s T_4$, permet les relations directes* père-fils, *sans tenir compte de l'ordre entre les nœuds de T_4.*

Comme Termier montre que son approche peut échouer à extraire tous les arbres fréquents dans la base de données, cette approche est considérée incomplète.

Entrées : D *//une base de données*,
σ *//support minimal*
Sorties : F, *//ensemble des arbres σ-fréquents*

1 $C \leftarrow \emptyset$, $F \leftarrow \emptyset$;
2 $C \leftarrow$ CLUSTERING;
3 $F \leftarrow$ ARBRESMAXIMAUX(C);
4 **retourner** F;

Algorithme 14 : TREEFINDER

Entrées : D *//une base de données*
σ *//un support minimal*
Sorties : $\mathcal{C}$ *//ensemble de clusters*

1 $I \leftarrow \emptyset$, $\mathcal{C} \leftarrow \emptyset$;
2 **pour chaque** *arbre* $T \in D$ **faire**
3 *// La fonction Transaction transforme un arbre T en une transaction. Les items $a * b$ d'une transaction représentent des paires (a, b) d'étiquettes de deux nœuds u et $v \in E_T^+$ tels que u est l'ancêtre de v*;
4 $\mathcal{T} \leftarrow \mathcal{T} \cup$ TRANSACTION(T);
5 *//La fonction Apriori fait le calcul des itemsets fréquents.*;
6 $I \leftarrow$ APRIORI($\mathcal{T}, \sigma$);
7 *//La fonction Clusters retourne les ensemble d'arbres qui supportent chaque itemset*;
8 $\mathcal{C} \leftarrow$ CLUSTERS($\mathcal{I}$);
9 **retourner** $\mathcal{C}$;

Algorithme 15 : CLUSTERING

3.5.1. Algorithme

L'algorithme TREEFINDER est composé de deux phases : 1) étape de *clustering* et 2) l'extraction des arbres communs maximaux (c.f. Algorithme 14 lignes 2 et 3 respectivement).

Dans la première phase, la fonction CLUSTERING (c.f. Algorithme 15) est chargée de mettre chaque arbre T de la base de données sous un format transactionnel. Ce codage représente la fermeture transitive des nœuds de T. Le format est composé par des items $a * b$ tels que a correspond à l'étiquette de l'ancêtre d'un nœud étiqueté b dans l'arbre T. Une fois les arbres transformés, on utilise une des nombreuses techniques pour la recherche d'itemsets fréquents maximaux extraits à partir des codages transactionnels des arbres de la base (e.g. APRIORI [AS94], SPAM [AGTF96], etc.). Les ensembles d'arbres supportant ces itemsets fréquents sont groupés sous le nom de *clusters*.

Exemple 34 *Le tableau 3.1 montre la base de données de la figure 3.1 transformée en un ensemble de transactions. Pour simplifier l'exemple, les relations entre nœuds non-étiquetés*

Tab. 3.1 – Représentation transactionnelle de la base de données D.

$T_1 = \{a * b, a * c\}$
$T_2 = \{a * b, a * c, \ldots\}$
$T_3 = \{a * b, b * c, a * c\}$
$T_4 = \{a * c, a * b\}$

ont été omises. Considérant un seuil minimal $\sigma = 4$ (i.e. un motif doit être inclus dans la totalité des arbres de la base de données), on identifie un seul itemset fréquent $I = \{a * b, a * c\}$. Ensuite, le cluster $C = \{T_1, T_2, T_3, T_4\}$ composé par les arbres de la base supportant l'itemset fréquent I, est crée.

Dans la deuxième phase, la fonction ArbresMaximaux (c.f. Algorithme 16) prend en entrée les clusters produits par l'étape précédente et calcule pour chaque cluster les plus gros arbres inclus dans tous les arbres du cluster. Pour extraire les arbres communs maximaux, TreeFinder propose deux codages relationnels pour un arbre T. Le premier, noté $Rel(T)$, décrit la relation *père-fils* entre les nœuds d'un arbre. Si u et v sont des nœuds appartenant à un arbre T avec des étiquettes a et b respectivement, alors $Rel(T)$ est la conjonction de tous les atomes $ab(u, v)$ tels que $(u, v) \in E_T$. Le deuxième codage, noté $Rel^+(T)$, sert à encoder la fermeture transitive des relations indirectes de parenté dans T. Le codage $Rel^+(T)$ est la conjonction d'atomes $a * b(u, v)$, tels que $(u, v) \in E_T^+$, avec u étiqueté a et v étiqueté b. Le tableau 3.2 montre les codages relationnels pour les arbres T_1, T_2, T_3 et T_4 dans la base de données illustrée dans la figure 3.1.

Entrées : $\mathcal{C}$ *//un ensemble de clusters*
Sorties : $\mathcal{F}$ *//l'ensemble des arbres communs maximaux*

```
1 début
2     F ← ∅;
3     pour chaque cluster C ∈ C faire
4         //rappel C = {T_1, ..., T_m};
5         L ← LGG(Rel+(T_i1)), ..., Rel+(T_im));
6         F ← F ∪ Rel^-1(star^-1(L));
7     retourner F;
8 fin
```

Algorithme 16 : ArbresMaximaux

Les arbres maximaux inclus dans un *cluster* $C_i = \{T_{i_1}, ..., T_{i_m}\}$ sont extraits par la *Least*

TAB. 3.2 – Codages relationnels des arbres dans la base de données illustrée par la figure 3.1.

$Rel(T_1)$	$Rel(T_2)$	$Rel(T_3)$	$Rel(T_4)$
$ab(2,3)$	...	$ab(14,15)$	$ac(17,18)$
$ab(2,4)$	$ac(6,13)$	$bc(15,16)$	$ab(17,19)$
$ac(2,5)$			

(a) Relations *père-fils*

$Rel^+(T_1)$	$Rel^+(T_2)$	$Rel^+(T_3)$	$Rel^+(T_4)$
$a*b(2,3)$	...	$a*b(14,15)$	$a*c(17,18)$
$a*b(2,4)$	$a*b(6,8)$	$b*c(15,16)$	$a*b(17,19)$
$a*c(2,5)$	$a*b(6,10)$	$a*c(14,16)$	
	$a*c(6,12)$		
	$a*c(6,13)$		

(b) Relations *ancêtre-descendant*

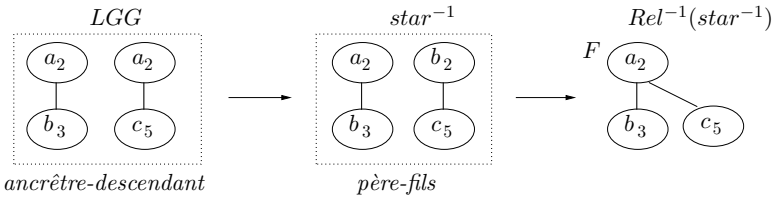


FIG. 3.16 – Décodage en arbre de $star^{-1}$

General Generalization (LGG) [Plo70] des formules relationnelles encodant les arbres :

$$LGG(Rel^+(T_{i_1})), ..., Rel^+(T_{i_m}))$$

La LGG de deux formules relationnelles $Rel(f_1)$ et $Rel(f_2)$ est la formule la plus spécifique qui θ-subsume $Rel(f_1)$ et $Rel(f_2)$. La θ-subsomption est définie de la façon suivante : soient C et C' deux formules de la logique de premier ordre. On dit que C θ-subsume C' s'il existe une correspondance θ des variables de C dans les variables et constantes de C' tel que tout atome de $C\theta$ apparaisse dans C'.

Exemple 35 *La LGG pour les arbres T_1, T_2, T_3 et T_4 de la base de données est :*

$$LGG(Rel^+(T_1)), Rel^+(T_2), Rel^+(T_3)), Rel^+(T_4)) = a*b(2,3) \wedge a*c(2,5).$$

Dans ce cas $Rel^+(T_1)$ θ-subsume $Rel^+(T_2)$, $Rel^+(T_3)$ et $Rel^+(T_4)$ puis que les relations d'ancestralité $a*b$ et $a*c$ à l'intérieur des arbres T_2, T_3 et T_4 se conservent dans T_1.

$LGG(Rel^+(C_{k_1})), ..., Rel^+(C_{k_m}))$ a une structure de forêt et la relation d'ancestralité explicite correspond à une structure de graphe orienté acyclique. La traversée de cette structure suivant un ordre topologique permet de reconstruire les relations *père-fils* à partir des relations *ancêtre-descendant* qui existent dans $LGG(Rel^+(C_{k_1})), ..., Rel^+(C_{k_m}))$. Cette reconstruction est notée $star^{-1}$. Le décodage de $star^{-1}$ en arbre est noté $Rel^{-1}(star^{-1})$.

Exemple 36 *La figure 3.16 illustre :*

– *les relations* ancêtre-descendant *produits par :*
$LGG(Rel^+(T_1)), Rel^+(T_2), Rel^+(T_3)), Rel^+(T_4))$,
– *les relations* père-fils *implicites extraites par :*
$star^{-1}(LGG(Rel^+(T_1)), Rel^+(T_2), Rel^+(T_3)), Rel^+(T_4)))$,
– *l'arbre décodé à partir des relations* père-fils*, produit par :*
$Rel^{-1}(star^{-1}(LGG(Rel^+(T_1)), Rel^+(T_2), Rel^+(T_3)), Rel^+(T_4)))$.

Notons que la LGG ne produit pas de relations ancêtre-descendant *car elles sont inexistantes dans l'arbre* T_1*, alors les relations dans la LGG sont les mêmes que dans* $star^{-1}(LGG)$.

L'arbre F *décodé est un sous-arbre maximal de l'arbre* T_1 *qui* θ*-subsume :*

1. *l'arbre* T_1 *avec une inclusion* induite *(*$F \sqsubseteq_i T_1$*),*
2. *l'arbre* T_2 *avec une inclusion* incrustée *(*$F \sqsubseteq_e T_2$*),*
3. *l'arbre* T_3 *avec une inclusion* par subsomption approximative *(*$F \sqsubseteq_s T_3$*) car elle ne respecte pas l'équivalence entre les arêtes de* F *et* T_3*, et*
4. *l'arbre* T_4 *avec une inclusion* par subsomption non-ordonnée *(*$F \sqsubseteq_s T_4$*) car elle ne respecte pas la relation d'ordre entre nœuds frères de* T_4.

3.6. Conclusion

Quand on s'intéresse à la problématique de la recherche d'arborescences fréquentes, on doit se poser la question : Quel type d'inclusion doit-on prendre en compte pour vérifier si un arbre est inclus dans un autre arbre ?

Dans ce chapitre nous avons présenté les approches existantes qui se différencient principalement par le type d'inclusion considéré. Tout d'abord nous avons présenté deux des principales méthodes de type Apriori pour l'extraction de sous-arbres fréquents Freqt [AAK+02] et TreeMiner [Zak02]. Ces algorithmes ont adapté respectivement le principe Apriori en largeur et en profondeur. L'algorithme Freqt réalise une recherche de sous-arbres fréquents avec une inclusion de type induite. Pour sa part, TreeMiner propose l'extraction de sous-arbres selon une inclusion de type induite et incrustée.

Cherchant à optimiser le processus d'extraction de motif fréquents, J. Han présente l'approche Pattern-Growth dans [HPY00]. Cette approche propose une extraction de motifs sans génération de candidats où les motifs de taille k sont étendus à une taille $k+1$. Les principaux algorithmes d'extraction d'arbres utilisant l'approche Pattern-Growth sont Chopper et XSpanner de C. Wang et al. [WHP+04]. Particulièrement l'algorithme Chopper transforme les arbres sous la forme de séquences qui sont ensuite traitées par l'algorithme PrefixSpan [PHMA+01] afin d'obtenir l'ensemble de séquences fréquentes. Ces dernières, représentant des sous-arbres candidats, sont évaluées par rapport à la base de données. Plus récemment nous trouvons les algorithmes IMB3-Miner proposé par H. Tan et al [TDH+06a], et Trips / Tides proposés par S. Tatikonda [TPK06]. Dans ce chapitre

nous nous sommes intéressés et avons détaillé TIDES car il utilise une représentation des arbres similaire à la structure de représentation que nous proposons dans ce travail (chapitre 4).

Le dernier algorithme exposé dans ce chapitre correspond à TREEFINDER proposé par A. Termier dans [TRS02]. Cependant cet algorithme est reconnu incomplet par son auteur. Dans cette méthode, la relaxation de la contrainte d'ordre entre nœuds frères, la perte de l'injectivité entre les relations *ancêtre-descendant*, la conversion des arbres dans un format transactionnel et l'utilisation de la programmation logique inductive pour l'extraction des arbres maximaux, constituent quelques éléments qui pénalisent cette approche lors du passage à l'échelle. A. Termier propose aussi, dans [Ter04], l'algorithme DRYAL où les motifs fréquents extraits sont des arbres dont aucun nœud ne peut avoir deux fils de même étiquette (une condition qui est cependant peu respectée dans les bases de données réelles). En reprenant la méthode DRYAL, il présente une nouvelle méthode appelée DRYADE [Ter04, TRS04] pour l'extraction de sous-arbres fermés, autrement dit, des sous-arbres contenus dans aucun sous-arbre fréquent ayant le même support.

Les deux chapitres suivants sont destinés à présenter RSF une approche que nous proposons pour la fouille d'arbres. Particulièrement le chapitre 4 est consacré à présenter la structure de données que nous utilisons pour la représentation et le traitement des arbres.

Chapitre 4

Le modèle de données

Le traitement de structures arborescentes par les différentes techniques de fouille d'arbres implique qu'elles soient restructurées et traduites en différents formats de représentation conformément aux besoins de chaque approche. L'objectif dans ce chapitre consiste à proposer une représentation peu coûteuse des structures employées pour représenter les données arborescentes. Cette représentation devra posséder néanmoins des propriétés intéressantes pour améliorer le processus de fouille des arbres.

Sommaire

4.1. Introduction **64**
4.2. Le Modèle Objet de Document (DOM) **65**
4.3. Une structure de données efficace pour la représentation d' arbres **72**
4.4. Conclusion **76**

4.1. Introduction

Dans ce chapitre nous proposons un modèle de données pour représenter des documents semi-structurés XML. Nous considérons un modèle de données comme une collection de descriptions des structures de données, ainsi que les opérations ou les fonctions qui les manœuvrent. Donc, un modèle de données est constitué de :

- Interface de Programmation d'Applications (API)[1]
- Une structure de données.

Ainsi, une API est un ensemble de fonctions ou méthodes appliquées sur une structure de données en particulier. Dans notre contexte, il s'agit de structures de données destinées au traitement des arbres.

Certainement, le choix d'un modèle de données dans toute application informatique dépend du domaine de l'application. Concrètement, dans le domaine de la recherche d'arbres fréquents nous devons choisir un modèle de données dont la structure de données conserve une représentation assez compacte et qui supporte de manière efficiente un ensemble d'opérations API appliqué sur les arbres. En fait, la difficulté de ce choix réside dans la différence entre la complexité spatiale (l'espace de mémoire utilisé) d'une structure de données particulière et la complexité temporelle (le temps d'exécution) des fonctions sur cette structure.

Afin d'illustrer les différents modèles utilisés les plus communément pour traiter les arbres dans le processus de la fouille d'arbres fréquents, nous employons l'arbre $T = (V, E, r, \lambda, \preceq)$ de la figure 4.1, dont les indices (resp. exposants) des nœuds indiquent une énumération en pré-ordre (resp. post-ordre).

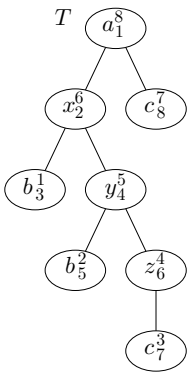


FIG. 4.1 – Un arbre enraciné, étiqueté et ordonné T

[1] de l'anglais *Application Programming Interface*

4.2. Le Modèle Objet de Document (DOM)

Initialement, Charles Golfard, Ed Mosher et Ray Lorie en 1969, ont proposé l'application d'un balisage pour les documents de texte brut dont l'idée principale était la séparation de la structure logique des documents de leur contenu. Cette standardisation appelée GML[2] a par la suite évolué en transposition publiée comme la norme ISO8879-1986 et est connue sous le nom de SGML[3]. Le SGML est chargé de définir les règles de création d'un balisage afin qu'un langage de balisage soit considéré comme une application du SGML. En 1989, Tim Berners-Lee et Anders Berglund ont développé le langage HTML[4] destiné au traitement de documents hypertexte. Ces derniers comprennent un en-tête et un corps composé de : texte, listes, liens, images, formulaires, cadres ou d'autres composants. Afin de résoudre les problèmes liés à l'interopérabilité et à l'évolution du Web, au-delà des limites du HTML, le W3C (*World Wide Web Consortium*) en 1998 a présenté la version 1.0 de la recommandation[5] pour le langage XML[BPSM98]. Le XML est un standard (sous-ensemble du SGML) qui spécifie les règles de la création de nouveaux langages de balises (e.g. les langages basés sur XML parmi lesquels nous pouvons citer : le CML (*Chemical Markup Language*), le ThML (*Theological Markup Language*) ou le MathML (*Mathematical Markup Language*)).

Un document XML est une représentation textuelle des données composée d'éléments, d'étiquettes, d'attributs et de valeurs. Un *élément* est un composant logique d'un document formé *d'étiquettes* entre balises(*tags*) d'ouverture et de fermeture. Les balises d'un élément sont délimitées par le signe < (resp. < /) et le signe > (e.g. <RUE> 643, Av. Prof. Louis Ravas < /RUE>). Le composant compris entre une balise ouvrante et une balise fermante est appelé *valeur*. La valeur peut être vide ou bien elle peut encadrer du texte, d'autres éléments ou les deux à la fois. Un élément peut avoir des informations additionnelles nommées *attributs*. Les attributs sont des couples spécifiés sous la forme $nom = "valeur"$ (e.g. <APPLET $width = "100"$ $height = "200"$ >).

Exemple 37 *La figure 4.2 illustre le texte correspondant à un document XML et sa représentation arborescente.*

Comme nous l'avons déjà vu, dans la section 2.3 les documents XML ont une représentation arborescente. Cette représentation est standardisée dans la recommandation DOM [6][WCH+04]. Ainsi donc, le DOM sert comme un modèle qui représente chacun des aspects des documents XML et définit une API pour accéder aux documents XML.

[2] de l'anglais *Generalized Markup Language*
[3] de l'anglais *Standard Generalized Markup Language*
[4] de l'anglais *HyperText Markup Language*
[5] Une recommandation indique qu'un standard est stable, qu'il contribue à l'interopérabilité du Web, et qu'il a été revu par les membres du W3C.
[6] terme traduit de l'anglais *Document Object Model*

```
<?xml version="1.0" encoding="UTF-8"?>
<library>
  <description>
     Collection de livres.
  </description>
  <book>
    <title>The C Programming Language</title>
    <authors>Brian W. Kernighan,Dennis M. Ritchie</authors>
    <abstract>Le langage C a été développé ...</abstract>
    <contents>
      <chapter>A Tutorial Introduction</chapter>
      <chapter>Types, Operators, and Expressions</chapter>
      <chapter>Control Flow</chapter>
    </contents>
  </book>
  <book>
    <title>The Linux Book</title>
    <authors>David Elboth</authors>
    <contents>
      <chapter>The Linux Book</chapter>
      <chapter>The Operating System</chapter>
    </contents>
  </book>
</library>
```

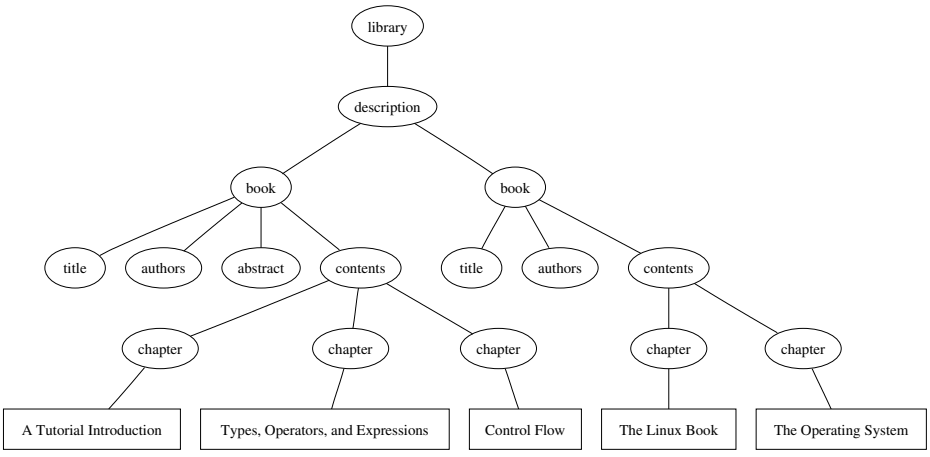


FIG. 4.2 – Un document XML et son arbre de données.

4.2.1. Les composants du DOM

Le modèle de données proposé par le *W3C* considère qu'un document XML est représenté par un *arbre enraciné, étiqueté et ordonné* [FMM+06] appelé *arbre de données*. Les éléments d'un document XML se trouvent imbriqués hiérarchiquement en commençant par un élément racine. À l'intérieur de la structure hiérarchique d'un document XML on peut distinguer les types de nœuds suivants[FMM+06] : les nœuds *document* qui encapsulent des documents XML, les nœuds *élément* correspondant à une balise, les *attributs* désignant les propriétés (et la valeur) des nœuds, les nœuds *texte* contenant des données textuelles, les nœuds *espace de noms* qui fournissent un contexte pour distinguer les noms identiques, mais qui appartiennent à des contextes différents, les nœuds *instructions de traitement* qui apportent des informations additionnelles à une application spécifique pour le traitement du document et finalement les *commentaires* avec des informations additionnelles pour les personnes qui lisent le document.

Interface de Programmation de Applications

Le modèle DOM fournit les fonctionnalités qui permettent grosso modo dans les documents XML de :

- Demander des informations sur le document, créer des documents ou bien parcourir un document. e.g.

  getDocumentElement() : retourne l'élément racine d'un document.

  createAttribute() : crée un attribut.

  createElement() : crée un élément vide.

  createTextNode() : crée un nœud texte.

  ...

- Trouver des informations sur un nœud (ou sur les enfants, le parent ou les frères du nœud) ou bien mettre à jour ou supprimer les informations d'un nœud. e.g.

  getNodeType() : retourne le type implicite d'un nœud.

  getNodeName() : retourne le nom d'un nœud.

  getAttributes() : retourne une liste d'attributs associés à un nœud.

  hasChildNodes() : retourne *vrai* si un nœud a des enfants ou *faux* dans le cas contraire.

  getFirstChild() : retourne *null* ou le premier fils d'un nœud.

  getLastChild() : retourne *null* ou le dernier fils d'un nœud.

  getChildNodes() : retourne une liste d'enfants d'un nœud.

  getParentNode() : retourne le parent d'un nœud.

getNextSibling() : retourne le frère suivant d'un nœud.

...

– Accéder et manipuler les *attributs* des *éléments* ou accéder à la balise des éléments. e.g.

getTagName() : retourne la balise associée à un élément.

getAttribute() : retourne la valeur d'un attribut spécifié.

removeAttibute() : supprime un attribut spécifié.

...

– Manipuler les attributs.

getName() : retourne le nom d'un attribut.

getValue : retourne la valeur d'un attribut.

setValue() : mettre à jour la valeur d'un attribut.

...

Nous avons évoqué ci-dessus l'API définie par le DOM. Nous allons maintenant nous intéresser à la définition d'une API qui constituera une bibliothèque de fonctions destinée au traitement d'arborescences dans le processus d'extraction d'arborescences fréquentes. Nous répertorions par la suite un ensemble de fonctions faisant partie de la API servant à extraire de l'information relative à un arbre T ou à un nœud $u \in T$. Les exemples donnés dans chaque fonction font référence à l'arbre illustré par la figure 4.1.

ancestors() – *Paramètres :* un nœud u, une position optionnelle p. – *Retourne :* a) l'ensemble de nœuds dans le chemin qui va de u à la racine de T si on reçoit le nœud u comme seul paramètre, b) $\emptyset$ si $u = root(T)$ et c) le nœud qui se trouve à la position p dans le chemin qui va de u à la racine de T. (e.g. $ancestors(5) = \{4, 2, 1\}$, $ancestors(5, 3) = 1$).

children() – *Paramètres :* un nœud u. – *Retourne :* a) les nœuds fils de u, b) $\emptyset$ si u est une feuille. (e.g. $children(2) = \{3, 4\}$).

depth() – *Paramètres :* un arbre T ou un nœud u. – *Retourne :* la profondeur de T ou la profondeur d'un sous-arbre de T enraciné au nœud u. (e.g. $depth(T) = 4$, $depth(2) = 3$).

descendants() – *Paramètres :* un nœud u, une position optionnelle p. – *Retourne :* a) l'ensemble des nœuds appartenant au sous-arbre enraciné au nœud u, b) $\emptyset$ si u est une feuille, et c) l'ensemble des nœuds appartenant au sous-arbre enraciné au nœud u dont la profondeur n'est pas plus grande que p. (e.g. $descendants(2) = \{3, 4, 5, 6, 7\}$, $descendants(2, 2) = \{5, 6\}$).

$first()$ – *Paramètres :* un nœud u. – *Retourne :* le fils le plus à gauche de u en considérant que les fils sont ordonnées de gauche à droite. (e.g. $first(2) = \{3\}$).

last() – *Paramètres :* un nœud u. – *Retourne :* le fils le plus à droite de u en considérant que les fils sont ordonnés de gauche à droite. (e.g. $last(2) = \{4\}$).

label() – *Paramètres :* un nœud u. – *Retourne :* l'étiquette $l \in \Sigma$ associée au nœud u. (e.g. $label(2) = x$).

lmb() – *Paramètres :* un arbre T. – *Retourne :* l'ensemble de nœuds dans le chemin allant de la racine de T à la feuille la plus à gauche de T. (e.g. $lmb(T) = \{3, 2, 1\}$).

lml() – *Paramètres :* un arbre T. – *Retourne :* la feuille la plus à gauche de T. (e.g. $lml(T) = 3$).

next() – *Paramètres :* un nœud u. – *Retourne :* le frère suivant de u. (e.g. $next(2) = 8$).

parent() – *Paramètres :* un nœud u. – *Retourne :* a) le parent de u, b) $\emptyset$ si $u = root(T)$. (e.g. $parent(2) = 1$).

previous() – *Paramètres :* un nœud u. – *Retourne :* le frère antérieur de u. (e.g. $previous(8) = 2$).

rmb() – *Paramètres : un arbre T. – Retourne : l'ensemble de nœuds dans le chemin allant de la racine de T au nœud numéroté $|T|$ en pré-ordre..* (e.g. $rmb(T) = \{8, 1\}$).

rml() – *Paramètres :* un arbre T. – *Retourne :* la feuille la plus à droite de T. (e.g. $rml(T) = 8$).

root() – *Paramètres :* un arbre T. – *Retourne :* la racine de T. (e.g. $root(T) = 1$).

siblings() – *Paramètres :* un nœud u. – *Retourne :* a) l'ensemble de frères de u. b) $\emptyset$ si $u = root(T)$. (e.g. $siblings(2) = 8$).

size() – *Paramètres :* un arbre T ou un nœud u. – *Retourne :* la taille de T ou la taille d'un sous-arbre de T enraciné au nœud u. (e.g. $size(T) = 8$, $size(2) = 6$).

Des structures de données

Le DOM ne propose pas de structure de données en particulier pour représenter les arbres. Les arbres dans ce modèle sont des structures logiques qui peuvent être implémentées en choisissant une structure de données la plus adaptée aux besoins. Par la suite nous présentons certaines des structures les plus communément utilisées pour représenter les arbres.

Liste d'adjacences Une liste d'adjacences est une représentation d'un k-arbre T, composée d'un vecteur de k vecteurs liés (lesquels servent à stocker la structure hiérarchique de l'arbre). Chaque élément dans le vecteur correspond à un nœud $u \in V$. La liste associée à un nœud u contient tous les nœuds $v \in V$ tels que $(u, v) \in E$. Pour maintenir la relation d'ordre $\preceq$, il faut que les listes d'adjacences soient ordonnées.

Pour illustrer les représentations des arbres, nous écrivons l'arbre enraciné, étiqueté et ordonné T dans la figure 4.3, dont les indices (resp. exposants) des nœuds de

T indiqués sont énumérés en pré-ordre (resp. post-ordre). La figure montre aussi la représentation de l'arbre T par des listes d'adjacences de l'arbre T.

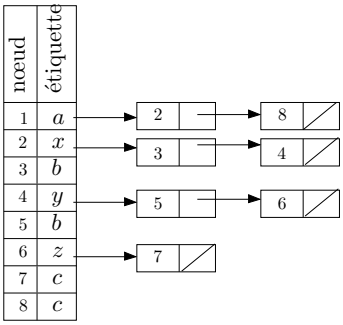


FIG. 4.3 – Représentation par une liste d'adjacence de l'arbre T

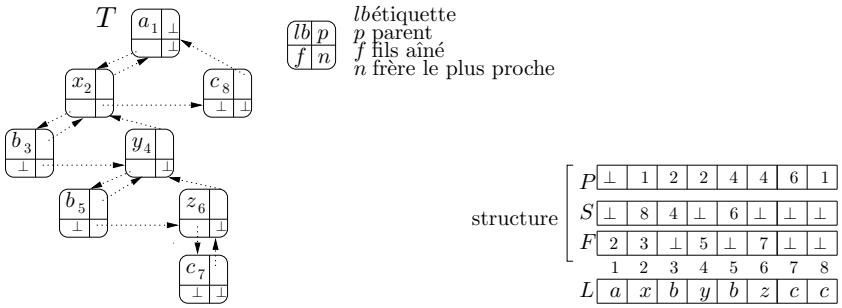
Notons n le nombre de nœuds composant T et $m = n - 1$ le nombre d'arêtes, alors on peut constater qu'une liste nécessite un espace $3n$ pour le vecteur de nœuds (nœud, étiquette, pointeur) plus un espace $2m$ pour les listes d'adjacences (nœud, pointeur)[7]. Donc, l'espace total nécessaire pour cette façon de représenter les arbres est $O(4n-2)$.

L'avantage principal de cette représentation est le temps d'exécution $O(1)$ des opérations suivantes : $root(T)$, $rml(T)$, $first(u)$, $next(u)$, $last(u)$. En revanche, un désavantage est la complexité temporelle $O(n)$ associée aux fonctions suivantes : $ancestors(u)$, $descendants(u)$, $parent(u)$, $children(u)$, $lmb(T)$, $rmb(T)$, $lml(T)$, $size(T)$, $size(u)$, $depth(u)$ et $previous(u)$. Ces dernières fonctions doivent balayer les listes d'adjacences pour retourner les résultats adéquats.

Fils-gauche/frère-droit Une représentation *fils-gauche/frère-droit* est une structure de données où chaque nœud possède : 1) une étiquette, 2) un pointeur vers son père, 3) un pointeur vers le fils aîné et 4) un lien vers son frère le plus proche. Cela revient à créer une liste chaînée pour coder la liste des frères à un niveau donné de l'arbre.

Cette représentation peut être implémentée sous forme de listes chaînées (c.f. Figure 4.4 a)) ou bien par un quadruplet de vecteurs tel que $T = (L, P, F, S)$ où L est destiné à stocker les étiquettes de chaque nœud, P les pointeurs au père de chaque nœud, F sert à garder la référence du première fils d'un nœud et N stocke la référence au nœud suivant. Pour tout nœud $u \in T$, les vecteurs ci-dessus sont définis par : $L[u] = \lambda(u)$, $P[u] = parent(v)$ (dans le cas où $u = root(T)$, alors $P[u] = \emptyset$), $F[u] = first(u)$, (si $first(u) = \emptyset$, alors il s'agit d'une feuille), et finalement $S[u] = next(u)$. La représentation d'un arbre par des vecteurs est illustrée dans la figure 4.4 b). La représentation fils-gauche/frère-droit conduit à quadrupler la taille d'un arbre, c'est-à-dire que cette représentation nous demande un espace $O(4n)$.

[7] L'espace s'incrémente à $3m$ si on traite des listes doublement liées



(a) par une liste chaînée (b) par des vecteurs

FIG. 4.4 – Représentation *fils-gauche/frère droit* de l'arbre T

L'avantage de cette représentation est l'exécution directe $O(1)$ des fonctions suivantes : $parent(u)$, $first(u)$, $next(u)$, $rml(T)$ et $root(T)$. L'inconvénient est sa grande consommation de place mémoire, car elle demande $O(4n)$ en espace. Pour les opérations : $anc(u)$, $des(u)$, $childs(u)$, $last(u)$, $previous(u)$, $lmb(T)$, $rmb(T)$ et $lml(T)$, il est nécessaire de parcourir les vecteurs L, P, F ou S (pour chercher les résultats), donc ces opérations sont exécutées en temps $O(n)$.

Représentation par des séquences de caractères Dans la section 2.3.1 nous avons traité la représentation récursive des arbres en utilisant des parenthèses imbriquées. Dans la séquence de caractères représentant l'arbre T illustrée dans la figure 4.5 nous trouvons 8 parenthèses ouvrantes et 8 fermantes indiquées au-dessus et au-dessous de la chaîne respectivement. Les lignes au-dessous représentent la correspondance entre les parenthèses. Si on construit des couples de parenthèses en respectant l'association indiquée par les lignes on obtient les couples : (1,8), (2,6), (3,1), (4,5), (5,2), (6,4), (7,3) et (8,7) où le premier (resp. deuxième) élément correspond à un parcours de T en pré-ordre (resp. post-ordre) [Knu04]. La suite de couples précédente correspond à la numérotation des nœuds de l'arbre T. L'espace occupé par cette représentation est $O(3n)$.

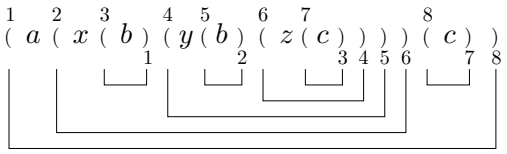


FIG. 4.5 – Représentation d'un arbre T par des parenthèse

Une simple modification à la représentation par parenthèse se fait, en éliminant les parenthèse ouvrantes tout en conservant la structure hiérarchique de T tel que l'illustre

la figure 4.7. Après cette modification, on constate que l'espace demandé par cette représentation est optimisé à $O(2n)$. Pour former une chaîne de caractères représentant l'arbre T, il faut lister les étiquettes des nœuds de T dans l'ordre où ils sont visités par une traversée en pré-ordre. Il est possible de remplacer les parenthèses fermantes) par un symbole spécial (dans ce cas #) pour indiquer la fin de chaque sous-arbre propre de T de telle sorte que la structure de l'arbre soit retenue pour sa reconstruction postérieure.

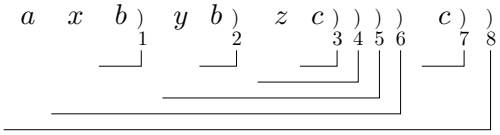


FIG. 4.6 – Représentation de T par des parenthèses fermantes

$$a\ x\ b\ \#y\ b\ \#\ z\ c\ \#\#\#\#\#c\ \#$$

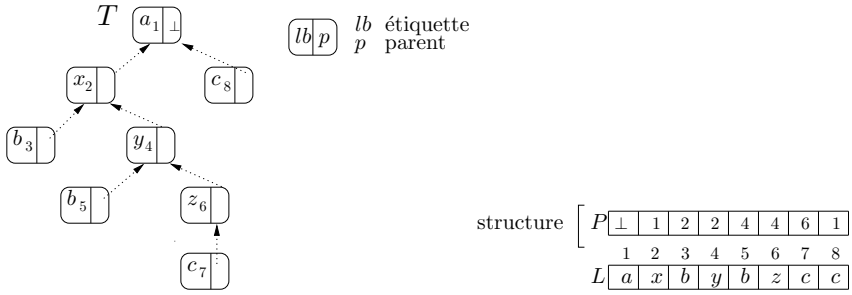
FIG. 4.7 – Représentation d'un arbre T par une chaîne d'étiquettes. Le symbole # n'appartient pas à l'alphabet Σ

La représentation d'arbres par une chaîne de caractères permet de conserver les étiquettes des nœuds ordonnées en pré-ordre, de telle sorte que la première étiquette (un symbole différent du symbole spécial, e.g. #), corresponde à la racine de l'arbre, et la dernière corresponde à la feuille la plus à droite de T. Une autre avantage est qu'une représentation par des caractères permet une comparaison efficace entre les arbres. Cette représentation économise l'espace, mais en revanche elle est pénalisée car elle permet seulement un accès séquentiel à chacune des fonctions définies dans l'API.

4.3. Une structure de données efficace pour la représentation d' arbres

S'il existe dans la littérature des représentations de données arborescentes qui peuvent paraître efficaces en terme de temps d'exécution, celles-ci sont handicapées par l'espace mémoire qu'elles nécessitent. Elles conduisent alors à de fréquents arrêts brutaux dans le processus d'extraction quand la capacité mémoire disponible est dépassée.

Dans le contexte de cette thèse pour la recherche d'arbres fréquents nous avons donc défini une structure de données qui supporte de manière efficace en terme de consommation mémoire l'ensemble d'opérations ou de fonctions appliquées sur les arbres.



(a) Par une liste chaînée (b) Par des vecteurs

FIG. 4.8 – Représentation *fils-gauche/frère-droit* de l'arbre T

4.3.1. Notre proposition

Dans le contexte de la fouille d'arbres, de nombreuses propositions ont été réalisées mais les méthodes de représentation des arborescences sont très souvent trop coûteuses. Dans cette section, nous proposons donc deux structures de données pour représenter les arborescences. Les propriétés de ces représentations peuvent être avantageusement utilisées par les algorithmes de recherche de sous-arbres fréquentes.

4.3.2. Représentation par un tableau de pères

Une manière simple de représenter un arbre est d'associer à chaque nœud un enregistrement contenant un ou plusieurs champs pour coder l'étiquette et un tableau de pointeurs vers les nœuds pères (c.f. Figure 4.8(a)). Pour la représentation d'un arbre T, nous utilisons la propriété suivante : chaque nœud dans l'arbre possède un seul parent. Nous proposons d'utiliser deux vecteurs pour représenter un arbre comme indiqué dans [Wei98]. Le premier, nommé P, conserve la position du père de chaque nœud. Les nœuds de l'arbre sont numérotés en profondeur d'abord, la racine de T correspondant à l'index 1 et ayant une valeur $P[1] = -1$ pour indiquer que la racine n'a pas de père. Les valeurs $P[i], i = 2, 3, ..., k$ correspondent alors aux positions du père des nœuds i, comme illustré sur la figure 4.8(b). Le deuxième vecteur, nommé L, est utilisé pour enregistrer les étiquettes de l'arbre avec $L[i], i = 1, 2, ..., k$ représentant l'étiquette de chaque nœud $u_i \in T$.

Cette représentation permet de retrouver en temps constant le père d'un nœud. De plus, elle permet la localisation directe de la *feuille la plus à droite* par rapport à l'index k. En parcourant l'arbre, on peut bâtir toutes les relations directes *père-fils* entre nœuds.

4.3.3. Représentation binaire

Afin de contrôler des arbres efficacement (comme décrit dans de prochaines sections), chaque arbre T est transformé en une représentation binaire notée T^B où chaque nœud ne peut pas avoir plus de deux enfants [Knu73]. À cette fin, nous proposons la transformation suivante : le premier fils d'un nœud est mis en tant que fils à gauche tandis que les autres enfants sont mis dans le chemin droit.

Une fois que l'arbre a été transformé en arbre binaire, les nœuds doivent être codés afin de pouvoir être recherchés. Le codage est alors employé d'abord pour identifier chaque nœud puis pour déterminer si un nœud est un fils ou un frère. Pour réaliser cette opération, nous considérons l'algorithme de Huffman [CLR90] que nous modifions légèrement afin de l'adapter à nos besoins. Á la racine correspond l'adresse 1. Les adresses des autres nœuds sont calculées en enchaînant l'adresse du père avec :

- 1 s'il s'agit d'un fils (chemin à gauche)
- 0 sinon (chemin à droite)

La structure de données considérée ici est employée afin de représenter les arbres et les candidats. Pour traiter d'énormes quantités de données, nous visons à développer des méthodes qui ne consomment pas beaucoup de la mémoire. Pour cette raison, vu un arbre ayant n nœuds, il ne doit pas être stocké dans plus que $2n$ espace, comme proposé dans [LLT05].

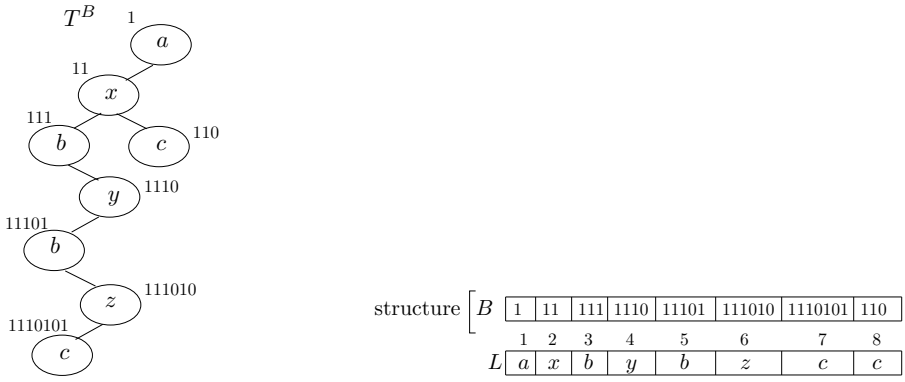
Comme nous avons dit précédemment, la structure que nous proposons stocke un arbre dans une structure de données de taille $2|T|$. Les nœuds d'un arbre sont stockés en suivant un parcours en pré-ordre. Cette structure nous permet de calculer très rapidement si un nœud v est dans portée[8] d'un autre nœud u.

L'adresse binaire de chaque nœud est obtenue de la façon suivante : le premier enfant d'un nœud u est codé par la concaténation de l'adresse de u (i.e. le nœud père) et 1. Tous nœuds frère restants sont codés en utilisant le code binaire du nœud père u auquel un 0 est enchaîné.

Exemple 38 *Le tableau illustré dans la figure 4.9 montre comment l'arbre T de la figure 4.1 est transformé en l'arbre binaire T^B. Le vecteur B de la structure de données stocke le codage de chaque nœud de T^B.*

Dans cette représentation chaque nœud u est associé l'intervalle de positions de ses descendants (i.e. le sous-arbre ayant le nœud u comme racine). Cependant, contrairement à Zaki qui maintient cet intervalle chargé en mémoire, nous n'avons pas besoin de le représenter dans un champ additionnel de la structure de données. [Zak02]. Nous le recherchons plutôt en employant notre structure de représentation de données puisque ce calcul est très efficace en utilisant des opérations binaires.

[8] la portée d'un nœud u est constituée par les nœuds pour lesquels u est un ancêtre



(a) Arbre binaire (b) Par des vecteurs

FIG. 4.9 – Représentation *fils-gauche/frère droit* de l'arbre T

Considérons les codes binaires de deux nœuds u et v . Pour déterminer si le nœud v est un descendant de u, on doit vérifier les conditions suivantes :

- La valeur de v doit être plus grande que la valeur de u.
- Il doit être possible la soustraction de l'adresse du nœud u à partir des premiers bits de l'adresse du nœud v (de gauche à droite). Si les bits de u ne correspondent pas aux premiers bits de v, alors u et v ne peuvent pas être connexes en tant qu'un ancêtre et descendant.
- Si le premier des bits restants du nœud v est 1, alors v est un descendant del u, en cas contraire (la valeur du bit est 0) v n'est pas un descendant de u.

Exemple 39 *Considérons les nœuds u et v codés par* 1110 *et* 1110101 *respectivement. La première condition est bien rempli car* $1110101 > 1110$. *Ensuite, nous cherchons les bits du nœud parent u dans les premiers bits du nœud descendant v. Finalement, comme le bit suivant est* 1, *alors le nœud v est un descendant de u. suivant les indications de la table 4.1.*

TAB. 4.1 – Codage binaire des nœuds 4 et 7 de l'arbre T dans la figure 4.1.

u	1	1	1	0			
v	1	1	1	0	***1***	0	1
	0	1	2	3	4	5	6

Par ailleurs, cette codage est très efficace pour calculer le nombre de nœuds entre un ancêtre et un descendant. En effet, ce nombre de nœuds est donné par le nombre des chiffres de 1 *dans l'adresse du nœud descendant comptés à partir de la adresse du nœud ancêtre.*

Nous pouvons constater cette remarque dans le tableau 4.1. Il y a deux chiffres 1 *à partir de fin de la adresse de u aux positions* 4 *et* 6*, signifiant que deux nœuds apparaître entre u et v.*

4.4. Conclusion

Dans ce chapitre nous nous sommes intéressés à l'élaboration d'une structure de données pour la représentation et le traitement des arborescences. D'abord nous avons proposé deux représentation dynamiques basées sur des pointeurs (les *listes d'adjacence* et *Fils-gauche/-frère-droit*). Les pointeurs dans ces structures de données permettent une grande mobilité à l'intérieur des arbres donnant comme principal avantage une diminution de la complexité temporelle de certaines fonctions API sur les arbres. En revanche ces représentations sont pénalisées par une grosse consommation de mémoire qui peut empêcher le passage à l'échelle des algorithmes de fouille d'arbres.

Ensuite, nous avons présenté des représentations des arborescences sous la forme des séquences de caractères. Cette représentation compacte a été bien exploitée par M. Zaki dans [Zak02] pour l'extension des classes d'équivalence dans le processus de génération de candidats. Cependant, cette représentation rend difficile de trouver les relations directes *père-fils* ou indirectes *ancêtre-descendant* cruciales dans l'étape de validation des candidats.

Pour améliorer ces représentations existantes, à la fin de ce chapitre (section 4.3) nous proposons une nouvelle représentation vectorielle des arborescences. Cette représentation est utilisée soit pour stocker la base de données ou bien pour le traitement des arborescences dans le processus de génération et validation des arbres candidats. Particulièrement dans l'étape de génération de candidats, notre structure de représentation est efficace car elle permet l'extension de candidats *par la branche la plus à droite* tout en ajoutant les nouveaux nœuds à la fin de chaque vecteur représentant une arborescence. Cette dernière propriété est aussi exploitée par la nouvelle méthode de génération de candidats appelée Pivot que nous présentons dans le chapitre suivant. Concernant l'étape de validation des candidats, dans certains cas notre première représentation compacte des arborescences (mémoire) n'est pas efficace par rapport à la vitesse d'exécution (temps). Pour cette raison nous avons donc proposé une transformation des arborescences en codage binaire pour profiter de la rapidité des opérations avec des masques de bits. Ces opérations sur les bits nous permettent en effet notamment de trouver plus rapidement les relations indirectes (*ancêtre-descendant*) entre les nœuds des arbres.

Nous pouvons maintenant étudier comment utiliser ces propriétés de représentation des arborescences dans le processus d'extraction de sous-arbres fréquents. Dans le chapitre suivant nous présentons formellement notre approche, appelée Rsf, pour l'extraction d'arborescences fréquentes.

Chapitre 5

RSF : une approche pour la fouille d'arbres

Dans ce chapitre nous détaillons la mise en oeuvre de notre approche nommée RSF. Dans les sections suivantes nous décrivons plus formellement les algorithmes proposés pour le traitement des arbres enracinés, étiquetés et ordonnés. Concernant la génération de sous-arbres candidats, nous présentons une nouvelle méthode appelée PIVOT. Puis pour la phase de validation des candidats, en prenant en compte les relations verticales entre nœuds d'un arbre, nous considérons différents types d'inclusion. Premièrement nous traitons la feuille d'arbres en considérant une inclusion de type *induite*. Cette inclusion est considérée comme restrictive car elle doit respecter les relations *père-fils*. Deuxièment nous présentons une technique plus relaxée et basée sur l'inclusion *incrustée* qui respecte les relations *ancêtre-descendant* au moment de valider le support d'un arbre dans une base de données. Enfin, nous introduisons une nouvelle approche qui repose sur une inclusion de type *floue*.

Sommaire

5.1. Introduction **78**
5.2. Algorithme RSF **79**
5.3. Génération de candidats **82**
5.4. Validation du support des candidats **95**
5.5. Conclusion **107**

5.1. Introduction

La recherche de structures fréquentes au sein de données arborescentes est une problématique actuellement très active qui trouve de nombreux intérêts dans le contexte de la fouille de données comme, par exemple, la construction automatique d'un schéma médiateur à partir de schémas XML. Plusieurs travaux de recherche récents ont abordé le problème de la *recherche de structures arborescentes fréquentes*[CRNK05].

Dans ce chapitre, nous proposons une nouvelle approche permettant l'extraction de sous-arbres fréquents à partir de bases de données arborescentes. Ces techniques sont groupées sous le nom générique RSF « Recherche de structures fréquentes ».

L'approche RSF utilise le modèle de données exposé dans le chapitre précédent (c.f. Section 4.3.1). Les propriétés de ce modèle sont avantageusement utilisées par nos algorithmes d'extraction de sous-arbres fréquents, en cherchant à optimiser les étapes de génération et validation de candidats. Nous utilisons la représentation *vectorielle* des arborescences pour diminuer les exigences de mémoire dans le processus de fouille, surtout en envisageant le traitement de données à large échelle. Cette représentation des arbres, illustrée par la représentation vectorielle de la base de données dans la figure 5.1, demande un espace mémoire $O(2n)$, où $n = |T|$.

Dans la deuxième section de ce chapitre nous proposons tout d'abord un survol de la méthode RSF destinée à l'extraction de sous-arbres fréquents. L'approche RSF est une méthode basée sur le principe APRIORI [AIS93] un processus qui, de manière générale, se divise en deux étapes à savoir :

- La *génération* de candidats
- La *validation* du support des candidats

En suivant cet ordre, nous consacrons la troisième section à la présentation de nos algorithmes pour la génération d'arbres candidats. Dans cette section nous présentons dans un premier temps un algorithme basé sur la méthode traditionnelle de génération que nous appellerons *par la branche la plus à droite*. Dans un deuxième temps nous exposons une nouvelle méthode de génération de candidats, que nous avons nommée PIVOT. Traditionnellement les différentes propositions reposent sur des approches de type "*générer-élaguer*" (e.g. TreeMiner) et se retrouvent confrontées au problème de la génération d'un trop grand nombre de candidats à vérifier sur la base. Ici, nous proposons une nouvelle approche qui réduit considérablement le nombre de sous arbres générés. L'originalité de notre approche réside dans l'utilisation d'un arbre *pivot* qui permet de définir des classes d'équivalence d'arbres et de valider plus finement que *toutes* les sous-parties d'un arbre doivent être fréquentes pour que celui-ci mérite l'appellation de *candidat*.

La quatrième section est dédiée à la phase de validation de candidats dans le processus de la fouille d'arbres. Ici, nous proposons trois algorithmes dans le but de valider l'inclusion d'un arbre dans une base de données. La différence entre ces algorithmes est déterminée par

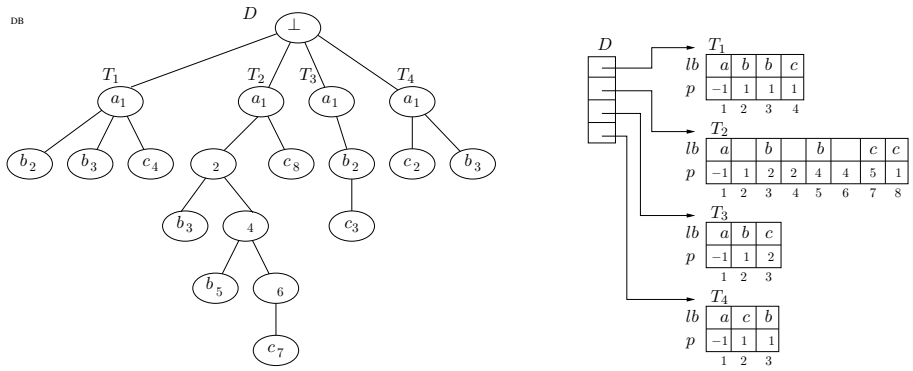


FIG. 5.1 – Une base de données et sa représentation vectorielle

le type d'inclusion (i.e. *induite, incrutée ou floue*) entre deux arbres.

Nous considérons une base de données arborescente $\mathcal{D} = (D, \Sigma_{\mathcal{D}}, \lambda_{\mathcal{D}})$ où l'arbre de données D est défini par la forêt $D = \{T_1, \ldots, T_n\}$. Pour simplifier les notations nous autorisons la notation $T \in D$ indiquant qu'un arbre T appartient à une base de données. Les nœuds d'un k-arbre T sont numérotés en suivant un parcours en pré-ordre tel que l'ensemble de nœuds de T est défini par $V = \{1, \ldots, k\}$. La figure 5.1 illustre une base de données sur laquelle nous baserons nos exemples.

5.2. Algorithme RSF

Comme nous l'avons observé précédemment, notre approche d'extraction est basée sur une approche classique de type « générer-élaguer » (i.e. à chaque étape, nous générons différents candidats et nous testons s'ils sont inclus dans la bases d'arbres).

L'algorithme RSF (c.f. Algorithme 17) fonctionne de la manière suivante : un premier parcours sur la base est réalisé pour extraire les arbres dont le nombre d'occurrences est supérieur au support minimal défini par l'utilisateur σ. Nous obtenons ainsi l'ensemble $\mathcal{F}_1$ des arbres fréquents de taille 1 (ligne 1). Ces derniers sont combinés entre eux pour former des candidats de taille 2 et un parcours sur la base permet d'obtenir l'ensemble $\mathcal{F}_2$ constitué des arbres de taille 2 (ligne 3). L'algorithme se poursuit en générant des candidats de taille $k+1$ et en effectuant un parcours sur la base pour compter le nombre d'occurrences de chaque candidat. Lorsque plus aucun candidat ne peut être généré l'algorithme se termine (lignes 4-15). Notons que cet algorithme reprend les principes de la littérature [AIS93, AAK+02, Zak02].

Étudions à présent comment les sous-arbres fréquents de taille 1 et 2 sont obtenus. Les 1-sous-arbres fréquents sont extraits par l'algorithme FREQUENTS1 (c.f. Algorithme 18). Dans cette fonction, les candidats de taille 1 sont tout d'abord proposés à partir de l'alphabet Σ, c'est-à-dire, les candidats à cette étape constituent des arbres résumés à un

Entrées : Σ *//un alphabet*,
$\mathcal{D}$ *//une base de données sur* Σ,
σ *//un support minimal*
Sorties : $\mathcal{F}$ *//l'ensemble de tous les sous-arbres fréquents dans D*

```
1  F_1 ← FREQUENTS1(Σ, D, σ);
2  F_2 ← FREQUENTS2(F_1, D, σ);
3  k ← 2;
4  tant que F_k ≠ ∅ faire
5      C_{k+1} ← GENCANDIDATS(F_k, F_2)        //génération des candidats;
6      pour chaque candidat C ∈ C_{k+1} faire
7          sup ← 0;
8          pour chaque arbre T ∈ D faire
9              //validation de l'inclusion (C ⊑ T);
10             si INCLUSION(C, T) alors
11                 ++sup;
12                 si sup ≥ σ alors
13                     F_{k+1} = ∪{C};
14                     couper                  //passage au candidat suivant;
15     k ← k + 1;
16 retourner F = F_1 ∪ F_2 ∪ ... ∪ F_{k-1};
```

Algorithme 17 : RSF : **R**echerche de **S**tructures (des sous-arbres) **F**réquentes

seul nœud (i.e. un arbre pour chaque étiquette $l \in \Sigma$). A cet effet nous initialisons un conteneur C avec les étiquettes $l \in \Sigma$ (ligne 1). Dans la mise en oeuvre réelle on utilise un conteneur de type *vector* car les étiquettes des arbres sont représentées par des valeurs entières. Dans ce mémoire on utilise des étiquettes représentées par des caractères pour établir une différence entre les étiquettes et la numérotation en préordre ou postordre des nœuds, afin de rendre plus compréhensibles les exemples. Chaque étiquette (ligne 4), voit son support augmenté lors du parcours de la base de données et seules sont conservées les étiquettes dont le support est supérieur au support σ (lignes 2-6). Nous rappelons que la fonction $\lambda(u) = l \in \Sigma$ retourne l'étiquette du nœud u.

Pour l'extraction des fréquents de taille 2 nous utilisons la fonction FREQUENTS2 détaillée par l'algorithme 19. Celle-ci procède à la création des candidats de taille 2 notés $\mathcal{C}$, obtenus en combinant deux à deux tous les fréquents de taille 1 tels que $\mathcal{C} = \mathcal{F}_1 \times \mathcal{F}_1$ (ligne 1). Chaque arbre candidat de taille 2 représente une relation *père-fils* entre deux nœuds. Cette relation est notée (l_1, l_2) où le premier élément correspond à l'étiquette du père et le deuxième correspond à l'étiquette du fils. Afin de valider le support des candidats en un simple balayage de la base de données, nous utilisons une matrice, notée S, de taille $\mathcal{F}_1 \times \mathcal{F}_1$

Entrées : Σ *//un alphabet*,
$\mathcal{D}$ *//une base de données sur* Σ,
σ *//un support minimal*
Sorties : $\mathcal{F}_1$ *//l'ensemble des 1-sous-arbres fréquents supportés par* $\mathcal{D}$

1 $\mathcal{C} \leftarrow \Sigma$, $\mathcal{F}_1 \leftarrow \emptyset$;
2 **pour chaque** *arbre* $T \in \mathcal{D}$ **faire**
3 **pour chaque** *nœud* $u \in T$ **faire**
4 $++\mathcal{C}[\lambda(u)]$;
5 **pour chaque** *étiquette* $l \in \Sigma$ **faire**
6 **si** $\mathcal{C}[l] \geq \sigma$ **alors** $\mathcal{F}_1 = \cup\{(l)\}$;
7 **retourner** $\mathcal{F}_1$;

Algorithme 18 : FREQUENTS1 : l'ensemble des 1-sous-arbres fréquents

Entrées : $\mathcal{F}_1$ *//l'ensemble de* 1*-sous-arbres fréquents*,
$\mathcal{D}$ *//une base de données sur* Σ,
σ *//un support minimal*
Sorties : $\mathcal{F}_2$ *//l'ensemble des 2-sous-arbres fréquents supportés par* $\mathcal{D}$

1 $\mathcal{C} \leftarrow \mathcal{F}_1 \times \mathcal{F}_1$, $S \leftarrow \mathcal{F}_1 \times \mathcal{F}_1$, $\mathcal{F}_2 \leftarrow \emptyset$;
2 **pour chaque** *arbre* $T \in \mathcal{D}$ **faire**
3 **pour chaque** *nœud* $u \in V \setminus r$ **faire**
4 $++S[\lambda(parent(u))][\lambda(u)]$;
5 **pour chaque** *couple* $(l_1, l_2) \in \mathcal{C}$ **faire**
6 **si** $S[l_1][l_2] \geq \sigma$ **alors** $\mathcal{F}_2 = \cup\{(l_1(l_2))\}$;
7 **retourner** $\mathcal{F}_2$;

Algorithme 19 : FREQUENTS2 : l'ensemble des 2-sous-arbres fréquents

pour enregistrer chaque occurrence des relations *père-fils*. On parcourt chaque nœud appartenant aux arbres dans la base (sauf la racine car elle n'a pas de père) et on augmente son support (lignes 2-4). Pour trouver le support de chaque candidat, il suffit de vérifier le support stocké dans S selon les coordonnées $[l_1][l_2]$. Évidemment, nous ne conservons que les couples dont le support est supérieur au seuil σ (lignes 5-6).

Exemple 40 *Considérons la base de données illustrée par la figure 5.1 et un support* $\sigma = 2$. *Après le parcours de la base de données, on obtient l'ensemble* $\mathcal{F}_1 = \{(a), (b), (c)\}$. *Ensuite, après un deuxième balayage de la base, on constate que l'ensemble des sous-arbres fréquents de taille* 2 *est* $\mathcal{F} = \{(a(b)), (a(c))\}$. *Nous employons une représentation par des parenthèses afin de pouvoir énumérer facilement tous les arborescences fréquentes résultantes.*

Notons que l'extraction des sous-arbres fréquents de taille 1 et 2 sera commune à tous

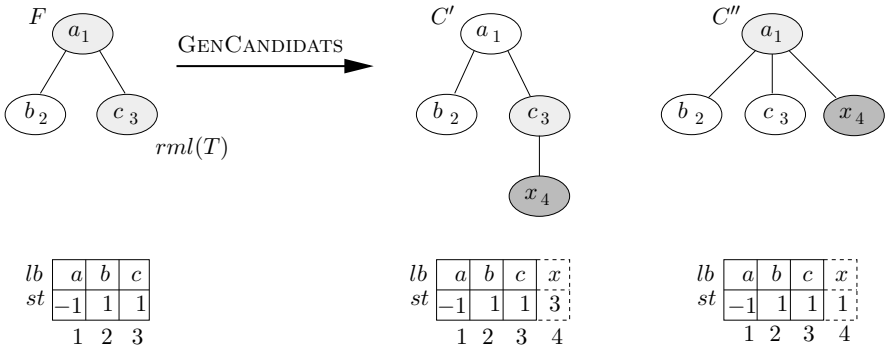


FIG. 5.2 – Génération des candidats

les types d'inclusion présentés ci-après (i.e. induite, incrustée, floue) sur les arbres.

5.3. Génération de candidats

5.3.1. Génération de candidats par la branche la plus à droite

La génération des candidats de taille $k \geq 3$ s'effectue de la même manière que dans les approches classiques de type Apriori par combinaison des fréquents de taille $k-1$. Nous adoptons la stratégie de génération de candidats selon la branche la plus à droite comme proposée dans la section 2.3.3 et montrée dans la figure 5.2. Cette stratégie demande le parcours de tous les nœuds appartenant à la branche la plus à droite $rmb(T)$ d'un arbre T. La représentation par des vecteurs nous permet d'obtenir directement la $rmb(T)$. Pour cela, on obtient d'abord la position de la feuille la plus à droite $rml(T)$ de l'arbre (i.e. elle est placée à l'extrémité droite du vecteur). Puis, il suffit de se déplacer en suivant la trace des pères de chaque nœud pour obtenir $rmb(T)$ (i.e. l'ensemble d'ancêtres de $rml(T)$).

Nous pouvons ainsi constater l'intérêt de notre structure de représentation puisque, naturellement, il suffit d'ajouter un nouvel élément dans la représentation de l'arbre en spécifiant le père du nouveau nœud. La méthode de représentation des arbres que nous proposons permet de générer de manière efficace les sous-arbres candidats puis d'élaguer les sous-arbres non fréquents (après calcul du support).

L'algorithme GENCANDIDATS (c.f. Algorithme 20) décrit la génération des candidats en utilisant la branche la plus à droite des sous-arbres fréquents de taille k afin de proposer des candidats de taille $k+1$. Pour chaque arbre fréquent de taille k, il génère un nouveau candidat en étendant l'arbre par chaque ancêtre de la feuille la plus à droite. Ainsi, pour chaque nœud, nous ajoutons les seules extensions possibles, i.e. celles qui s'avèrent fréquentes dans $\mathcal{F}_2$.

Entrées : $\mathcal{F}_k$ *//des sous-arbres fréquents de taille* k,
$\mathcal{F}_2$ *//sous-arbres fréquents de taille 2*
Sorties : $\mathcal{C}_{k+1}$ *//Un liste de candidats de taille* $k+1$

```
1 C_{k+1} ← ∅;
2 pour chaque arbre F ∈ F_k faire
3     pour chaque nœud u ∈ rmb(F) faire
4         pour chaque arbre (l1(l2)) ∈ F_2 faire
5             si λ(u) == l1 alors
6                 p ← pre(u);
7                 C ← F ⊕ (p, l2);
8                 C_{k+1} ← ⋃C;
9 retourner C_{k+1};
```

Algorithme 20 : GenCandidats : génération de candidats

5.3.2. Pivot : une nouvelle méthode pour la génération de candidats

Les approches traditionnelles de fouille d'arbres permettent d'obtenir des sous-structures ou des sous-arbres fréquents en utilisant des stratégies de type « générer-élaguer ». Même si elles sont efficaces, elles sont cependant pénalisées par le fait que le nombre de candidats générés est trop important. Dans cette section, nous proposons une nouvelle approche, appelée Pivot qui réduit considérablement le nombre de sous-arbres candidats générés.

Notre solution est fondée sur la construction de classes d'équivalence, elles-mêmes définies à partir de l'arbre *pivot* qui rassemble les arbres de la classe. L'originalité de notre approche consiste à considérer trois types d'arbres pivots : gauche, droit ou racine. Un *arbre pivot* gauche (resp. droit), pour un arbre T, correspond à l'arbre constitué de tous les nœuds sauf la feuille la plus à gauche (resp. droite) de T. Pour un arbre dont la racine n'a pas plus d'un fils, on parlera d'arbre *pivot racine*. A tout ensemble d'arbres de taille k, nous associons l'ensemble des pivots qui les régissent pour construire des classes d'équivalence. A partir de celles-ci, nous pouvons alors construire les candidats de taille $k+1$ en combinant deux arbres partageant le même pivot et en insérant les deux nœuds spécifiques, de manière très similaire aux méthodes par niveau dans le cadre de la fouille de données (*e.g.* recherche d'itemsets). Par cette méthode, nous réduisons le nombre de candidats générés en nous assurant que tout pivot d'un arbre candidat est fréquent. En outre, le nombre de candidats est complet dans la mesure où il ne peut pas exister de fréquent dans la base qui n'ait pas été généré.

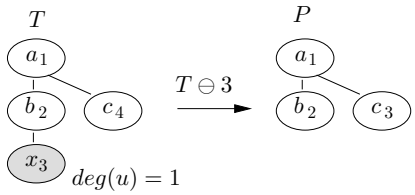


FIG. 5.3 – Pivot P obtenu en enlevant le nœud u de l'arbre T.

Des opérateurs de suppression et d'ajout

Dans cette partie nous proposons l'opérateur $\ominus$ et nous modifions l'opérateur $\oplus$ qui servent de base à notre approche.

Soient un arbre T et une position p en pré-ordre d'un nœud $u \in V$ tel que $deg(u) = 1$, nous définissons l'opérateur $\ominus$ sur les arbres tel que $T \ominus p$ retourne le sous-arbre P de taille $k-1$ inclus dans T auquel a été supprimé le nœud u placé à la position p. Le sous-arbre résultant P est nommé *pivot*.

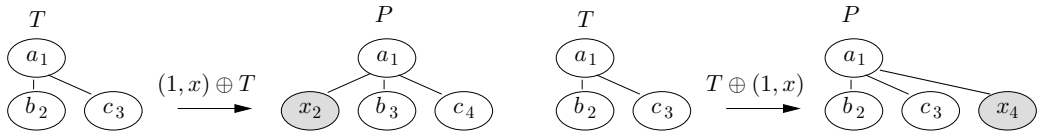
Exemple 41 *La figure 5.3 illustre le fonctionnement de l'opérateur $\ominus$. L'arbre pivot P dans cette figure est obtenu en enlevant le nœud placé à la position p (la feuille la plus à gauche de T) dont le degré est égal à 1.*

Remarque 1 *L'opérateur $\ominus$ ne peut s'appliquer que sur des arbres dont le degré du nœud supprimé est égal à 1 afin d'empêcher l'obtention d'une forêt à l'issue de l'opération.*

En contrepartie à l'opérateur de suppression, nous redéfinissons l'opérateur d'insertion ou ajout déjà utilisé dans la section 2.3.3. Cet opérateur permettra d'ajouter un nouveau nœud à gauche ou à droite d'un arbre. Soient un arbre T de taille k et un couple (p, l), l'opérateur $\oplus$ est défini, selon la valeur de p, de la façon suivante :

- Si $p = -1$ alors -1 indique que le nouveau nœud sera ajouté comme la racine de l'arbre T. Dans ce cas avec l'utilisation de $(-1, l) \oplus T$ ou $T \oplus (-1, l)$ ont obtient le même résultat.
- Si $1 \leq p \leq k$,alors :
  - $T \oplus (p, l)$ insère un nouveau nœud étiqueté l comme *dernier fils* du nœud se trouvant à la position p dans T,
  - $(p, l) \oplus T$ insère un nouveau nœud étiqueté l comme *premier fils* du nœud se trouvant à la position p dans T.

Exemple 42 *La figure 5.4 illustre le fonctionnement de l'opérateur d'insertion. Le nouveau nœud est ajouté selon la position de la (p, l)-extension par rapport à l'opérateur $\oplus$.*



(a) à gauche (b) à droite

FIG. 5.4 – Insertion d'un nouveau nœud à l'arbre T

Des classes d'équivalence à droite

Soit $\mathcal{F}_k$, l'ensemble des arbres fréquents de taille k . Nous définissons la fonction $\delta(F) = F \ominus rml(F)$, qui enlève la feuille la plus à droite d'un arbre $F \in \mathcal{F}_k$, et retourne un arbre pivot P_δ de taille $k-1$. Cette fonction permet d'établir une relation d'équivalence, notée $\mathcal{P}_\delta$, entre deux arbres F et $F' \in \mathcal{F}_k$:

$$F \mathcal{P}_\delta F' \Longleftrightarrow \delta(F) = \delta(F')$$

Preuve. Notons qu'il est trivial de montrer que $\mathcal{P}_\delta$ est une relation d'équivalence puisqu'elle est fondée sur une égalité ($\mathcal{P}_\delta$ est donc réflexive, symétrique et transitive). □

Deux arbres F et F' appartiennent à la même *classe d'équivalence à droite* si et seulement s'ils partagent une structure commune (un arbre pivot P_δ), sauf pour la feuille la plus à droite. Cette classe d'équivalence est définie par :

$$\mathcal{P}_\delta(F) = \{F' \in \mathcal{F}_k | F \mathcal{P}_\delta F'\}$$

Une classe d'équivalence dont l'arbre pivot est P_δ, est notée $[P_\delta]$. Notons également que tout arbre $F \in \mathcal{F}_k$ appartient à une et une seule classe d'équivalence $[P_\delta]$ et peut donc être représenté à partir de son pivot et de la feuille la plus à droite pour laquelle on donnera l'étiquette et la position dans P_δ. Une classe d'équivalence à droite $[P_\delta]$ peut être alors représentée par un arbre pivot P_δ et une liste de couples $R = \{R[1], \ldots, R[|R|]\}$.

Dans la liste, chaque couple $(R[i]_1, R[i]_2)$ représente un arbre F appartenant à une classe d'équivalence dont l'arbre pivot est P_δ. Ainsi donc, chaque couple représente une (p,l)-extension d'un pivot avec la quelle nous pouvons reconstruire un arbre fréquent F. Le premier élément $R[i]_1$ représente alors une position p dans l'arbre pivot où on va ajouter un nœud d'étiquette $R[i]_2$. Une classe d'équivalence dont l'arbre pivot est P_δ sera représenté par :

$$[P_\delta] \sim \{P_\delta, R\}$$

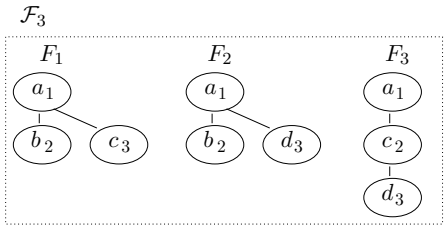


FIG. 5.5 – Des arbres fréquents de taille 3

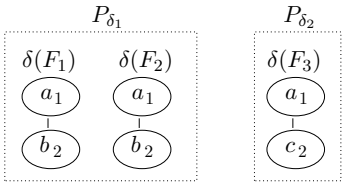


FIG. 5.6 – Des arbres *pivots* à droite

Nous rappelons que pour obtenir les arbres candidats $\mathcal{C}_{k+1}$ à partir des classes d'équivalence à droite, Zaki à proposé dans [Zak02] l'utilisation d'un opérateur de jointure (produit) afin d'obtenir une extension des classes d'équivalence.

Soit une classe d'équivalence $[P_\delta] \sim \{P_\delta, R\}$ et deux arbres F, F' tels que $F\mathcal{P}_\delta F'$ et soient $(p,l),(p',l')$ deux couples dans R tels que $F = P_\delta \oplus (p,l)$ et $F' = P_\delta \oplus (p',l')$. Zaki applique l'opérateur de jointure (c.f. Définition 1), noté $(p,l) \otimes (p',l')$, sur les deux couples afin d'obtenir une nouvelle classe d'équivalence $[F]$:

- **si** $(p == p')$ **alors**
  $[F].R+ = \{(l', parent(rml(F))), (rml(F), l')\}$
- **si** $(p > p')$ **alors**
  $[F].R+ = \{(parent(rml(F))), l')\}$

L'opérateur $\otimes$ est appliqué à tous les éléments dans la liste R y compris à lui-même.

Exemple 43 *Considérons l'ensemble d'arbres fréquents de taille 3, noté $\mathcal{F}_3$, illustré par la figure 5.5. Nous appliquons la fonction δ sur l'ensemble $\mathcal{F}_3$ en éliminant la feuille la plus à droite de chaque arbre tel que : $\delta(F_1) = F_1 \ominus rml(F_1)$, $\delta(F_2) = F_2 \ominus rml(F_2)$ et $\delta(F_3) = F_3 \ominus rml(F_3)$. Comme $\delta(F_1) = \delta(F_2)$, alors il s'agit d'un pivot commun noté P_{δ_1}. Par ailleurs, le pivot obtenu à partir de F_3 est identifié par P_{δ_2}. Les arbres pivots P_{δ_1} et P_{δ_2} sont illustrés par la figure 5.6. Ces arbres pivots permettent la création des classes d'équivalence suivantes :*

$$[P_{\delta_1}] \sim \{P_{\delta_1}, \{(1,c),(1,d)\}\}$$
$$[P_{\delta_2}] \sim \{P_{\delta_2}, \{(2,d)\}\}$$

Notons que la première classe $[P_{\delta_1}]$ *regroupe les arbres fréquents* F_1 *et* F_2 *qui peuvent être reconstruits par les* (p,l)*-extensions* $(1,c)$ *et* $(1,d)$*, c'est-à-dire* $F_1 = P_{\delta_1} \oplus (1,c)$ *et* $F_1 = P_{\delta_1} \oplus (1,d)$*.*

Ces candidats ont comme arbre pivot commun l'arbre fréquent F_1*. La jointure des couples dans les classe* $[P_{\delta_1}]$*, génère les deux nouvelles classes suivantes :*

$$[F_1] \sim \{F_1, \{(1,c),(3,c),(1,d),(3,d)\}\}$$
$$[F_2] \sim \{F_2, \{(1,d),(3,d),(1,c),(3,c)\}\}$$

La classe d'équivalence qui a comme pivot F_3 *est obtenue en employant l'opérateur de jointure sur les couples de* $[P_{\delta_2}]$*.*

$$[F_3] \sim \{F_3, \{(2,d),(3,d)\}\}$$

Les arbres candidats $C_1, C_2, ..., C_{10}$ *illustrés dans la figure 5.7, sont obtenus par la combinaison du pivot de chaque classe d'équivalence* $[F_1]$*,* $[F_2]$ *et* $[F_3]$ *avec chaque couple de la liste* R *dans ces classes. Ainsi donc, l'arbre candidat* C_1 *est généré par* $C_1 = F_1 \oplus (1,c)$*,* $C_2 = F_1 \oplus (3,c)$*, etc.*

A chaque niveau k de l'algorithme, les candidats de niveau $k+1$ sont générés en calculant le produit de tous les couples d'éléments de taille $k-1$ dans chacune des classes d'équivalence (partageant donc leurs $k-2$ premiers nœuds). Nous obtenons ainsi toutes les combinaisons possibles d'arbres compatibles au sens du pivot droit.

Cependant l'utilisation d'une seule relation d'équivalence pour la génération de candidats produit de très nombreux candidats non valables (dont on sait déjà qu'ils ne seront pas fréquents). Nous pouvons constater que seul le sous- arbre C_4 dans la figure 5.7 est un candidat valable car à l'intérieur des autres arbres, il existe des sous-arbres de taille 3 (en gros) qui n'appartiennent pas à l'ensemble $\mathcal{F}_3$ (c.f. Figure 5.5). Alors, par la propriété d'anti-monotonie on sait a priori que ces arbres ne sont pas fréquents.

Nous proposons donc de nous appuyer sur plusieurs relations d'équivalence afin de réduire le nombre de candidats générés tout en conservant une approche complète. Nous nous basons à la fois sur la relation d'équivalence à droite, mais également sur les relations d'équivalence à gauche et *racine*. Nous proposons donc d'adopter une stratégie de génération de candidats alternative basée sur trois classes d'équivalence permettant de manipuler un sous-ensemble de candidats plus restreint mais complet.

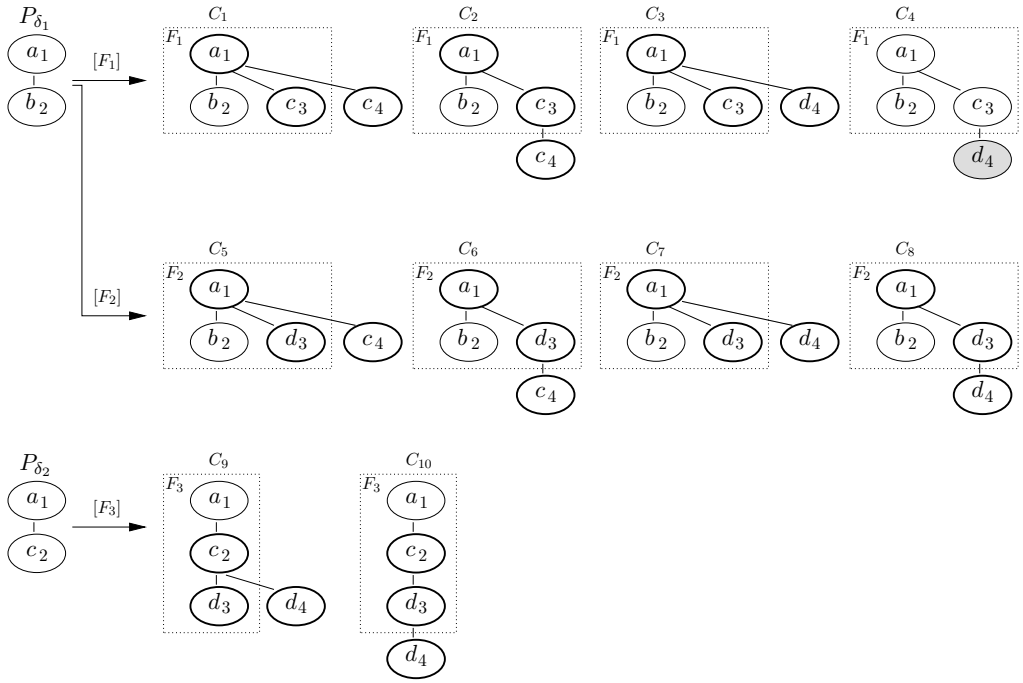


FIG. 5.7 – Des arbres candidats de taille 4 obtenus par l'extension de classes d'équivalence $[P_{\delta_1}]$ et $[P_{\delta_2}]$

Des classes d'équivalence à gauche

Soient deux arbres $F, F' \in \mathcal{F}_k$, nous définissons la fonction $\omega(F) = F \ominus lml(F)$ qui retourne un arbre pivot P_ω après avoir enlevé la feuille la plus à gauche de F. Cette fonction permet d'établir une relation d'équivalence :

$$F \mathcal{P}_\omega F' \Longleftrightarrow \omega(F) = \omega(F)'$$

Une classe d'équivalence à gauche est définie par :

$$\mathcal{P}_\omega(F) = \{F' \in \mathcal{F}_k | F \mathcal{P}_\omega F'\}$$

On ajoute la connaissance des feuilles les plus à gauche qui ont été enlevées des arbres à une liste de couples $L = \{L[1], \ldots, L[|L|]\}$. Chaque couple $(L[i]_1, L[i]_2)$ correspond à la feuille la plus à gauche enlevée à un arbre F. De même que pour les classes d'équivalence à droite, dans les classes à gauche, les couples dans L représentent des (p, l)-extension avec lesquelles on peut reconstruire les arbres appartenant à cette classe.

Alors la classe d'équivalence dont l'arbre pivot est P_ω, sera représentée par :

$$[P_\omega] \sim \{P_\omega, L\}$$

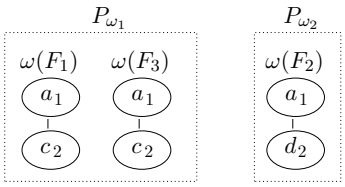


FIG. 5.8 – Des arbres *pivots* à gauche

Exemple 44 *Appliquant la fonction ω sur les arbres de la figure 5.5, on obtient les pivots $\omega(F_1) = F_1 \ominus rml(F_1)$, $\omega(F_2) = F_2 \ominus rml(F_2)$ et $\omega(F_3) = F_3 \ominus rml(F_3)$ illustrés par la figure 5.8. Comme $\omega(F_1) = \omega(F_3)$, il s'agit d'un pivot commun identifié par P_{ω_1}. Le pivot $\delta(F_3)$ est noté P_{ω_2}. Ces arbres pivots permettent le groupement des arbres appartenant à $\mathcal{F}_3$ dans les classes d'équivalence suivantes :*

$$[P_{\omega_1}] \sim \{P_{\omega_1}, \{(1, b), (2, d)\}\}$$
$$[P_{\omega_2}] \sim \{P_{\omega_2}, \{(1, b)\}\}$$

Des classes d'équivalence par la racine

Soit $\gamma(F)$ une fonction qui enlève la racine d'un arbre F et retourne un arbre pivot P_γ de taille $k-1$. $\gamma(F)$ est défini par :

$$\gamma(F) = \begin{cases} F \ominus root(F), \text{ si } |children(root(F))| = 1 \\ \emptyset, \text{ sinon} \end{cases}$$

Si nous permettons à la fonction $\gamma(F)$ d'être appliquée aux arbres avec une racine de degré supérieur à 1, cette fonction retournera une forêt (dont chaque arbre aura une taille inférieure à $(k-1)$. Pour éviter les opérations entre arbres et forêts nous établissons que $\gamma(F) = \emptyset$ si $|children(root(F))| \neq 1$.

Soient deux k-arbres racines $F, F' \in \mathcal{F}_k$. Nous définissons la relation d'équivalence racine $\mathcal{P}_\gamma$ de la façon suivante :

$$F \mathcal{P}_\gamma F' \Longleftrightarrow \gamma(F) = \gamma(F')$$

Cette classe d'équivalence racine est définie par :

$$\mathcal{P}_\gamma(F) = \{F' \in \mathcal{F}_k | F \mathcal{P}_\gamma F'\}$$

Une classe d'équivalence dont l'arbre pivot est P_γ, est notée $[P_\gamma]$. Tout arbre $F \in \mathcal{F}_k$ appartient à une et une seule classe d'équivalence $[P_\gamma]$. Une classe d'équivalence par la

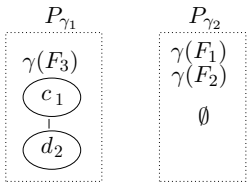


FIG. 5.9 – Des arbres *pivots* racine

racine $[P_\gamma]$ alors, peut être représentée par un arbre pivot P_γ et une liste de couples $O = \{O[1], \ldots, O[|O|]\}$. Chaque couple $O[i]$ correspond à la racine qui a été enlevée à un arbre F. Comme précédemment, une classe d'équivalence par la racine $[P_\gamma]$ peut être décrite par son pivot racine P_γ et une liste d'étiquettes. On a donc :

$$[P_\gamma] \sim \{P_\gamma, O\}$$

Exemple 45 *La figure 5.9 illustre l'obtention des arbres pivots* $\gamma(F_1) = \gamma(F_2) = \emptyset$ *et* $\gamma(F_3) = F_3 \ominus root(F_3)$ *à partir de l'univers* $\mathcal{F}_3$ *(c.f. Figure 5.5). En conséquent, on peut construire les classes d'équivalence :*

$$[P_{\gamma_1}] \sim \{P_{\gamma_1}, \{(-1, c)\}\}$$
$$[P_{\gamma_2}] \sim \{\emptyset, \{\emptyset\}\}$$

Notons qu'on associe $[P_{\gamma_2}]$ *à* $\emptyset$ *car nous ne pouvons pas utiliser les arbres de cette classe pour les coupler avec d'autres.*

Rassemblement des classes d'équivalence

Afin d'obtenir une seule classe d'équivalence avec un arbre pivot P commun, nous rassemblons les classes d'équivalence $[P_\delta]$, $[P_\omega]$ et $[P_\gamma]$ dans une nouvelle classe notée $\mathcal{P}$, définie par :

$$\mathcal{P}(F) = \{F' \in \mathcal{F}_k | F \mathcal{P}_\delta F' \vee F \mathcal{P}_\omega F' \vee F \mathcal{P}_\gamma F'\}$$

et représentée par :

$$[P] \sim \{P, R, L, O\}$$

où P est l'arbre pivot, R (resp. L) est une liste contenant les feuilles les plus à droite (resp. à gauche) enlevées aux arbres fréquents par la fonction $\delta(F)$ (resp. $\omega(F)$) et O est une liste

qui stocke les racines enlevées par la fonction $\gamma(F)$. Les listes des classes d'équivalence $[P]$ sont remplies selon les critères suivants :

- si $F\mathcal{P}_\delta F'$ alors le couple *(position, étiquette)* est ajoutée à la liste R
- si $F\mathcal{P}_\omega F'$ alors le couple *(position,étiquette)* est ajoutée à la liste L
- si $F\mathcal{P}_\gamma F'$ alors le couple *(-1, étiquette)* est ajoutée à la liste O

L'algorithme 21, décrit la procédure pour obtenir les classes d'équivalence à partir d'un ensemble de sous-arbres fréquents de taille k. Dans cette algorithme nous employons une table de hachage (ligne 1), notée P, a fin générer que une seule fois un arbre pivot (en évitant les doublons). Les instructions dans la boucle des lignes 3-13, traitent chaque arbre fréquent F. Dans un premier temps, nous vérifions si le degré de la racine est égal à 1, cela pour éviter la création d'une forêt à partir de F. Si cette condition est rempli, on supprime la racine de F. Puis nous utilisons le pivot résultant S_1 comme clé dans la table P et on ajoute la racine à la liste O dans la classe d'équivalence. Dans un seconde temps, la feuille la plus à gauche de F est enlevé et nous obtenons le pivot S_2. Le pivot obtenu est à nouveau employé comme clé dans la table de hachage et on ajoute le feuille enlevé à la liste L. Nous rappelons qu'on ajoute une couple (position, étiquette) à la liste L. Dans un troisième temps, nous obtenons le pivot S_3 en supprimant la feuille la plus à droite et l'ajoutant à la liste R. Finalement une fois obtenues les classes d'équivalence, nous appelons la méthode Pivot chargée de construire les arbres candidats.

Entrées : $\mathcal{F}_k$ *//un ensemble de* (k)*-sous-arbres fréquents*
Sorties : C_{k+1} *//*$(k+1)$*-candidats*

```
1  map P;
2  //Génération des classes d'équivalence;
3  pour chaque F ∈ F_k faire
4      //enlève la racine;
5      si children(root(F)) = 1 alors
6          S_1 ← (F ⊖ root(F));
7          P[S_1].O+ = (−1, root(F));
8      //enlève la feuille la plus à gauche;
9      S_2 ← (F ⊖ lml(F));
10     P[S_2].L+ = (parent(lml(F)), lml(F));
11     //enlève la feuille la plus à droite;
12     S_3 ← (F ⊖ rml(F));
13     P[S_3].R+ = (parent(rml(F)), rml(F));
14 //Génération de candidats;
15 C ← Pivot(P);
16 retourner C;
```

Algorithme 21 : GenCandPivot

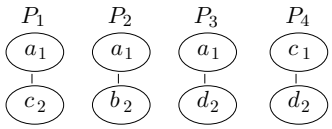


FIG. 5.10 – Des arbres *pivots*

Exemple 46 *Considérons les arbres fréquents dans la figure 5.5 pour la formation des classes ayant un pivot commun P. Dans le premier arbre F_1, nous ne pouvons pas enlever la racine car le résultat serait une forêt. Alors, nous continuons en enlevant le feuille numérotée* 2 *et nous obtenons l'arbre pivot P_1 illustré dans la figure 5.10. Cet arbre servira à la construction de la classe $[P_1]$ et dans sa liste L nous ajoutons la couple $(1, b)$ tel que : $[P_1] \sim \{P_1, P = \emptyset, L = \{(1, b)\}, O = \emptyset\}$. En continuant avec l'arbre F_1, si nous enlevons la feuille* 3*, nous obtenons un nouveau pivot identifié P_2 dans la figure 5.10. À la différence du cas précèdent, on ajoute la couple $(1, c)$ à la liste R tel que : $[P_2] \sim \{P_2, R = \{(1, c)\}, L = \emptyset, O = \emptyset\}$. Les arbres dans F_2 et F_3 sont traités de la même manière en obtenant les classes d'équivalence suivantes :*

$$
\begin{aligned}
[P_1] \sim & \{P_1, R = \{(2, d)\}, L = \{(1, b), (2, d)\}, O = \emptyset\} \\
[P_2] \sim & \{P_2, R = \{(1, c), (1, d)\}, L = \emptyset, O = \emptyset\} \\
[P_3] \sim & \{P_3, R = \emptyset, L = \{(1, b)\}, O = \emptyset\} \\
[P_4] \sim & \{P_4, R = \emptyset, L = \emptyset, O = \{(-1, a)\}\}
\end{aligned}
$$

L'algorithme Pivot

Les candidats de taille $k + 1$, notés $\mathcal{C}_{k+1}$, sont générés à partir de la combinaison des éléments des classes d'équivalence construites sur les fréquents de taille k qui ont donc un pivot commun de taille $k - 1$ (c.f. Algorithme 22).

Dans un premier temps (lignes 2-5), nous faisons les combinaisons des listes de couples à droite et de la racine pour produire l'ensemble de candidats $\mathcal{C}^{\gamma}_{k+1}$ suivant :

$$\mathcal{C}^{\gamma}_{k+1} = \{O[i] \oplus P \oplus R[j] \colon i \in [1, |O|] \text{ et } j \in [1, |R|]$$

Ensuite (lignes 6-8), on considère le pivot P et on fait toutes les combinaisons des listes de couples à droite et à gauche pour produire l'ensemble de candidats $\mathcal{C}^{\omega}_{k+1}$ suivant :

$$\mathcal{C}^{\omega}_{k+1} = \{L[i] \oplus P \oplus R[j] \colon i \in [1, |L|] \text{ et } j \in [1, |R|]$$

Entrées : $[P]_{k-1}$ //$(k-1)$-*classes d'équivalence*
Sorties : C_{k+1} //$(k+1)$-*candidats*

```
1 C ← ∅;
2 pour chaque p ∈ [P]_{k-1} faire
3     pour i ← 0 to p.O.size() faire
4         pour j ← 0 to p.R.size() faire
5             C+ = p.O[i] ⊕ P ⊕ P.R[j];
6     pour i ← 0 to p.L.size() faire
7         pour j ← 0 to p.R.size() faire
8             C+ = p.L[i] ⊕ P ⊕ P.R[j];
9 retourner C;
```

Algorithme 22 : Pivot :génération d'arbres candidats

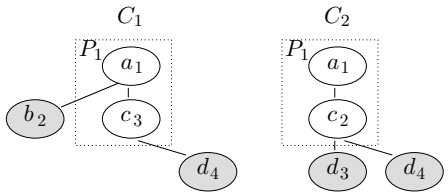


Fig. 5.11 – Des arbres *candidats* obtenus par l'approche Pivot

Finalement l'ensemble total de candidats est généré à partir de l'union des deux sous-ensembles de candidats générés en largeur (classes d'équivalence gauche et droite) et en profondeur (classes d'équivalence racine et droite) :

$$\mathcal{C}_{k+1} = \mathcal{C}^{\omega}_{k+1} \bigcup \mathcal{C}^{\gamma}_{k+1}$$

Exemple 47 *La figure 5.11 illustre le candidat obtenu en combinant les composants dans les classe d'équivalence* $[P_1], \ldots, [P_4]$. *Avec la classe* $[P_1]$, *le candidat* C_1 *est généré avec le premier élément de la liste à gauche* L *plus le seul élément dans la liste à droite* R *de manière que :* $C_1 = (1, b) \oplus P_1 \oplus (2, d)$. *Le deuxième candidat est obtenu par la combinaison du deuxième élément dans* L *et l'élément dans* R *tel que* $C_1 = (2, d) \oplus P_1 \oplus (2, d)$. *Comme la liste d'éléments racine* O *est vide, alors n'est pas possible la construction de candidats en combinant cette liste avec la liste* R. *Pour les classes restantes* $[P_2], [P_3]$ $[P_4]$, *aucun candidats est généré soit parce que les listes* R *se trouvent vides soit parce que les listes* L *ou* O *sont vides.*

À différence des 10 *candidats générés avec une seule classe d'équivalence comme proposé par Zaki (c.f. Figure 5.7), la méthode* Pivot *génère que* 2 *candidats.*

5.3.3. Preuve de complétude

Les candidats sont générés à partir des classes d'équivalence définies ci-dessus :

$$\mathcal{C}_{k+1} = \mathcal{C}^{\omega}_{k+1} \bigcup \mathcal{C}^{\gamma}_{k+1}$$

Nous montrons ici que tous les sous arbres fréquents sont trouvés, et qu'ils sont donc générés comme candidats.

Proposition 2 *La méthode PIVOT permet de retrouver tout sous-arbre fréquent.*

Preuve 1 *Considérons que l'algorithme de calcul du support est correct. Il s'agit de montrer que tout sous-arbre fréquent a été généré comme un candidat :* $\forall F_k \in \mathcal{F}, F_k \in \mathcal{C}_k$.

Nous montrons cette propriété par récurrence sur la taille k *du sous-arbre fréquent.*

$\mathbf{k = 1}$ *Les arbres fréquents de taille* 1 *(nœuds) sont retrouvés par un parcours de la base de façon similaire aux autres approches, ce qui est donc complet.*

$\mathbf{k = 2}$ *Les arbres fréquents de taille* 2 *sont retrouvés en combinant de toutes les manières possibles les nœuds fréquents, ce qui rend donc notre approche complète au niveau* 2.

$\mathbf{k > 2}$. *Considérons que tous les arbres de taille* $t \leq k$ *ont été retrouvés. Montrons que tous les arbres fréquents de taille* $k+1$ *sont générés comme candidat.*

On considère $F_{k+1} \in \mathcal{F}_{k+1}$*, tout sous-arbre de* F_{k+1} *est donc fréquent.*

De plus, il est aisé de montrer que pour tout arbre T *comportant au moins* 3 *nœuds, supprimer d'abord la feuille la plus à droite puis la feuille la plus à gauche ou bien d'abord la feuille la plus à gauche puis la feuille la plus à droite revient au même. Et de manière similaire, si la configuration des arbres s'y prête (arbre racine), supprimer d'abord la racine puis la feuille la plus à droite ou d'abord la feuille la plus à droite puis la racine revient également au même :*

$$\begin{aligned}(T \ominus lml(T)) \ominus (rml(T \ominus lml(T))) \\ = (T \ominus rml(T)) \ominus (lml(T \ominus rml(T)))\end{aligned}$$

$$\begin{aligned}(T \ominus lml(T)) \ominus (root(T \ominus lml(T))) \\ = (T \ominus root(T)) \ominus (rml(T \ominus root(T)))\end{aligned}$$

Ainsi, le pivot droit du pivot gauche d'un arbre t *et le pivot gauche du pivot droit du même arbre sont égaux. Donc ces arbres sont combinés par notre approche puisque les arbres sont générés en calculant le produit cartésien des arbres contenus dans les classes d'équivalence* $[P_\gamma]$ *et* $[P_\delta]$ *d'une part et* $[P_\omega]$ *et* $[P_\delta]$ *d'autre part. Dans ce contexte, deux cas de figure se présentent (expansion en largeur ou en profondeur) :*

soit *il existe* $F \in F_{k+1}$ *de taille* k *tel que* $depth(F) = depth(F_{k+1})$. *Alors on note* $F_k^1 = F_{k+1} \ominus rml(F_{k+1})$ *et* $F_k^2 = F_{k+1} \ominus lml(F_{k+1})$ *les deux arbres fréquents trouvés en supprimant respectivement les feuilles les plus à droite et à gauche. Comme mentionné ci-dessus, le pivot de la classe d'équivalence gauche de* F_k^1 *(*$F_k^1 \ominus lml(F_k^1)$*) est égal au pivot de la classe d'équivalence droite de* F_k^2 *(*$F_k^2 \ominus rml(F_k^2)$*). Donc* F_k^1 *et* F_k^2 *seront combinés par notre méthode, pour donner* F_{k+1} *par expansion en largeur.*

sinon *il existe forcément un arbre* F *de profondeur* $depth(F_{k+1}) - 1$ *tel que* F *est fréquent puisque tout sous-arbre est fréquent. Alors on note* $F_k^3 = F_{k+1} \ominus root(F_{k+1})$ *et* $F_k^4 = F_{k+1} \ominus rml(F_{k+1})$ *avec* $depth(F_k^3) = depth(F_{k+1}) - 1$. *Or le pivot droit de* F_k^3 *est égal au pivot racine de* F_k^4. *Il existe donc une classe d'équivalence droite à l'étape* $k-1$ *ayant même pivot qu'une classe d'équivalence racine, ce qui produira bien* F_{k+1} *par expansion en profondeur.*

□

5.4. Validation du support des candidats

La fouille d'arbres est fortement liée à la définition de l'inclusion d'un arbre dans un autre. Dans les sections suivantes nous présentons les approches pour l'inclusion : *induite*, *incrustée* et *floue*.

5.4.1. RSF$_i$, une approche *induite*

Notons que dans le cas d'une inclusion *induite*, nous recherchons une instanciation *exacte* du candidat au sein des arbres de la base. Nous disons qu'un arbre est inclus de façon induite si nous cherchons des occurrences ϕ d'un arbre dans une base de données qui vérifient les conditions suivantes. Elles doivent :

1. être injectives
2. préserver les étiquettes
3. préserver les relations *père-fils* des arêtes
4. préserver l'ordre entre nœuds frères

Pour le comptage du support d'un arbre candidat nous ne considérons que des occurrences dans les documents $doc(\phi)$. Le calcul du support de chaque candidat consiste à compter le nombre d'arbres de la base qui contiennent ce sous-arbre candidat. Ainsi pour chaque arbre de la base, nous recherchons les *points d'ancrage* sur lesquels la racine du sous-arbre à tester peut s'accrocher. Ces points représentent en fait aux nœuds dans l'arbre qui correspondent à la racine de l'arbre à tester. Pour chaque point d'ancrage trouvé, on cherche alors à instancier l'ensemble des nœuds de l'arbre candidat au sein de l'arbre courant testé,

```
   Entrées : C //un arbre motif, T //un arbre cible
   Sorties : trv //vrai si C ⊑ T
 1 trv ← FAUX;
 2 si size(C) > size(T) alors
 3     retourner trv
 4 M.resize(size(C))        //suivre la trace d'une occurrence de C dans T;
 5 pour chaque nœud v ∈ T faire
 6     u ← root(C);
 7     si λ(u) = λ(v) alors
 8         //Selon le type d'inclusion on appelle les fonctions :
 9         //INDUITE (⊑_i), INCRUSTÉE (⊑_e) ou FLOUE (⊑_f);
10         trv ← INDUITE(C, T, u, v, M);
11         si (trv = VRAI) alors
12             retourner trv;
13 retourner trv;
```

Algorithme 23 : INCLUSION : verifie si C est inclus dans T

i.e. les fils du nœud à tester. Si tous les nœuds du candidat ont été trouvés, l'arbre supporte le candidat et le support de la structure candidate est alors augmenté. Ce n'est pas le cas si tous les nœuds de l'arbre ont été parcourus sans trouver l'ensemble des nœuds du candidat.

Dans RSF (c.f. Algorithme 17, lignes 6-14), pour chaque arbre de la base, une recherche est effectuée pour voir s'il existe des *points d'ancrage* sur lesquels la racine du sous-arbre C à tester peut s'instancier (appel à l'algorithme INCLUSION) dans un arbre T. Si un sous-arbre existe alors son support est augmenté. Si le support est plus grand que le seuil σ, alors l'arbre candidats C en tant qu'arbre fréquent.

Considérons l'algorithme de gestion des points d'ancrage (c.f. Algorithme 23). Pour chaque point d'ancrage trouvé, i.e. pour chaque nœud du sous arbre candidat C qui possède le même label dans l'arbre T, on cherche à accrocher l'ensemble des nœuds de l'arbre candidat au sein de l'arbre couramment testé T. En d'autres termes, nous souhaitons projeter le sous-arbre candidat C dans l'arbre T. Ceci est réalisé par l'intermédiaire des algorithmes récursifs INDUITE et INCRUSTÉE (c.f. Algorithmes 24 et 25) selon un type d'inclusion choisi par l'utilisateur.

L'algorithme INDUITE est utilisé pour chercher une occurrence *induite* du candidat au sein des arbres de la base. Si tous les nœuds du candidat ont été trouvés, l'algorithme retourne alors la valeur $VRAI$ (l'arbre supporte le candidat). Il retourne la valeur $FAUX$ si tous les nœuds de l'arbre ont été parcourus sans trouver l'ensemble des nœuds du candidat.

Entrées : C, T *//un arbre cible et un arbre motif,*
u, v *//des points d'ancrage de* $u \in C$ *avec* $v \in T$,
M *//stocke une occurrence de* C
Sorties : $VRAI$ *//si* C *est inclus dans* T

```
1  occ ← FAUX;
2  M[u] ← v;
3  u ← u + 1;
4  si u <= rml(C) alors
5  |  P ← {w : w ∈ T tel que parent(w) = M[parent(u)] et λ(w) = λ(u)};
6  |  pour chaque nœud w ∈ P faire
7  |  |  occ ← INDUITE(C, T, u, w, M);
8  |  |  si occ = VRAI alors
9  |  |  |  retourner occ;
10 sinon
11 |  occ ← VRAI;
12 si occ = FAUX alors
13 |  u ← u − 1;
14 retourner occ;
```

Algorithme 24 : INDUITE : recherche des points d'ancrage de C dans T

5.4.2. RSF$_e$, une approche *incrustée*

Exemple 48 *Pour illustrer nos propos, nous reprenons la base de données D de la figure 5.1 et un support minimal* $\sigma = 2$. *En considérant une inclusion assez restrictive telle que l'inclusion* induite*, on doit respecter les relations* père-fils *à l'intérieur des arbres et l'ordre entre noeuds frères. Alors, l'ensemble d'arbres fréquents qu'on extrait de la base de données D est* $\mathcal{F} = \{(a), (b), (c), (a(b)), (a(c)\}$. *Cependant, si nous relaxons la contrainte des relations directes* père-fils*, en permettant les rations indirectes* ancêtre-descendant *au moment de valider l'inclusion des arbres candidats dans la base, alors l'ensemble d'arbres fréquents est obtenu est* $\mathcal{F} = \{(a), (b), (c), (a(b)), (a(c), (a(b)(b)), (a(b)(c)), (a(b)(b)(c))\}$.

5.4.3. RSF$_f$, une approche *floue*

Comme on l'a vu dans l'algorithme 17, un arbre candidat doit être « entièrement » inclus dans un arbre de la base de données pour considérer un incrément au moment de calculer son support. Dans cette section, nous arguons du fait qu'un arbre peut être considéré dans une certaine mesure dans ce calcul même s'il n'est pas entièrement inclus . À ce propos, nous définissons quatre manières de considérer une inclusion floue d'un arbre dans un autre :

Entrées : C, T *//un arbre cible et un arbre motif,*
u, v *//des points d'ancrage de* $u \in C$ *avec* $v \in T$,
M *//stocke une occurrence de* C
Sorties : $VRAI$ *//si* C *est inclus dans* T

```
1  occ ← FAUX;
2  M[u] ← v;
3  u ← u + 1;
4  si u <= rml(C) alors
5  |  P ← {w: w ∈ T tel que w ∈ scope(M[parent(u)]) et w ∉
   |  scope(previous(M[u])) et λ(w) = λ(u)};
6  |  pour chaque nœud w ∈ P faire
7  |  |  occ ← INCRUSTE(C, T, u, w, M);
8  |  |  si occ = VRAI alors retourner occ;
9  sinon
10 |  occ ← VRAI;
11 si occ = FAUX alors
12 |  u ← u − 1;
13 retourner occ;
```

Algorithme 25 : INCRUSTEE : recherche d'une occurrence de C incrustée dans T

- La première manière vise à considérer les relations verticales des arbres. Contrairement à l'inclusion induite, l'inclusion incrustée nous permet de relaxer ou ramollir les rapports *ancêtre-descendant* entre nœuds. Néanmoins, aucune méthode n'existe pour contrôler le degré de ces rapports.
- Dans la deuxième, nous nous intéressons aux relations horizontales des arbres. Tandis que les approches classiques considèrent que les nœuds frères sont ordonnés ou non-ordonnés, il n'y a aucune méthode qui considère seulement une proportion des nœuds ordonnés ou non-ordonnés.
- La troisième manière généralise la précédente en considérant la proportion des nœuds.
- La dernière considère des similitudes entre les nœuds.

Ces définitions sont alors intégrées dans un nouvel algorithme (c.f. Algorithme 26) qui extrait les sous-arbres fréquents dans une approche floue. Ils visent à décrire le point auquel un arbre est inclus dans un autre. Tandis que les approches classiques traitent *l'inclusion crisp*, signifiant qu'un arbre est ou non inclus dans un autre, nous proposons d'employer un degré d'inclusion, défini entre 0 et 1. Les quatre façons par lesquelles ce degré peut être obtenu sont décrites ci-dessous. Elles correspondent aux quatre manières pour lesquelles la fonction $Fuzzy\_Inclusion\_Degree$ dans l'algorithme 26 peut être définie.

Entrées : Σ *//un alphabet*,
$\mathcal{D}$ *//une base de données sur* Σ,
σ *//un support minimal*
Sorties : $\mathcal{F}$ *//l'ensemble de tous les sous-arbres fréquents dans D*

```
1  F_1 ← Frequents1(Σ, D, σ);
2  F_2 ← Frequents2(F_1, D, σ);
3  k ← 2;
4  tant que F_k ≠ ∅ faire
5      C_{k+1} ← GenCandidats(F_k, F_2);
6      pour chaque arbre candidat C ∈ C_k faire
7          C.cpt ← 0;
8          pour chaque arbre T ∈ D faire
9              C.cpt ← ComputeNewCpt(C.cpt, FuzzyInclusionDegree(C, T));
10         si C.cpt ≥ σ alors
11             F_k ← F_k ∪ {C};
12     k++;
13 retourner F = F_1 ∪ F_2 ∪ ... ∪ F_{k-1};
```

Algorithme 26 : Fouille de arbres fréquents flous

Dans l'algorithme 26, le degré *cpt* d'un candidat est mis à jour après chaque balayage de l'arbre par la fonction ComputeNewCpt. Plusieurs possibilités sont données en fusionnant ces degrés, selon la manière dont le comptage est exécuté. Ces possibilités sont décrites par l'algorithme 27 en considérant les méthodes de comptage : *sigma-count*, *thresholded-count*, *thresholded-sigma-count*. Cette méthode est supposée être connue, aussi bien que la valeur du seuil ω au besoin.

Nous considérons un calcul nœud par nœud réalisé dans les approches traditionnelles. Cependant, ce calcul ne vise pas un matching *exact* entre un nœud d'un arbre candidat et un nœud d'un arbre de la base de données mais aboutir plutôt *la mesure* dans laquelle un nœud correspond à un autre. Ce calcul nœud par nœud à un degré *deg* retourné par la fonction de FuzzyMatch. Cette fonction tient compte du nœud n considéré, l'arbre S dont ce nœud est lancé, et l'arbre cible T. L'arbre S doit être considéré afin de garder la structure arborescente, et non seulement les nœuds sans aucun raccordement. Nous avons ainsi traité un ensemble de degrés flous, s'étendant entre 0 et 1. Ils se fusionnent pour effectuer le calcul de FuzzyInclusionDegree.

Afin de fournir à l'utilisateur un ensemble de solutions, nous considérons les opérateurs OWA (*Ordered Weighted Averaging Operator*) [Yag93], comme l'illustre l'algorithme 28. Un

Entrées : *od //l'ancien dregré, i //degré d'inclusion floue*
Sorties : *d //le nouveau dregré*

si *la méthode de comptage est* sigma-count **alors**
  $d \leftarrow od + i$;
/* ω *est un degré établi par l'utilisateur*/
si *la méthode de comptage est* thresholded-count *avec un degré minimal* ω **alors**
  si $i \geq \omega$ **alors**
    $d \leftarrow od + 1$;
  sinon
    $d \leftarrow od$;
si *la méthode de comptage est* thresholded-sigma-count *avec un degré minimal* ω **alors**
  si $i \geq \omega$ **alors**
    $d \leftarrow od + i$;
  sinon
    $d \leftarrow od$;
retourner d;

Algorithme 27 : COMPUTENEWCPT : Fusionnant les degrés flous.

opérateur OWA de dimension n est un mapping

$$F : R^n \rightarrow R$$

cela a un n vecteur associé $W = (w_1, w_2, \ldots, w_n)^T$ tel que $w_i \in [0.1]$ et $\sum_{i=1}^{n} w_i = 1$. Nous avons $F(a_1, a_2, \ldots, a_n) = \sum_{j=1}^{n} w_j \cdot b_j$ où b_j est la $j^{\text{ième}}$ plus grande valeur de a_i.

Dans ce cadre, il est ainsi possible de calculer le degré final de différentes manières. De la manière la plus pessimiste, on calcule le *minimum* (*AND* logique), représentant la frontière inférieure. D'une manière très optimiste, le *maximum* est calculé (*OR* logique), représentant la frontière supérieure. Entre ces deux frontières, toutes les possibilités sont offertes en changeant les poids ou en définissant plusieurs degrés d' *andness/orness*.

La mesure *orness* est définie comme suit :

$$orness(W) = \frac{1}{n-1} \sum_{i=1}^{n} ((n-i) \cdot w_i)$$

La mesure *andness* est définie comme

$$andness(W) = 1 - orness(W)$$

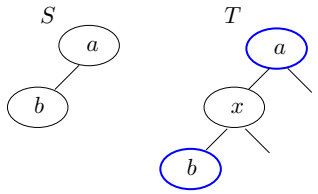


FIG. 5.12 – Chemins verticaux flous

Par conséquent, l'utilisateur peut choisir s'il veut *tous* les nœuds semblables, ou seulement *certains d'entre eux* (en utilisant un degré désiré d'andness/orness), ou *au moins un* d'une façon très flexible.

Entrées : S, T *//un arbre candidat et un arbre cible*
Sorties : $d \in [0,1]$ *//un degré flou*

1 **pour chaque** *nœud* $u \in S$ **faire**
2 $\quad deg_u \leftarrow$ FUZZYMATCH(u, S, T);
3 $d \leftarrow OWA_{u=1}^{|S|} deg_u$;
4 **retourner** d;

Algorithme 28 : FUZZYINCLUSIONDEGREE : Calcul du degré d'inclusion flou.

Notons que tous les opérateurs OWA ne sont pas associatifs. Pour cette raison le calcul du degré d dans l'algorithme 28 peut seulement être effectué après le balayage de la totalité de la base de données. Dans certains cas, ce degré est calculé en employant un opérateur associatif (e.g. minimum), lequel permet le calcul du degré à la volée. De cette façon, il est possible d'optimiser le calcul. Par exemple, en calculant le minimum, le processus peut être arrêté dès que le degré deviendra plus bas que le seuil minimum, si un seuil est considéré dans l'algorithme 27.

Des relations verticales floues

Considérons les liens indirects flous dans des arbres. Par exemple, la figure 5.12 montre l'arbre S qui est inclus (incrusté) dans l'arbre T. Lorsqu'on prend en considération des arbres incrustés, leur support est calculé sans compter le type de relation *ancêtre-descendant* qui existe parmi les nœuds. C'est-à-dire, qu'il n'existe aucune considération sur le nombre de nœuds qui séparent un ancêtre d'un descendant. Même s'il est de grand intérêt de considérer deux nœuds comme relatés même s'ils ne sont pas directement liés, nous arguons du fait qu'il devient non-pertinent de considérer que deux nœuds sont connexes s'ils ne sont séparés que par un nombre élevé des nœuds.

Dans notre approche, nous proposons ainsi de donner une portée à ce rapport d'ancêtre-descendant. Cette portée est définie en considérant le nombre de nœuds entre l'ancêtre et

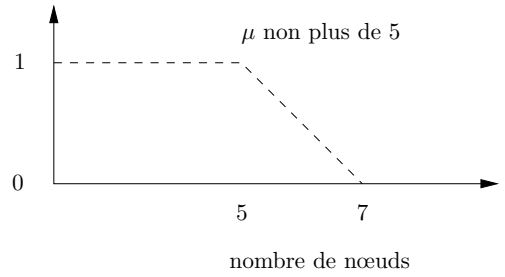


FIG. 5.13 – Degré des relations *ancêtre-descendant* floues. Quantificateur absolu flou.

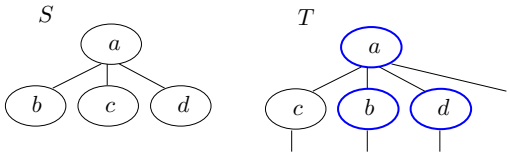


FIG. 5.14 – Chemins horizontaux flous

le descendant. Puisque considérer des frontières floues n'a pas de sens, nous proposons de considérer une portée floue pour les rapports ancêtre-descendant. Nous considérons des fonctions floues décrivant les relations ancêtre-descendant selon le nombre de nœuds séparant les deux nœuds, comme montré sur la figure 5.13 où nous ne voulons pas plus de cinq nœuds d'une manière floue. Á cet effet, nous employons des quantificateurs flous [YB97, Zad83].

La théorie des ensembles flous permet de représenter des quantificateurs flous, comme *au moins 2 (at least), la plupart (most)*, par des fonctions d'appartenance (comme illustré par la figure 5.13 et la figure 5.15). Dans ce cadre, les quantificateurs *absolus* comme *au moins 2* sont distingués de ceux *relatifs* (par exemple *la plupart*).

La fonction FUZZYINCLUSIONDEGREE de l'algorithme 26 est alors calculée en fusionnant les degrés auxquels des nœuds peuvent être assortis. Ce degré assorti résulte du fait qu'un nœud peut être encore lié dans une relation floue ancêtre-descendant définie par l'utilisateur.

Des relations horizontales floues

Dans cette section nous considérons maintenant l'inclusion floue par niveau. Lorsqu'on considère des arbres ordonnés dans les approches non-floues (crisp), un sous-arbre S est inclus dans un arbre T seulement si tous les nœuds de S peuvent être liés aux nœuds de T dans le même ordre. Dans notre approche, nous proposons d'atténuer cette définition en considérant la proportion de nœuds inclus et bien ordonnés. Par exemple, la figure 5.14 montre un arbre ordonné S, lequel n'est pas incrusté dans un arbre ordonné T dans les approches classiques car le nœud étiqueté b est mal ordonné.

Dans notre approche, nous considérons S comme inclus dans T à un certain degré

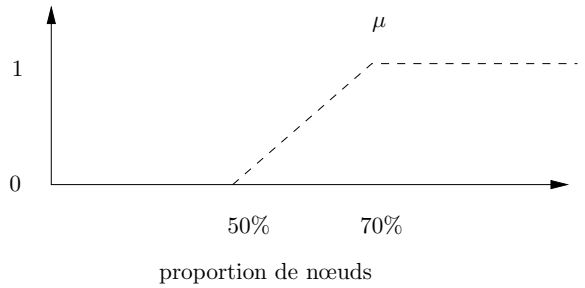


FIG. 5.15 – Quantificateur relatif flou : La proportion de nœuds

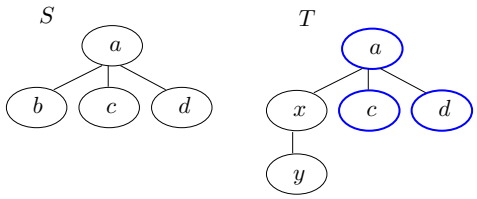


FIG. 5.16 – Inclusion partielle

puisque les autres nœuds satisfont l'inclusion. Ce degré est obtenu par la fonction *Fuzzy_Inclusion_Degree* et utilisé dans l' algorithme 26. Il dépend de la proportion de nœuds qui sont bien-ordonnés à un certain niveau. Nous considérons ainsi la fonction floue définissant le degré (s'étendant de 0 à 1) à ce que *la plupart* des nœuds sont bien ordonnés, comme l'illustre la figure 5.15.

Inclusion partielle

Dans cette section, nous considérons l'inclusion partielle des nœuds. D'une façon générale, *tous* les nœuds d'un sous-arbre S doivent être inclus dans un arbre T si $S \sqsubseteq T$. Cependant, nous arguons du fait que cela est trop restrictif en fouillant des données du monde réel où les imperfections sont souvent présentes. Par exemple, figure 5.16 montre un arbre S n'ayant que 75% de ses nœuds inclus dans T.

Dans notre approche, nous proposons ainsi de définir l'inclusion partielle en considérant la proportion de nœuds de S présents dans T. Cette proportion est alors liée à un quantificateur flou qui évalue dans quelle mesure le nœud doit être pris en considération en calculant la fonction *Fuzzy_Inclusion_Degree*. Notons que dans le cas classique, l'extraction des arbres totalement inclus permet d'arrêter le balayage de base de données, car à chaque fois qu'un nœud ne peut pas être assorti, il n'y a aucun besoin de rechercher les autres. Dans notre approche, des *outliers* sont acceptés, qui peuvent être considérés comme un inconvénient en considérant le passage à l'échelle. Cependant, il est encore possible de arrêter la recherche quand la proportion a été dépassée.

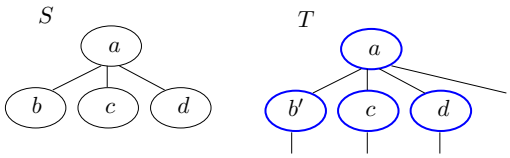


FIG. 5.17 – Des arbres semblables

Des arbres semblables

Dans cette section, nous considérons des similitudes floues [BMRB96]. Ces similitudes sont liées à la connaissance du sujet dans le domaine que nous traitons. Par exemple, en traitant des données dans le domaine de la biologie, il est possible de savoir dans quelle mesure le nœud marqué par quelque bactérie est semblable au nœud encore non marqué. Considérons maintenant que nous sommes équipés d'une certaine connaissance sur les données. Un inconvénient des approches d'extraction de schémas est celui de la détection où l'inclusion est seulement effectuée sur des nœuds ayant mêmes étiquettes. Selon le point de vue sémantique du Web, cette restriction est utilisée puisque deux étiquettes différentes pourraient décrire des concepts semblables.

Dans cette méthode, nous considérons des approches floues pour surmonter cet inconvénient. Par exemple, la figure 5.17 montre un arbre S qui devrait être assorti avec T si nous savons que le concept b est près de b'. À cet effet, nous considérons des ontologies floues comme dans [Par04], et plus généralement des relations floues [KF88] décrivant à quel point deux nœuds sont semblables.

Rappelons-nous qu'une relation floue R entre deux ensembles de référence X et Y est définie comme sous-ensemble flou de $X \times Y$. En particulier, quand X et Y sont finis, la relation peut être décrite par une matrice $M(R)$ comprenant les valeurs de sa fonction de correspondance.

Il est donc possible de calculer la fermeture transitive d'une relation floue. Supposons que G soit le graphe associé à la relation floue R et que f_R soit sa fonction de correspondance. La *fermeture transitive* de R est R_T telle que $f_{R_T}(x, y) \neq \emptyset$, ssi il existe un chemin dans G du nœud x au nœud y.

Soit R_T la *max-min* fermeture transitive de R et soit $R \circ R$ la max-min composition de R et de R. R_T est calculé comme suit :

1. $R' = R$
2. tant que R' change

   $R = R'$

   $R' = R \bigcup (R \circ R)$
3. Arrêter. $R_T = R'$

Dans notre approche, nous considérons une relation floue R définie sur l'univers d'étiquettes des nœuds N. Nous avons ainsi $R : N \times N \rightarrow [0.1]$ décrivant dans quelle mesure un nœud $n \in N$ est semblable à un nœud $n' \in N$. En calculant la fermeture transitive de R, tous les nœuds peuvent être comparés aux autres nœuds. Des relations symétriques sont considérées puisque nous traitons des similitudes.

Ce degré de similitude est pris en considération dans la fonction de $Fuzzy\_Match$ en calculant le $Fuzzy\_Inclusion\_Degree$, comme cela est illustré par l'algorithme 29. Dans cet algorithme, la conservation de la structure arborescente est vérifiée. Cette vérification est exigée afin d'empêcher que la structure arborescente soit détruite. Cette vérification doit tenir compte ainsi des nœuds précédents qui ont été assortis ensemble.

Entrées : u, S, T *//un nœud, un arbre candidat et un arbre cible*
Sorties : $d \in [0, 1]$ *//un degré flou*

pour chaque *nœud* $v \in T$ **faire**
  si *u match v et respecte la structure de l'arbre S* **alors**
    $deg_v \leftarrow f_R(u, v)$;
$d \leftarrow OWA_{v=1}^{|T|} deg_v$;
retourner d;

Algorithme 29 : FuzzyMatch : Calcul du "matching" flou entre nœuds.

L'algorithme 29 propose de calculer le degré final comme la valeur fusionnée de tous les degrés de similitude avec nœuds de T. Dans le cas classique, l'algorithme est plus efficace puisqu'il n'y a aucun besoin de balayer tous les nœuds de T. Une autre possibilité sera ainsi d'arrêter le balayage de T dès que le nœud semblable $n \in T$ est trouvé, ce qui nous mène à l'algorithme 30. Dans ce cas-ci, nous considérons un seuil défini pour l'utilisateur γ.

Entrées : u, S, T *//un nœud, un arbre candidat et un arbre cible*
Sorties : $d \in [0, 1]$ *//un degré flou*

pour chaque *nœud* $u' \in T$ **faire**
  /* γ *attend un seuil défini par l'utilisateur* */
  if *u matches* u' *et respecte la structure de l'arbre S et* $f_R(u, u') \geq \gamma$ **then**
    retourner $f_R(u, u')$;

Algorithme 30 : FuzzyMatch : Calcul du "matching" flou entre nœuds avec une condition d'arrêt.

Des morphismes brouillés peuvent également être considérés afin de traiter ce problème [PB02, Koh03].

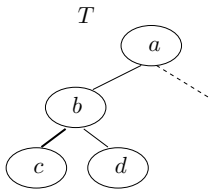


FIG. 5.18 – Un arbre flou

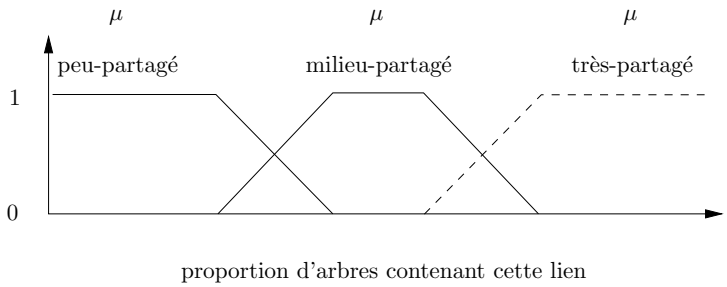


FIG. 5.19 – Quantificateurs flous pour des arbres flous.

Des sous-arbres fréquents flous

Les approches classiques extraient les arbres fréquents qui ne fournissent pas beaucoup d'informations au sujet des occurrences dans la base de données excepté le fait que ces sous-arbres sont enfoncés dans un nombre suffisant d'arbres de la base de données (selon le seuil de support minimal défini pour l'utilisateur). Nous proposons ainsi de prolonger la connaissance sur les structures sémantiques extraites en fournissant des liens flous dans les arbres fréquents. Ces liens flous sont plus ou moins épais. Ils aident à savoir s'ils sont *très* partagés, *demi* partagés ou *peu* partagés, comme illustrés sur la figure 5.18.

Dans ce cadre, les sous-arbres fréquents sont des graphes acycliques flous. Chaque lien entre deux nœuds est marqué par un degré s'étendant de 0 à 1. Ce degré est obtenu en considérant la proportion d'arbres de la base de données qui contiennent le lien. Cette proportion est alors tracée avec des quantificateurs flous comme *très partagé*, demi partagé ou peu partagé, comme montré par figure 5.19. Elle est graphiquement représentée en tenant compte du quantificateur flou ayant le plus grand degré de correspondance. Chaque quantificateur brouillé est alors représenté par une épaisseur différente.

Dans ce chapitre, nous avons défini une nouvelle méthode de génération de candidats chargée d'élaguer de nombreux éléments dont nous savons par avance qu'il ne seront pas fréquents. Nous avons aussi présenté des différents types d'inclusion et les algorithmes associés. Les algorithmes décrits dans ce chapitre ont permis d'obtenir de très bon résultats (voir Chapitre 6 "Expérimentations"). Dans le chapitre suivant nous introduisons donc une nouvelle méthode optimisée de génération de candidats.

5.5. Conclusion

Ce chapitre a été consacré à la présentation de notre approche originale pour la fouille d'arbres fréquents à partir de bases de données arborescentes.

Nous intéressant dans un première temps à la génération des candidats, nous avons introduit la définition et la formalisation de PIVOT, une nouvelle approche, s'appuyant sur le principe *Pattern-Growth* [HPY00], pour l'extraction d'arborescences fréquentes. Cette méthode est originale par rapport aux techniques classiques qui se basent sur l'extension des candidats par la branche la plus à droite, dans le sens où elle utilise trois classes d'équivalence (droite, gauche et racine). Le but principal de PIVOT est d'engendrer moins de candidats par rapport aux méthodes de type *générer-élaguer*. Nous avons aussi inclus une preuve de complétude en démontrant que tout arbre fréquent est extrait par PIVOT.

Dans un deuxième temps, nous nous sommes intéressés à la validation de candidats indépendamment de la méthode de génération des candidats. Dans ce cadre nous avons introduit dans l'approche RSF une validation des candidats selon une inclusion *induite*, *incrustée* ou *floue* notées respectivement RSF_i, RSF_e et RSF_f.

Traditionnellement la littérature propose des algorithmes basés sur les inclusions *induite* ou *incrustée*. Dans ces cas un arbre candidat doit être entièrement inclus dans un arbre de la base de données pour pouvoir augmenter son support. Toutefois, dans le cas de données réelles ces deux types d'inclusion peuvent s'avérer très/trop restrictifs. Pour répondre à cette problématique nous proposons une technique de validation de candidats *floue*. Nous considérons alors qu'un arbre candidat est inclus « partiellement » dans un arbre de la base. Cette inclusion n'est pas exacte dans le sens où elle considère un *degré d'inclusion*. L'inclusion *floue* relaxe ou modifie une des quatre conditions concernant l'inclusion des arbres visant une application de notre approche dans les cas des données réelles (e.g. dans le cas de la construction de schémas médiateurs). Dans ce cadre nous proposons les types suivants de validation des arbres candidats selon une inclusion *floue* :

Des relations verticales floues, incluant une portée dans les relations *ancêtre-descendant* des nœuds permettant de contrôler ces relations selon le nombre de nœuds entre l'ancêtre et le descendant.

Des relations horizontales floues, relaxant la relation d'ordre entre frères de telle sorte que l'inclusion d'un arbre candidat dans la base de données est considéré malgré l'existence de certains nœuds non-ordonnés.

Une inclusion partielle, qui modifie la condition d'injectivité chargée d'établir la correspondance entre les nœuds d'un arbre candidat et un arbre de la base.

Des arbres semblables, modifiant la condition qui nous oblige à préserver les étiquettes dans la validation des candidats en permettant une certaine relation de synonymie.

Chapitre 6

Expérimentations

Dans ce chapitre nous présentons les résultats des expérimentations menées afin de valider notre approche visant deux objectifs. Le premier vise à analyser le comportement de nos algorithmes sur des jeux de données synthétiques. Le deuxième consiste à vérifier le comportement de nos algorithmes appliqués sur des bases de données réelles.

Sommaire

6.1. Implémentation **110**
6.2. Expérimentations sur des données synthétiques **110**
6.3. Expérimentations sur des données réelles **112**
6.4. Conclusions **113**

Afin de mesurer l'efficacité de notre approche RSF pour l'extraction d'arborescences fréquentes. celle-ci a été testée sur des bases de données synthétiques et réelles. Nous avons aussi testé les algorithmes FREQT [AAK+02] et VTREEMINER [Zak02] sur les mêmes jeux de données, afin de pouvoir comparer l'efficacité de RSF par rapport à ces algorithmes. Les expériences présentées dans ce chapitre mettent en évidence différentes mesures de performance des algorithmes mentionnés ci-dessus, en fonction des quatre mesures suivantes : le nombre d'arbres fréquents extraits, le nombre d'arbres candidats générés, le temps d'exécution et la consommation de mémoire.

6.1. Implémentation

Les expérimentations conduites dans ce chapitre ont été réalisées sur un Pentium cadencé à 1400 Mhz et disposant de 256 Mo de RAM et de 1024 Ko de mémoire cache. Le système d'exploitation installé sur ce poste a été un système Linux 2.6. Les algorithmes dans l'approche RSF ont été développés en C++ et compilés avec gcc 4.0. Pour la manipulation de données, nous avons employée les structures de données fournies par la bibliothèque STL[1].

Nus avons choisi FREQT et VTREEMINER comme algorithmes de référence car ils sont basés sur l'approche *Apriori*. Nous avons utilisé, pour le premier algorithme, l'implémentation (récursive) écrite en langage C++ par Taku Kudo [Kud03], et pour le deuxième, nous avons employé le code en C++ mis gracieusement à notre disposition par M. Zaki (disponible aussi sur [Zak]). L'emploi des algorithmes développés sur un langage commun nous a permis d'utiliser une même fonction (i.e. la fonction *gettimeofday()* qui fournit une résolution temporelle de l'ordre de la micro-seconde) pour mesurer le temps de réponse dans le processus de fouille d'arbres fréquents. La consommation de mémoire à été rapportée dans tous les algorithmes par la fonction *mallinfo()* avec laquelle nous obtenons des informations sur l'attribution de mémoire dynamique.

6.2. Expérimentations sur des données synthétiques

6.2.1. Jeux de données

Nous avons généré une base de données synthétiques construites en employant le programme de génération d'arbres XML proposé par [TRS02]. Ce programme permet l'application de différents paramètres pour spécifier différentes contraintes telles que le nombre d'arbres à générer, la profondeur des arbres, le nombre d'étiquettes maximales, etc. Les différentes valeurs utilisées pour les génération des bases de données synthétiques lors de nos expérimentations sont indiquées dans le tableau 6.1.

[1] Standard Template Library

TAB. 6.1 – Paramètres pour la construction de la base de données synthétiques.

Paramètres	Valeurs
Profondeur maximale d'un arbre	5
Nombre maximal de branches par nœud	5
Nombre maximal d'étiquettes	50
Arbres fréquents semés dans la base générée	10
Probabilité qu'un nœud soit père	0.4

Fouille d'arbres avec une inclusion *induite*

Pour la réalisation de notre première expérience, nous avons utilisé une base de données composée de 10000 arbres générés artificiellement. Le support minimal a été pris dans l'intervalle entre 0.01 et 0.9. L'objectif consiste à comparer l'extraction d'arbres fréquents ordonnés en utilisant une définition d'inclusion *induite* ; à cet effet on a décidé de comparer les algorithmes FREQT et RSF. Pour l'algorithme RSF nous présentons les résultats avec une génération de candidats *traditionnelle* (c.f. Section 5.3.1), notée simplement RSF et nous obtenons aussi des résultats pour l'approche RSF en utilisant la méthode PIVOT (c.f. Section 5.3.2) pour la génération optimisée de candidats.

La figure 6.1(a) illustre le nombre d'arbres fréquents extraits pour les trois algorithmes. Cette courbe permet de confirmer d'une façon naïve la complétude de notre approche RSF car ici nous constatons qu'on extrait le même nombre d'arbres fréquents par rapport aux autres approches (La preuve formelle se trouve dans la section 5.3.3). La figure 6.1(b) illustre le nombre d'arbres candidats générés. Nous remarquons que le nombre de candidats produits par RSF (surtout avec PIVOT) est toujours au dessous de notre approche de référence. Malgré le nombre de candidats plus petit de RSF, le temps d'exécution de notre algorithme dans les deux versions (c.f. Figure 6.2(a)), est au-dessus du temps de réponse de FREQT. Cette différence est expliquée par l'utilisation des *listes d'occurrences* qui optimisent la recherche d'arbres fréquents dans la base de données ; mais qui en revanche augmentent la consommation de mémoire comme l'illustre la figure 6.2(b).

Fouille d'arbres avec une inclusion *incrustée*

Dans le cadre d'un inclusion de type *incrustée*, c'est-à-dire qui respecte les relations *ancêtre-descendant*, afin de comparer notre approche en conditions similaires d'inclusion d'arbres, nous avons utilisé l'algorithme VTREEMINER. Nous avons mené nos expérimentations sur la même base de données synthétique de la section précédente avec une variation du support de 0.9 à 0.01.

La figure 6.3(a) montre qu'on obtient le même nombre d'arbres fréquents à partir des techniques différentes (RSF, PIVOT et VTREEMINER). Considérant l'inclusion d'arbres en

respectant les relations *ancêtre-descendant* entre nœuds, on peut voir que concernant la génération de candidats, vTreeMiner et Pivot, restent très proches (c.f. Figure 6.3(b)). Une nouvelle fois l'emploi de listes d'étendue (voir Section 3.3.2) permettent à vTreeMiner de réussir un meilleur temps de réponse avec un sur-coût en l'utilisation de mémoire (c.f. Figure 6.4).

6.3. Expérimentations sur des données réelles

6.3.1. Jeux de données

Dans cette section nous présentons les résultats de nos expérimentations effectuées sur la base de données DBLP[2] disponible sur [oCS&EUoW02]. La base DBLP fournit des informations bibliographiques sur les principaux journaux et conférences en informatique. Cette base de données est constituée de 328458 références bibliographiques.

Fouille d'arbres avec une inclusion *induite*

Dans un premier temps, nous reportons ici les résultats expérimentaux obtenus en comparant les résultats de l'algorithme Freqt et Rsf à ceux obtenus en incluant une génération optimisée des candidats par la méthode Pivot. Nous souhaitons, comme dans le cas des données synthétiques, évaluer notre proposition selon deux aspects : temps de réponse et occupation mémoire. En fait, les expérimentations réalisées prouvent que notre proposition répond aux deux critères. Pour évaluer les performances de nos algorithmes, nous les avons comparés à l'algorithme Freqt en permettant la recherche des inclusions *induites*.

La figure 6.5(a) représente le nombre de candidats extraits par les trois algorithmes en faisant varier le support minimal de 0.9 à 0.01. La figure 6.5(b) nous montre que le nombre de candidats générés est très nettement inférieur avec notre méthode Pivot. Par contre, l'algorithme Rsf explose à partir d'un support 0.01.

La figure 6.6(a) indique les temps d'exécution obtenus pour les trois algorithmes pour différents supports. Nous pouvons constater que Pivot obtient une performance de temps similaire au temps de réponse de la méthode Freqt. La figure 6.6(b) représente l'occupation mémoire utilisée pour la représentation de la base de schémas XML. Comme nous nous y attendions Rsf et Pivot occupent moins d'espace mémoire puisqu'ils adoptent une structure de représentation plus réduite que Freqt.

Fouille d'arbres avec une inclusion *incrustée*

Dans un deuxième temps, nous présentons nos résultats expérimentaux obtenus en comparant les résultats de nos algorithmes Rsf et Pivot avec vTreeMiner.

[2]Digital Bibliography & Library Project

Dans la figure 6.7(a) nous pouvons constater la validité de nos algorithmes car les trois algorithmes extraient la même quantité d'arbres fréquents. La figure 6.7(b) montre l'évolution du nombre de candidats en fonction du support pour une valeur de support comprise entre 0.9 et 0.01. Comme l'illustre la courbe, le nombre de candidats générés par notre méthode Pivot reste légèrement inférieur au nombre de candidats générés par vTreeMiner.

Pour évaluer les performances sur les temps d'exécution, nous présentons la figure 6.8(a). Sur celle-ci, il s'avère que les temps d'exécution sont nettement améliorés pour Pivot. On trouve qu'à partir d'un support inférieur à 0.09, le temps d'exécution de vTreeMiner explose (comportement qui a lieu aussi avec Rsf). La figure 6.8(b) représente l'occupation mémoire utilisée pour les trois algorithmes. Comme nous nous y attendions Rsf et Pivot occupent moins d'espace mémoire puisqu'ils adoptent une structure de représentation plus réduite et nous ne stockons pas de résultats temporaires dans le processus de validation de l'inclusion des sous-arbres candidats (i.e. des listes d'étendue).

6.4. Conclusions

Dans ce chapitre nous avons montré les résultats obtenus au cours de nos expérimentations sur des données artificielles et réelles.

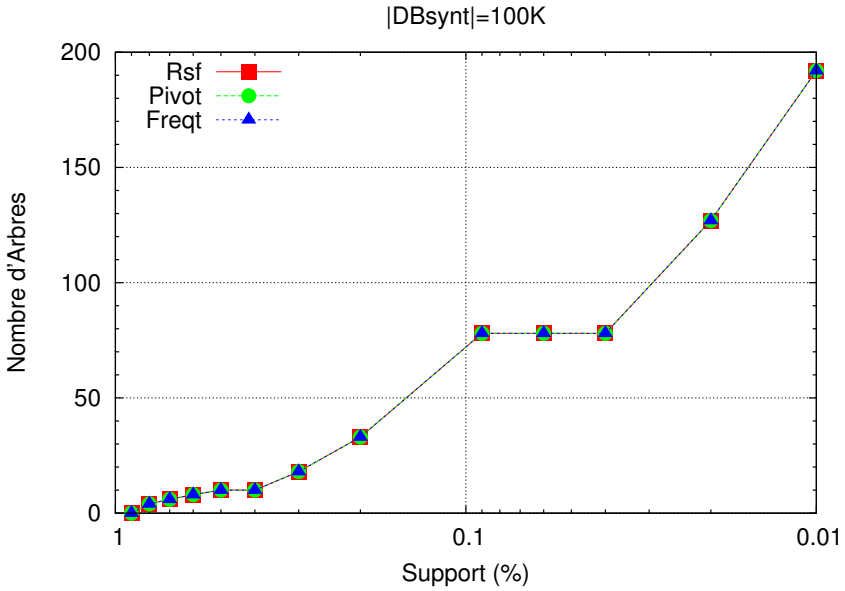
Premièrement, nous réaffirmons l'intérêt de notre approche Pivot car particulièrement dans les deux bases de données sélectionnées pour lancer nos tests, nous avons obtenu une diminution très significative du nombre d'arbres candidats par rapport aux approches de référence.

Deuxièmement, nous constatons que le modèle de données que nous proposons pour la représentation des arborescences dans le chapitre 4 permet à notre approche Rsf de maintenir une consommation de mémoire toujours au-dessous des approches Freqt et TreeMiner.

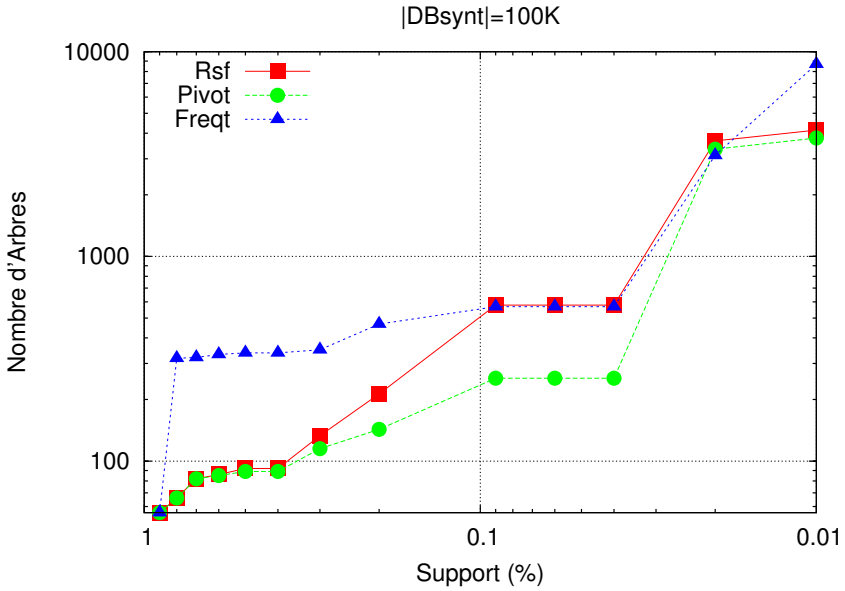
Troisièmement, concernant les temps d'exécution, dans certains cas où les arbres de la base de données sont construits en profondeur, nous notons que la représentation compacte des arborescences permet une diminution de l'utilisation de la mémoire au détriment de la vitesse de nos algorithmes (c.f. Figures 6.2(a) et 6.4(a)). En revanche, dans le cas des bases de données contenant des arbres arrangés en largeur (i.e. ayant une profondeur 1 ou 2), nos temps d'exécution sont très proches ou bien meilleurs (considérant des supports bas) comme l'illustre les figures 6.6(a) et 6.8(a).

Finalement, nous soulignons le fait que les algorithmes Freqt et TreeMiner se sont arrêtés aux supports assez bas en raison d'une grosse consommation de mémoire (c.f. Figures 6.6(a) et 6.8(a)). En conséquence, cette consommation de mémoire ne permet pas l'application de ces techniques dans certains scénarios réels. En revanche, notre approche Rsf optimisée avec la méthode Pivot, a continué à travailler au-dessous des supports qui représentent une limitation pour les autres approches. En d'autres termes, une faible

et constante consommation de mémoire nous permet l'extraction d'arbres fréquents en considérant un support assez bas.

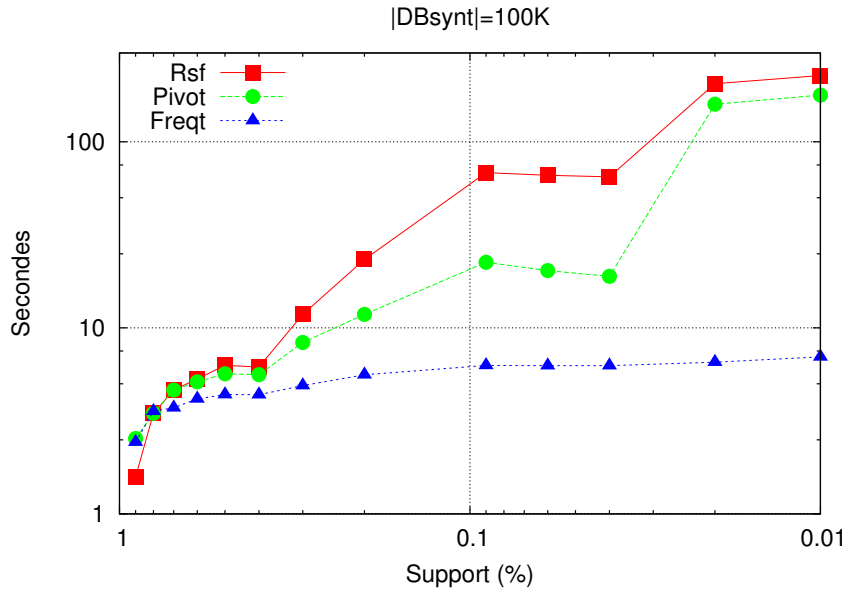


(a) Fréquents

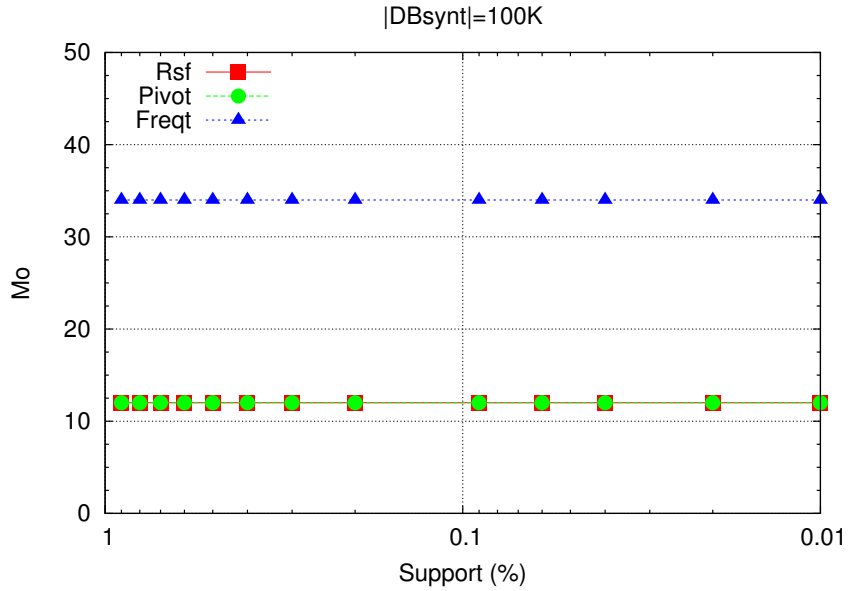


(b) Candidats

FIG. 6.1 – Nombre d'arbres sur une base de données synthétique et une inclusion *induite*.

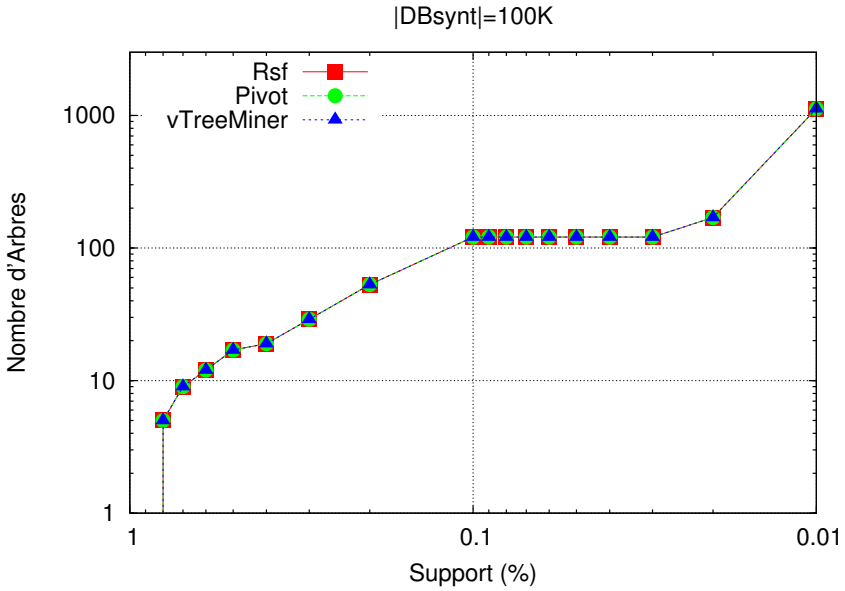


(a) Temps d'exécution

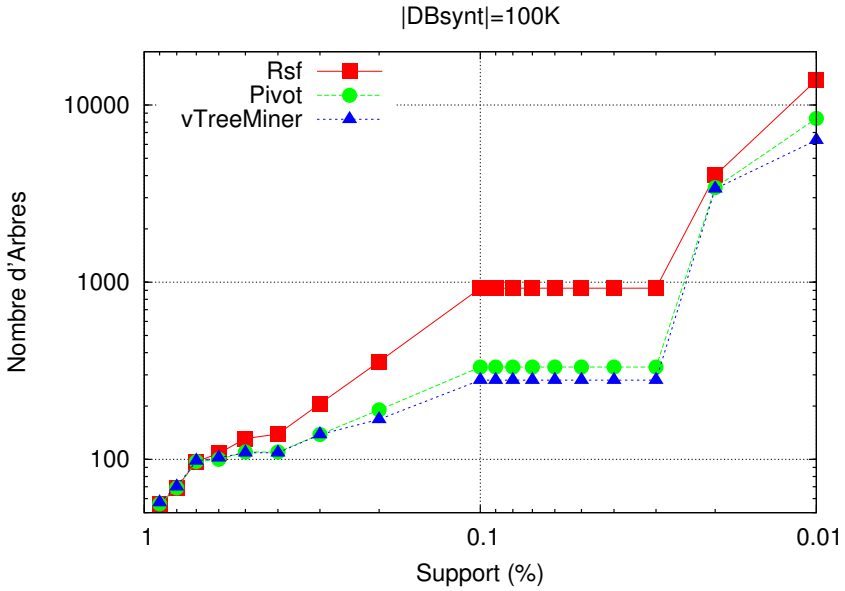


(b) Mémoire utilisée

FIG. 6.2 – Variation de temps et mémoire sur une base de données synthétique et une inclusion *induite*.

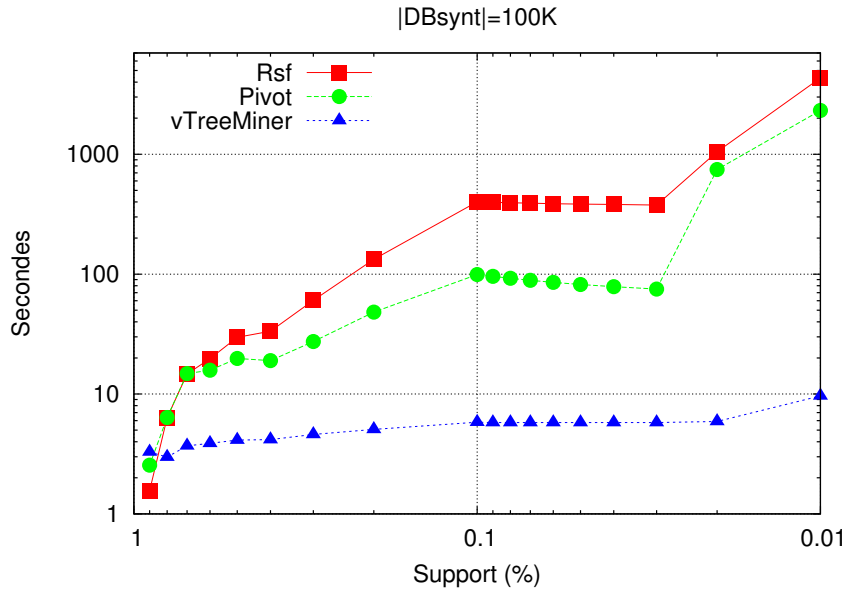


(a) Fréquents

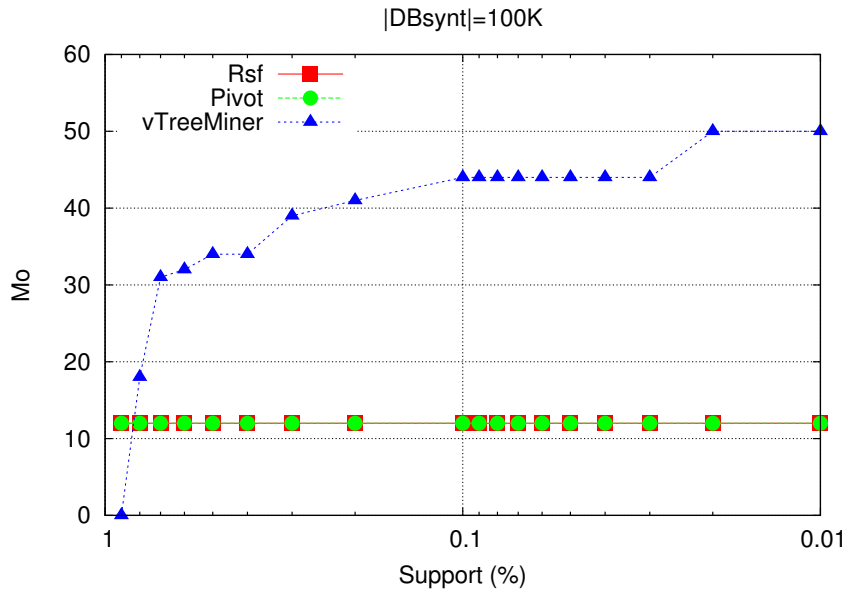


(b) Candidats

FIG. 6.3 – Nombre d'arbres sur une base de données synthétique et une inclusion *incrustée*.

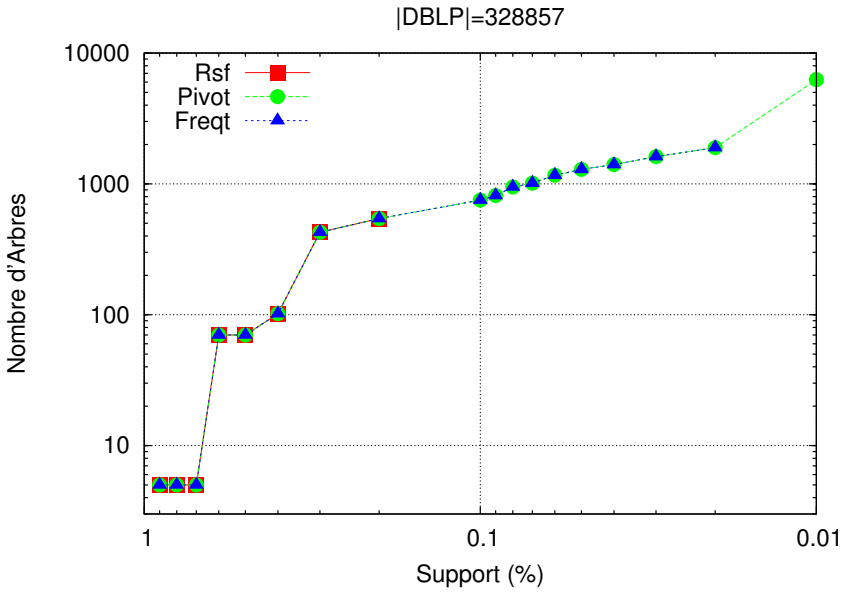


(a) Temps d'exécution

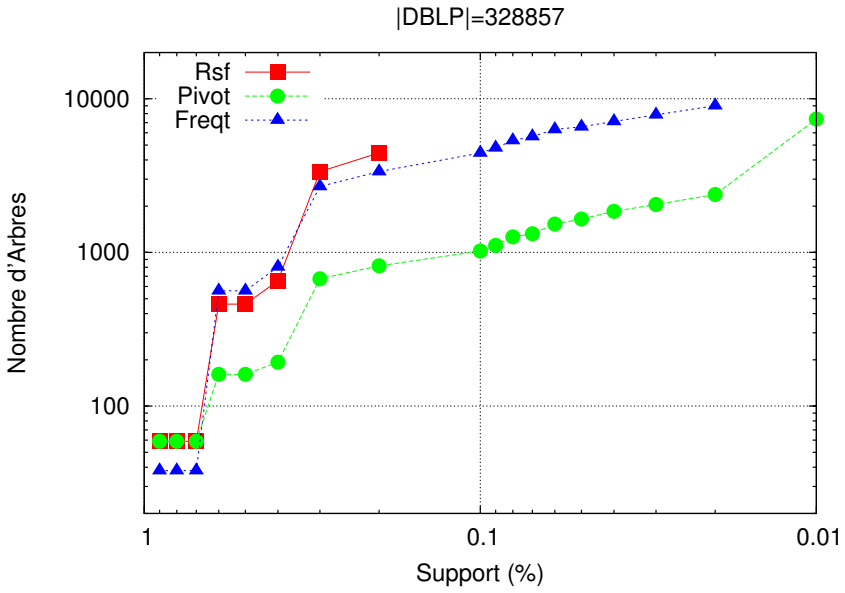


(b) Mémoire utilisée

FIG. 6.4 – Variation de temps et mémoire sur une base de données synthétique et une inclusion *incrustée*.

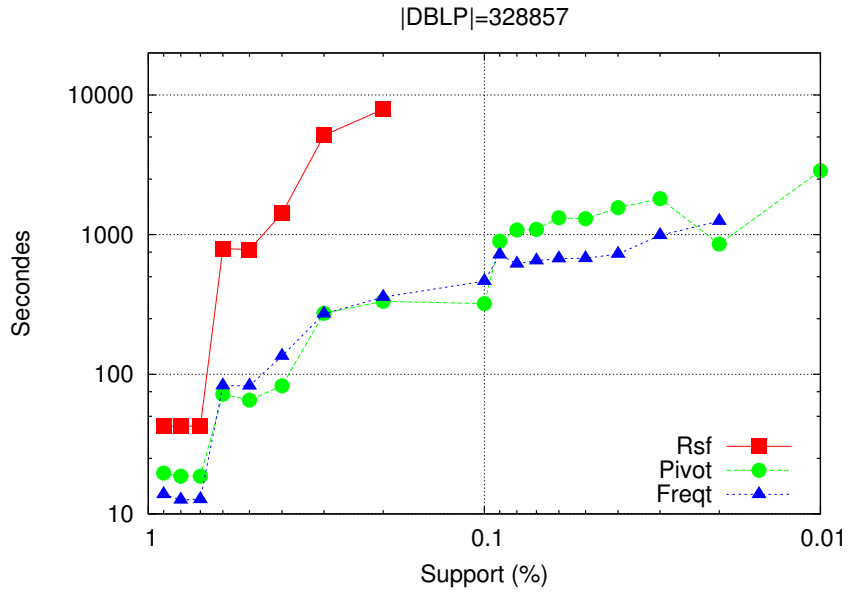


(a) Fréquents

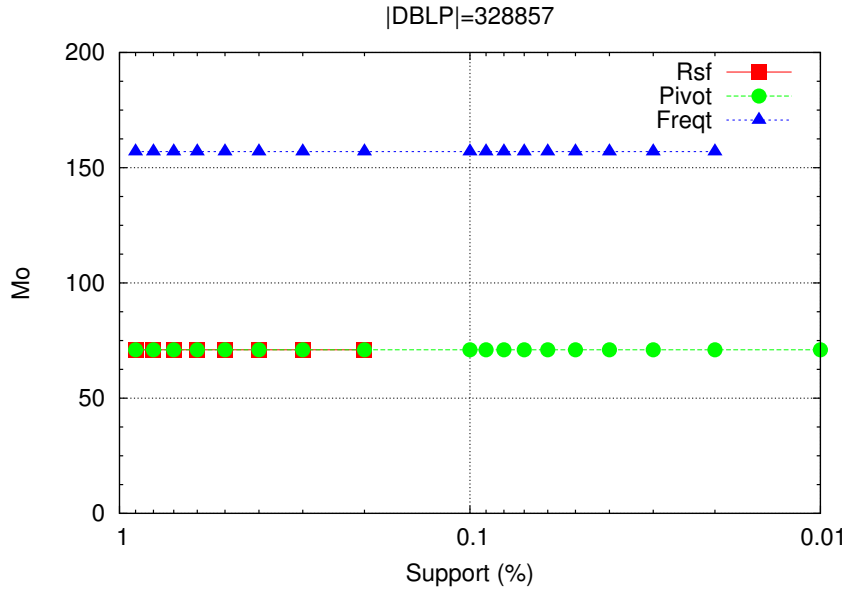


(b) Candidats

FIG. 6.5 – Nombre d'arbres sur la base DBLP en considérant une inclusion *induite*.

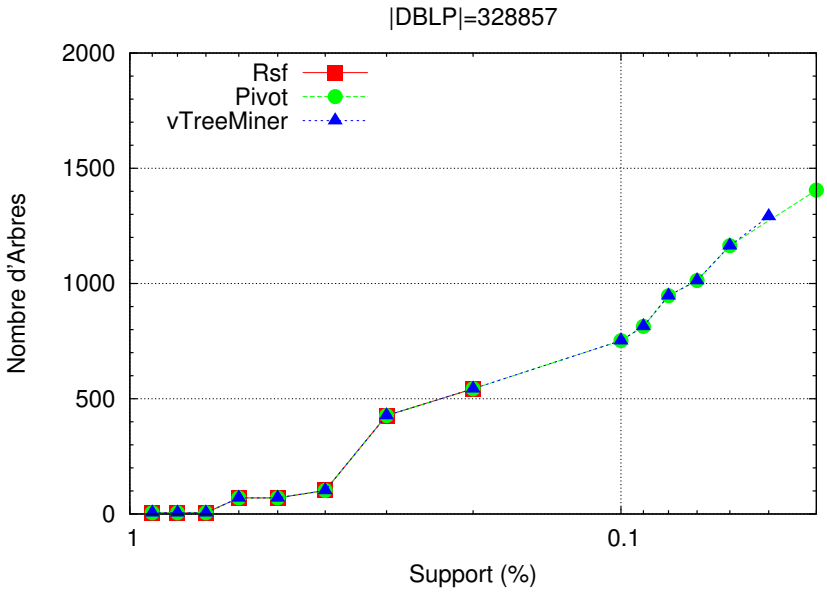


(a) Temps d'exécution

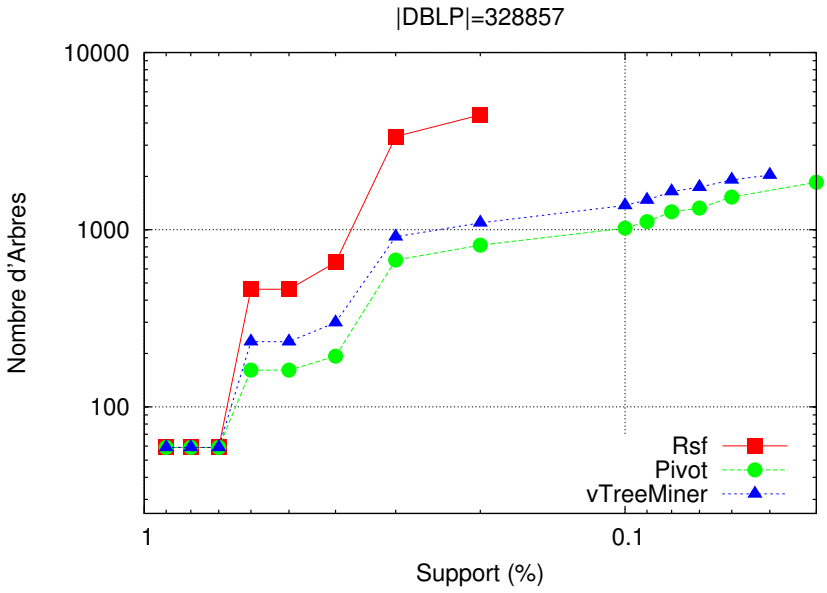


(b) Mémoire utilisée

FIG. 6.6 – Variation de temps et mémoire sur la base DBLP en considérant une inclusion *induite*.

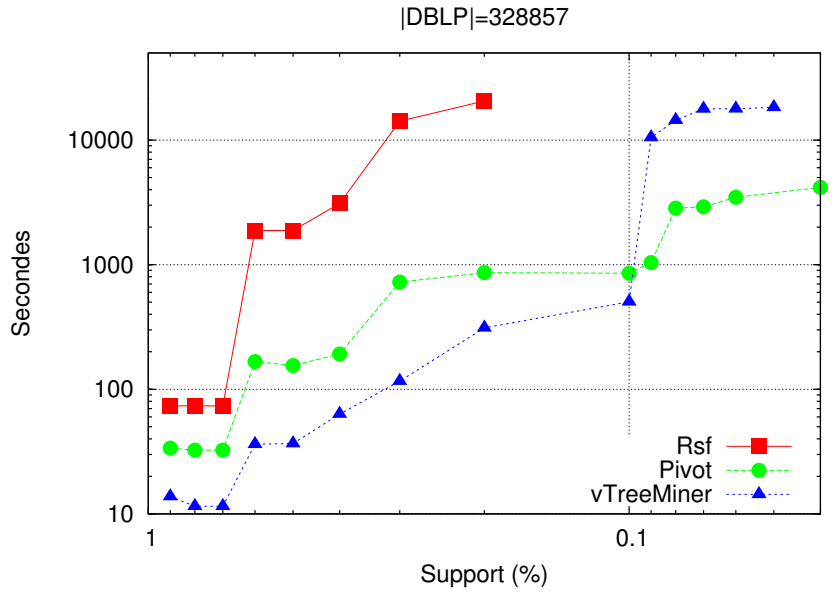


(a) Fréquents

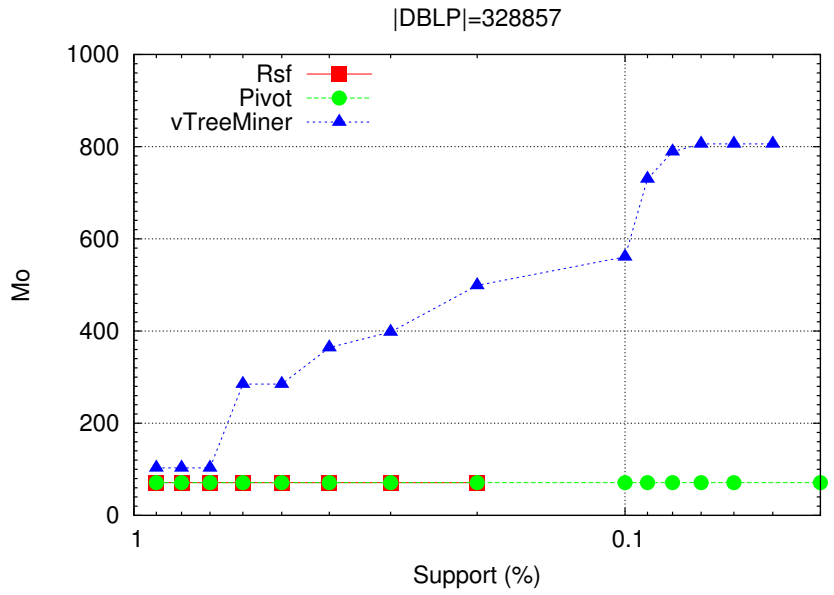


(b) Candidats

FIG. 6.7 – Nombre d'arbres sur la base de données DBLP et une inclusion *incrustée*.



(a) Temps d'exécution



(b) Mémoire utilisée

FIG. 6.8 – Variation de temps et mémoire sur la base de données DBLP en considérant une inclusion *incrustée*.

Chapitre 7

Conclusion et Perspectives

7.1. Introduction

Rappel de la problématique. La recherche de structures fréquentes ou fouille d'arbres, est devenue un problème de recherche important pour la communauté fouille de données. Les principales raisons de cet intérêt sont liées à l'augmentation croissante de données semi-structurées (*e.g.* bases de données XML) et à la nécessité d'extraire de la connaissance de ces bases de données.

En considérant une base de données semi-structurée où chaque composant est représenté par une arborescence, le problème de la « fouille d'arbres » consiste à extraire les sous-arbres qui apparaissent fréquemment dans la base de données (*i.e.* les sous arbres dont le nombre d'occurrences est supérieur à un support minimal spécifié par l'utilisateur).

7.2. Contributions

Modèle de données. Dans ce mémoire nous abordons cette problématique en introduisant tout d'abord un modèle de données. Ce modèle est composé d'une interface de programmation et d'une structure de données pour le traitement et la représentation des arborescences dans le processus d'extraction d'arbres fréquents. Le modèle de données a pour objectif la définition d'une structure de données pour la représentation de données arborescentes dont les propriétés permettent le traitement efficace des arbres. À partir de ce modèle de données, deux représentations ont été proposées :

Une représentation vectorielle : la particularité de cette représentation est qu'elle utilise un espace de mémoire égal à deux fois la taille d'un arbre dans la base de données. Cette représentation est bien adaptée surtout quand on extrait des arborescences en considérant une inclusion *induite*, car les relations directes *père-fils* entre les nœuds des arbres sont stockées de façon récursive. Les propriétés de cette représentation sont

aussi exploitées dans la phase de génération de candidats soit dans l'approche classique, soit dans l'approche PIVOT où on fait grandir les arbres par la branche la plus à droite.

Une représentation binaire : cette représentation possède les avantages de la représentation vectorielle, mais, dans ce cas nous transformons les arbres dans la base de données (des arbres à arites arbitraires[1]) comme des arbres binaires. Une fois les arbres transformés, leurs nœuds sont encodés selon un codage Huffman (légèrement modifié). Cette représentation permet de déterminer efficacement (par des manipulations de bits) si un nœud se trouve à la portée d'un autre nœud. La représentation binaire que nous proposons est bien adaptée à l'extraction d'arbres selon une inclusion de type *incrustée* ou *floue*.

Comme nous l'avons vu dans la section 2.3.2, l'extraction d'arbres fréquents, basée sur l'approche APRIORI [AIS93], est un processus itératif ou récursif qui peut être divisé en deux étapes : 1) la génération de candidats et 2) la validation de candidats.

Génération de candidats. En ce qui concerne l'étape de génération de candidats, plusieurs approches ont été proposées pour optimiser cette étape. Cependant ces méthodes génèrent un grand nombre de candidats. Or, il s'agit là d'une étape cruciale des algorithmes de fouilles de motifs et en minimisant le nombre de candidats nous permettons d'améliorer les temps de réponse. Nous proposons donc de réduire le nombre de candidats générés, en utilisant la méthode PIVOT. Notre solution est fondée sur la construction de classes d'équivalence. Ces classes sont définies à partir d'un arbre appelé *pivot* qui rassemble les arbres dans une classe d'équivalence. A tout ensemble d'arbres de taille k, il est alors facile d'associer l'ensemble des pivots qui les régissent et de construire des classes d'équivalence. A partir de ces classes d'équivalence, nous pouvons alors construire les candidats de taille $k+1$ en combinant deux arbres partageant le même pivot et en insérant les deux nœuds spécifiques, de manière très similaire aux méthodes par niveau dans le cadre de la fouille de données (itemsets). Ainsi, par la méthode PIVOT, nous réduisons le nombre de candidats générés en nous assurant que tout pivot d'un arbre candidat est un fréquent.

Validation du support. L'étape de validation des sous arbres est l'étape la plus coûteuse en temps de calcul dans la mesure où chaque candidat doit être testé sur la base de données. En ce qui nous concerne, à cette étape, nous avons abordé le problème de l'extraction d'arbres en considérant une inclusion de type *floue*, autrement dit, une inclusion progressive ou partielle d'un arbre dans un autre selon. Premièrement nous nous concentrons sur les relations *verticales* ou *ancêtre-descendant* des nœuds d'un arbre, en considérant un degré entre 0 et 1 pour indiquer dans quelle mesure un nœud est un ancêtre d'un autre nœud.

[1] en anglais *unranked trees*

Ensuite, nous nous intéressons aux *relations horizontales* (i.e. l'ordre entre nœuds frères) pour permettre l'existence d'un certain des-ordre entre nœuds frères d'un candidat tout en validant l'inclusion de ce candidat dans un arbre de la base de données. Puis, nous avons présenté une inclusion *partielle* qui valide l'inclusion d'un candidat dans un arbre de la base en considérant simplement une proportion des nœuds du candidat. Finalement, Nous avons proposé une inclusion en considérant les arbres *semblables* où nous permettons un degré indiquant à quel point deux nœuds sont semblables selon ses étiquettes.

Ces différents propositions et améliorations du processus de la fouille d'arbres ont été comparées (représentation compacte des arbres, génération efficace des candidats et occupation de mémoire constante et réduite lors du processus d'extraction) aux méthodes proposées dans la littérature. Nous mettons en évidence leur intérêt grâce à des expérimentations menées sur des bases de données synthétiques et réelles.

7.3. Perspectives

Nous proposons dans ce travail une approche d'extraction de sous-arbres fréquents. L'approche RSF est une proposition de recherche de sous-arbres fréquents selon une inclusion induite, incrustée ou floue à l'aide d'une représentation de la base de schémas en deux fois la taille des arbres. A l'issue de cette thèse, les perspectives sont nombreuses. Nous citons ci-dessus quelques unes d'entre elles.

Modèle de données. Comme perspective immédiate concernant la représentation de données nous pensons à une amélioration de notre approche PIVOT. Nous rappelons que cette méthode est basée sur la construction de trois classes d'équivalence : *droite*, *gauche* et *racine*. Si la méthode PIVOT nous permet de diminuer le nombre de candidats générés, en revanche, le nombre de classes d'équivalence générées peut croître. Il sera donc intéressant d'étudier comment traiter encore plus efficacement les classes d'équivalence en maintenant un bon rapport entre la mémoire et le temps d'exécution. L'utilisation de la structure de représentation des arbres basée sur un parcours en *pré-ordre*, comme proposé dans la section 4.3, est bien adaptée à la construction et traitement des classes à *droite*. Cependant, cette représentation, implique dans le cas des classes à *gauche* de parcours additionnels des vecteurs représentant les arborescences. Notre idée est donc d'intégrer un traitement des classes d'équivalence à gauche basé sur un encodage *Prüfer* [Prü18] construit à partir d'un parcours en *post-ordre* des arborescences.

Médiation de données. Étant donnée l'explosion du volume de données disponibles sur Internet, il devient indispensable de proposer de nouvelles approches pour faciliter l'interrogation de ces grandes masses d'information afin de retrouver les informations souhaitées. L'une des conditions sine qua non pour permettre d'interroger des données hétérogènes est

de disposer d'un (ou de plusieurs) *schéma général* que l'utilisateur pourra interroger et à partir duquel les données sources pourront être directement accédées. Malheureusement les utilisateurs ne disposent pas de moyen de connaître les modèles sous-jacents des données qu'ils souhaitent accéder et l'un des challenges dans ce contexte est donc de fournir des outils pour extraire, de manière automatique, ces schémas médiateurs. Un schéma médiateur est alors considéré comme une interface permettant à l'utilisateur l'interrogation des sources de données : l'utilisateur pose ses requêtes de manière transparente et n'a pas à tenir compte de l'hétérogénéité et de la répartition des données. Dans l'optique de la médiation de données, nous dirigeons nos perspectives, à moyen et long terme, vers la construction automatique de schémas médiateurs où les arbres fréquents extraits serviront comme modèles communs qui intègrent des données provenants de sources hétérogènes. L'étude des différents types d'inclusion et notamment de l'inclusion floue nous permettra de prendre en compte les imperfections des données réelles et de proposer des solutions pertinentes et efficaces.

Fouille de données en ligne. Dans le cadre de la *fouille de données en ligne*[2], notre approche RSF, de même que les approches de type *générer-élaguer*, ne peuvent pas être adaptées directement, car on n'a accès aux données qu'une seule fois (au moment d'essayer de valider si un candidat est fréquent ou non, la base de données originale aura disparu). Cependant dans ce domaine on est toujours à la recherche d'une représentation efficace en mémoire des motifs fréquents. Nous envisageons donc que la structure de représentation des arborescences que nous proposons soit adoptée dans le cadre de la fouille de données en ligne pour le traitement à la volée de données sous la forme de flots. Dans ce cas en particulier, nous pouvons utiliser nos structures pour stocker tous les motifs fréquents et pour les mettre à jour facilement en fonction des données du flot.

Par ailleurs, une notion importante employée dans les flots de données sont les *résultats approximatifs* (e.g. on sait qu'il y a un an les clients ont à peu près acheté 20% de tel produit). A ce propos G. Giannella dans [GHP+02] a proposé la notion de *Tilted-time windows* qui permet de regrouper automatiquement l'historique des fréquents en les agrégeant. L'avantage de cette approche est qu'on connaît ce qui se passe sur le flot avec une très bonne précision mais on est aussi capable de savoir ce qu'il s'est passé avant avec une bonne approximation. Cette technique de *Tiltled-time* pourrait être adaptée à notre approche RSF pour l'analyse de données XML à la volée.

Fouille d'arbres selon le principe non-derivable. Afin d'améliorer l'efficacité de notre approche basée actuellement sur APRIORI, nous envisageons l'application du principe NON-DERIVABLE proposé par T. Calders [CG02] pour la fouille d'itemsets fréquents étendu aux données semi-structurées. La méthode NON-DERIVABLE, cherche à élaguer plus

[2] en anglais *data streams*

efficacement les éléments candidats par rapport aux techniques appuyées sur la propriété d'anti-monotonie.

En autre, il serait intéressant d'examiner et de considérer les travaux de R. Ali Mohammadzadeth dans [AZRC06] où est mise en évidence une contradiction dans la propriété d'anti-monotonie quand on considère le *support pondéré* au moment de calculer la fréquence des arbres candidats.

Bibliographie

[AAK+02] Tatsuya Asai, Kenji Abe, Shinji Kawasoe, Hiroki Arimura, and Horoshi Sakamoto. Efficient substructure discovery from large semi-structure data. In *2nd Annual SIAM Symposium on Data Mining, SDM2002*, Arlington, VA, USA, 2002. Springer-Verlag.

[AAUiN03] Tatsuya Asai, Hiroki Arimura, Takeaki Uno, and Shin ichi Nakano. Discovering frequent substructures in large unordered trees. 2003.

[Abi97] Serge Abiteboul. Querying semi-structured data. In *ICDT*, pages 1–18, 1997.

[AFS93] Rakesh Agrawal, Christos Faloutsos, and Arun N. Swami. Efficient similarity search in sequence databases. In David B. Lomet, editor, *FODO*, volume 730 of *Lecture Notes in Computer Science*, pages 69–84. Springer, 1993.

[AGTF96] Jay Ayres, Johannes Gehrke, Tomi Tiu, and Jason Flannick. Sequential pattern mining using a bitmap representation. *Fuzzy Sets and Systems*, 84 :143–153, 1996.

[AIS93] Rakesh Agrawal, Tomasz Imielinski, and Arun N. Swami. Mining association rules between sets of items in large databases. In Peter Buneman and Sushil Jajodia, editors, *Proceedings of the 1993 ACM SIGMOD International Conference on Management of Data*, pages 207–216, Washington, D.C., 26–28 1993.

[ALSS95] Rakesh Agrawal, King-Ip Lin, Harpreet S. Sawhney, and Kyuseok Shim. Fast similarity search in the presence of noise, scaling, and translation in time-series databases. In Umeshwar Dayal, Peter M. D. Gray, and Shojiro Nishio, editors, *VLDB*, pages 490–501. Morgan Kaufmann, 1995.

[AMS+96] Rakesh Agrawal, Heikki Mannila, Ramakrishnan Srikant, Hannu Toivonen, and A. Inkeri Verkamo. Fast discovery of association rules. *Advances in Knowlage Discovery and Data Mining*, pages 307–328, 1996.

[AS94] Rakesh Agrawal and Ramakrishnan Srikant. Fast algorithms for mining association rules. In Jorge B. Bocca, Matthias Jarke, and Carlo Zaniolo,

editors, *Proc. 20th Int. Conf. Very Large Data Bases, VLDB*, pages 487–499, Santiago, Chile, 12–15 1994. Morgan Kaufmann.

[AS01a] Laurent Alonso and René Schott. On the tree inclusion problem. *Acta Inf.*, 37(9) :653–670, 2001.

[A.S01b] Clifford A.Shaffer. *Practical Introduction to Data Structures and Algorithm Analysis (C++ Edition).* Prentice Hall, 2001.

[AZRC06] Rahman AliMohammadzadeh, Ashkan Zarnani, Masoud Rahgozar, and Mostafa H. Chehreghani. Complete discovery of weighted frequent subtrees in tree-structured datasets. *International Journal of Computer Science and Network Security (IJCSNS)*, 6(8A) :188–196, 2006.

[BFSO84] Leo Breiman, Jerome Friedman, Charles J. Stone, and R. A. Olshen. *Classification and Regression Trees.* Chapman & Hall/CRC, January 1984.

[BG05] Philip Bille and Inge Li Gørtz. The tree inclusion problem : In optimal space and faster. In Luís Caires, Giuseppe F. Italiano, Luís Monteiro, Catuscia Palamidessi, and Moti Yung, editors, *ICALP*, volume 3580 of *Lecture Notes in Computer Science*, pages 66–77. Springer, 2005.

[BLPP99] Elena Barcucci, Alberto Del Lungo, Elisa Pergola, and Renzo Pinzani. Eco : a methodology for the enumeration of combinatorial objects. *Journal of Difference Equations and Applications*, 5 :435–490, 1999.

[BMRB96] Bernadette Bouchon-Meunier, Maria Rifqi, and Sylvie Bothorel. Towards general measures of comparison of objects. *Fuzzy Sets and Systems*, 84(2) :143–153, 1996.

[BPSM98] Tim Bray, Jean Paoli, and C. M. Sperberg-McQueen. XML 1.0 recommendation. W3C recommendation, W3C, February 1998. http ://www.w3.org/TR/1998/REC-xml-19980210.

[Bun97] Peter Buneman. Semistructured data. In *PODS '97 : Proceedings of the sixteenth ACM SIGACT-SIGMOD-SIGART symposium on Principles of database systems*, pages 117–121, New York, NY, USA, 1997. ACM Press.

[CG02] Toon Calders and Bart Goethals. Mining all non-derivable frequent itemsets. In Tapio Elomaa, Heikki Mannila, and Hannu Toivonen, editors, *PKDD*, volume 2431 of *Lecture Notes in Computer Science*, pages 74–85. Springer, 2002.

[CGHK78] F. R. K. Chung, R. L. Graham, V. E. Hoggat, and M. Kleiman. The number of Baxter permutations. *Journal of Combinatorial Theory, Series A*, 24 :382–394, 1978.

[Che98] Weimin Chen. More efficient algorithm for ordered tree inclusion. *Journal of Algorithms*, 26(2) :370–385, 1998.

[CLR90] Thomas Cormen, Charles Leiserson, and Ronald Rivest. *Introduction to Algorithms*. The MIT press, 1990.

[CRNK05] Yun Chi, Richard R.Muntz, Siegfried Nijssen, and Joost N. Kok. Frequent subtree mining - an overview. *Fundamenta Informaticae XXI*, pages 1001–1038, 2005.

[CS90] Peter Checkland and Jim Scholes. *Soft systems methodology in action*. John Wiley & Sons, Inc., New York, NY, USA, 1990.

[CYM04a] Yun Chi, Yirong Yang, and Richard R. Muntz. Cmtreeminer : Mining both closed and maximal frequent subtrees. In *The Eighth Pacific-Asia Conference on Knowledge Discovery and Data Mining (PAKDD'04)*, 2004.

[CYM04b] Yun Chi, Yirong Yang, and Richard R. Muntz. Hybridtreeminer : An efficient algorithm for mining frequent rooted trees and free trees using canonical forms. Technical report, UCLA Computer Science Department, 2004.

[Dea03] Enrica Duchi and Andrea Frosini et al. A note on rational succession rules. *Journal of Integer Sequences*, 6(03.1.7), 2003.

[Dze03] Saso Dzeroski. Multi-relational data mining : an introduction. *SIGKDD Explorations*, 5(1) :1–16, 2003.

[FFDB02] Stefano Ferilli, Nicola Fanizzi, Nicola Di Mauro, and Teresa M.A. Basile. Efficient theta-subsumption under object identity. In F. Esposito and D. Malerba, editors, *Atti del Workshop AI*IA 2002 su Apprendimento Automatico : Metodi e Applicazioni dell'Ottavo Convegno della Associazione Italiana per l'Intelligenza Artificiale*, pages 59–68, 2002.

[FMM+06] Mary Fernández, Ashok Malhotra, Jonathan Marsh, Norman Walsh, and Marton Nagy. XQuery 1.0 and XPath 2.0 data model (XDM). Candidate recommendation, W3C, July 2006. http ://www.w3.org/TR/2006/CR-xpath-datamodel-20060711/.

[FPSS96] Usama Fayyad, Gregory Piatetsky-Shapiro, and Padhraic Smyth. From data mining to knowledge discovery in databases. *Ai Magazine*, 17 :37–54, 1996.

[FRM94] Christos Faloutsos, M. Ranganathan, and Yannis Manolopoulos. Fast subsequence matching in time-series databases. In *Proceedings 1994 ACM SIGMOD Conference, Mineapolis, MN*, pages 419–429, 1994.

[GHP+02] C. Giannella, J. Han, J. Pei, X. Yan, and P. Yu. Mining frequent patterns in data streams at multiple time granularities, 2002.

[Han01] Jiawei Han. *Data Mining : Concepts and Techniques*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 2001.

[HKPT98] Yka Huhtala, Juha Kinen, Pasi Porkka, and Hannu Toivonen. Efficient discovery of functional and approximate dependencies using partitions. In *ICDE*, pages 392–401, 1998.

[HPY00] Jiawei Han, Jian Pei, and Yiwen Yin. Mining frequent patterns without candidate generation. In Weidong Chen, Jeffrey Naughton, and Philip A. Bernstein, editors, *2000 ACM SIGMOD Intl. Conference on Management of Data*, pages 1–12. ACM Press, 05 2000.

[HSM01] David J. Hand, Padhraic Smyth, and Heikki Mannila. *Principles of data mining*. MIT Press, Cambridge, MA, USA, 2001.

[JD88] Anil K. Jain and Richard C. Dubes. *Algorithms for clustering data*. Prentice-Hall, Inc., Upper Saddle River, NJ, USA, 1988.

[KF88] George J. Klir and Tina A. Folger. *Fuzzy Sets, Uncertainty, and Information*. Prentice Hall, 1988.

[Kil92] Pekka Kilpeläinen. *Tree Matching Problems with Applications to Structured Text Databases*. PhD thesis, University of Helsinki, 1992.

[KM91] Pekka Kilpeläinen and Heikki Mannila. The tree inclusion problem. In Samson Abramsky and T. S. E. Maibaum, editors, *TAPSOFT, Vol.1*, volume 493 of *Lecture Notes in Computer Science*, pages 202–214. Springer, 1991.

[KM95] Pekka Kilpelainen and Heikki Mannila. Ordered and unordered tree inclusion. *SIAM J. Comput.*, 24(2) :340–356, 1995.

[KMRS92] M. Kantola, H. Mannila, K-J. Räihä, and H. Siirtola. Discovering functional and inclusion dependencies in relational databases. *International Journal Intellence Systems*, 1 :591–607, 1992.

[Knu73] Donald E. Knuth. *The art of computer programming, Fudamental Algorithms*, volume 1. Addison-Wesley, 1973.

[Knu04] Donald E. Knuth. *The art of computer programming, Generating All Trees*, volume 4, fascicle 4. Addison-Wesley, 2004.

[Koh03] Ladislav J. Kohout. Defining homomorphisms and other generalized morphisms of fuzzy relations in monoidal fuzzy logics by means of bk-products. In *JCIS 2003 - 7th Joint Conf. on Information Sciences (Subsection : 9th Internat. Conf. on Fuzzy Theory and Technology)*, page 13, 2003.

[Kud03] Taku Kudo. Freqt : An implementation of freqt, 2003. http ://chasen.org/ taku/software/freqt/.

[LLPT05] Federico Del Razo López, Anne Laurent, Pascal Poncelet, and Maguelonne Teisseire. Rsf - a new tree mining approach with an efficient data structure.

In *Proceedings of the joint Conference : 4th Conference of the European Society for Fuzzy Logic and Technology (EUSFLAT 2005)*, pages 1088–1093, Barcelona, Spain, September 2005.

[LLPT06] Federico Del Razo López, Anne Laurent, Pascal Poncelet, and Maguelonne Teisseire. Recherche de sous-structures fréquentes pour l'intégration de schémas XML. In *Conférence Extraction et Gestion des Connaissances (EGC 2006)*, volume II, pages 487–498, Lille, France, January 2006.

[LLPT07a] Federico Del Razo López, Anne Laurent, Pascal Poncelet, and Maguelonne Teisseire. Fuzzy tree mining : Go soft on your nodes. In *Foundations of Fuzzy Logic and Soft Computing, 12th International Fuzzy Systems Association World Congress IFSA*, volume 4529 of *Lecture Notes in Computer Science*, pages 145–154. Springer, 2007.

[LLPT07b] Federico Del Razo López, Anne Laurent, Pascal Poncelet, and Maguelonne Teisseire. Pivot : Equivalence classes-based optimized generation of candidates for tree mining. Technical report, LIRMM, June 2007.

[LLT05] Federico Del Razo López, Anne Laurent, and Maguelonne Teisseire. Représentation efficace des arborescences pour la recherche de sous-structures fréquentes. In *Deuxième atelier sur la Fouille de données complexes dans un processus d'extraction des connaissances*, pages 113–120, Paris, France, January 2005.

[LLT06] Federico Del Razo López, Anne Laurent, and Maguelonne Teisseire. Une représentation des arborescences pour la recherche de sous-structures fréquentes. *Revue des Nouvelles Technologies de l'Information. Numéro spécial Extraction des connaissances : Etat et perspectives.*, E-5 :299–308, 2006.

[LMP00] P. A. Laur, Florent Masseglia, and Pascal Poncelet. Schema mining : Finding structural regularity among semistructured data. In *PKDD '00 : Proceedings of the 4th European Conference on Principles of Data Mining and Knowledge Discovery*, pages 498–503, London, UK, 2000. Springer-Verlag.

[LSL+07] Federico Del Razo López, Stéphane Sanchez, Anne Laurent, Pascal Poncelet, and Maguelonne Teisseire. Data structures for efficient tree mining : from crisp to soft embedding constraints. *International Journal of Applied Mathematics and Computer Science(AMCS), AMCS Special Issue, Soft computing for information management on the Web*, To appear, 2007.

[MT92] Jiří Matoušek and Robin Thomas. On the complexity of finding iso- and other morphisms for partial k-trees. *Discrete Mathematics*, 108 :343–364, 1992.

[Nak02] Shin-Ichi Nakano. Efficient generation of plane trees. *Inf. Process. Lett.*, 84(3) :167–172, 2002.

[NC00] Noel Novelli and Rosine Cicchetti. Mining functional and embedded dependencies using free sets. In Anne Doucet, editor, *BDA*, 2000.

[NK03] Siegfried Nijssen and Joost N. Kok. Efficient discovery of frequent unordered trees. In *First International Workshop on Mining Graphs, Trees and Sequences*, 2003.

[NK04] Siegfried Nijssen and Joost N. Kok. A quickstart in frequent structure mining can make a difference. In *KDD '04 : Proceedings of the tenth ACM SIGKDD international conference on Knowledge discovery and data mining*, pages 647–652, New York, NY, USA, 2004. ACM Press.

[oCS&EUoW02] Department of Computer Science & Engineering University of Washington. Xmldata repository, 2002. http ://www.cs.washington.edu/research/xmldatasets/.

[Par04] David Parry. A fuzzy ontology for medical document retrieval. In *CRPIT '32 : Proceedings of the second workshop on Australasian information security, Data Mining and Web Intelligence, and Software Internationalisation*, pages 121–126. Australian Computer Society, Inc., 2004.

[PB02] Aymeric Perchant and Isabelle Bloch. Fuzzy morphisms between graphs. *Fuzzy Sets and Systems*, 128(2) :149 – 168, 2002.

[PH02] Jian Pei and Jiawei Han. Constrained frequent pattern mining : A pattern-growth view. *ACM SIGKDD*, 2(2), 2002.

[PHMA+01] Jian Pei, Jiawei Hana, Behzad Mortazavi-Asl, Helen Pinto, Qiming Chen, Umeshwar Dayal, and Mei-Chun Hsu. Prefixspan : mining sequential patterns efficiently by prefix-projected pattern growth. In *17th International Conference on Data Engineering*, 2001.

[Plo70] Gordon D. Plotkin. A note on inductive generalization. In *Machine Intelligence*, volume 5, pages 153–163. Edinburgh University Press, 1970.

[Prü18] Heinz Prüfer. Neuer beweis eines satzes fiber permutationen. *Archiv für Mathematik und Physik*, 27 :742–744, 1918.

[Ric97] Thorsten Richter. A new algorithm for the ordered tree inclusion problem. In A. Apostolico and J. Hein, editors, *Proceedings of the 8th Annual Symposium on Combinatorial Pattern Matching*, pages 150–166, Aarhus, Denmark, 1997. Springer-Verlag, Berlin.

[RK04] Ulrich Rückert and Stefan Kramer. Frequent free tree discovery in graph data. In Hisham Haddad, Andrea Omicini, Roger L. Wainwright, and Lorie M. Liebrock, editors, *SAC*, pages 564–570. ACM, 2004.

[SF01] Richard P. Stanley and Sergey Fomin. *Enumerative Combinatorics Volume 2*. Cambridge University Press, February 2001.

[SWZ04] Dennis Shasha, Jason T. L. Wang, and Sen Zhang. Unordered tree mining with applications to phylogeny. In *ICDE '04 : Proceedings of the 20th International Conference on Data Engineering*, page 708, Washington, DC, USA, 2004. IEEE Computer Society.

[TBBT04] John Tranier, Renaud Baraer, Zohra Bellahsene, and Maguelonne Teisseire. Where's charlie : Family based heuristics for peer-to-peer schema integration. In *Proceedings of the 8th International Database Engineering and Applications Symposium (IDEAS '04)*, Coimbra, Portugal, July, 7th - 9th 2004.

[TDH+06a] Henry Tan, Tharam S. Dillon, Fedja Hadzic, Elizabeth Chang, , and Ling Feng. IMB3-miner : Mining induced/embedded subtrees by constraining the level of embedding. In *The Eighth Pacific-Asia Conference on Know ledge Discovery and Data Mining*, 2006.

[TDH+06b] Henry Tan, Tharam S. Dillon, Fedja Hadzic, Elizabeth Chang, and Ling Feng. Imb3-miner : Mining induced/embedded subtrees by constraining the level of embedding. In *PAKDD*, pages 450–461, 2006.

[Ter04] Alexandre Termier. *Extraction d'arbres fréquents dans un corpus hétérogène de données semi-structurées : Application à la fouille de documents XML*. PhD thesis, Université Paris-Sud, avril, 2004.

[TPK06] Shirish Tatikonda, Srinivasan Parthasarathy, and Tahsin Kurc. Trips and tides : new algorithms for tree mining. In *CIKM '06 : Proceedings of the 15th ACM international conference on Information and knowledge management*, pages 455–464, New York, NY, USA, 2006. ACM Press.

[TRS02] Alexandre Termier, Marie-Christine Rousset, and Michele Sebag. Treefinder, a first step towards xml data mining. In *IEEE Conference on Data Mining (ICDM)*, pages 450–457, 2002.

[TRS04] Alexandre Termier, Marie-Christine Rousset, and Michèle Sebag. DRYADE : a new approach for discovering closed frequent trees in heterogeneous tree databases. . In *International Conference on Data Mining (ICDM 2004)*, pages 543–546, 2004.

[UGP96] Fayyad Usama, Piatetsky-Shapiro Gregory, and Smyth Padhraic. The kdd process for extracting useful knowledge from volumes of data. In *Communication of the ACM*, volume 29, pages 27–34, November 1996.

[Val02] Gabriel Valiente. *Algorithms on Trees and Graphs*. Springer-Verlag, Berlin, 2002.

[WCH+04] Lauren Wood, Mike Champion, Arnaud Le Hors, Steve Byrne, Gavin Nicol, Philippe Le Hégaret, and Jonathan Robie. Document object model (DOM) level 3 core specification. W3C recommendation, W3C, April 2004. http ://www.w3.org/TR/2004/REC-DOM-Level-3-Core-20040407.

[Wei98] Mark Allen Weiss. *Data Structures And Algorithm Analysis In C*. Addison Wesley, 1998.

[Wes96] Julian West. Generating trees and forbidden subsequences, 1996.

[WHP+04] Chen Wang, Mingsheng Hong, Jian Pei, Haofeng Zhou, Wei Wang, and Baile Shi. Efficient pattern-growth methods for frequent tree pattern mining. In *The Eighth Pacific-Asia Conference on Knowledge Discovery and Data Mining (PAKDD04)*, 2004.

[WK91] Sholom M. Weiss and Casimir A. Kulikowski. *Computer systems that learn : classification and prediction methods from statistics, neural nets, machine learning, and expert systems*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 1991.

[WYZ+04] Chen Wang, Qingqing Yuan, Haofeng Zhou, Wei Wang, and Baile Shi. Chopper : An efficient algorithm for tree mining. *Journal of Computer Science and Technology*, 19 :309–319, May 2004.

[Xyl01] Lucie Xyleme. A dynamic warehouse for xml data of the web. In *IEEE Data Engineering Bulletin*, 2001.

[Yag93] Ronald R. Yager. Families of OWA operators. *Fuzzy Sets and Systems*, 57(3) :125 – 148, 1993.

[YB97] M. Ying and B. Bouchon. Quantifiers, modifiers and qualifiers in fuzzy logic. *Journal of Applied Non-classical Logics*, 7(3) :335–342, 1997.

[Zad83] L. A. Zadeh. A computational approach to fuzzy quantifiers in natural languages. *Computing and Mathematics with Applications*, 9(1) :149–184, 1983.

[Zak] Mohammed J. Zaki. Treeminer code. http ://www.cs.rpi.edu/ zaki/software/.

[Zak02] Mohammed J. Zaki. Efficiently mining frequent trees in a forest. In *SIGKDD'02*, Edmonton, Alberta, Canada, 2002. ACM.

[Zak05] Mohammed J. Zaki. Efficiently mining frequent embedded unordered trees. *Fundamenta Informaticae*, 2005.

[ZPOL97] Mohammed J Zaki, Srinivasan Parthasarathy, Mitsunori Ogihara, and Wei Li. New algorithms for fast discovery of association rules. Technical report, University of Rochester, Rochester, NY, USA, 1997.

[ZW06] Sen Zhang and Jason Tsong-Li Wang. Mining frequent agreement subtrees in phylogenetic databases. In Joydeep Ghosh, Diane Lambert, David B. Skillicorn, and Jaideep Srivastava, editors, *SDM*. SIAM, 2006.

Bibliographie Personnelle

[LSL+ 07] Federico Del Razo López and Stéphane Sanchez and Anne Laurent and Pascal Poncelet and Maguelonne Teisseire. Data structures for efficient tree mining : from crisp to soft embedding constraints. In *International Journal of Applied Mathematics and Computer Science(AMCS), AMCS Special Issue, Soft computing for information management on the Web*, To appear 2007.

[LLPT07a] Federico Del Razo López and Anne Laurent and Pascal Poncelet and Maguelonne Teisseire. Fuzzy Tree Mining : Go Soft on Your Nodes. In *Foundations of Fuzzy Logic and Soft Computing, 12th International Fuzzy Systems Association World Congress IFSA-2007*, volume 4529 of Lecture Notes in Computer Science, pages 145-154, Springer, June 2007.

[LLT06] Federico Del Razo López and Anne Laurent and Maguelonne Teisseire. Une représentation des arborescences pour la recherche de sous-structures fréquentes. *Revue des Nouvelles Technologies de l'Information. Numéro spécial Extraction des connaissances : Etat et perspectives.*, volume E-5, pages 299-308, 2006.

[LLPT06] Federico Del Razo López and Anne Laurent and Pascal Poncelet and Maguelonne Teisseire. Recherche de sous-structures fréquentes pour l'intégration de schémas XML. In *Conférence Extraction et Gestion des Connaissances (EGC 2006)*, volume II, pages 487-498, Lille France, January 2006.

[LLPT05] Federico Del Razo López and Anne Laurent and Pascal Poncelet and Maguelonne Teisseire. RSF - A New Tree Mining Approach with an Efficient Data Structure. In *Proceedings of the joint Conference : 4th Conference of the European Society for Fuzzy Logic and Technology (EUSFLAT 2005)*, pages 1088-1093, Barcelona, Spain, September 2005.

[LLT05] Federico Del Razo López and Anne Laurent and Maguelonne Teisseire. Représentation efficace des arborescences pour la recherche de sous-structures fréquentes. In *Deuxième atelier sur la Fouille de données complexes dans un processus d'extraction des connaissances*, pages 113-120, Paris, France, January 2005.

Résumé

La recherche de structures arborescentes fréquentes, également appelée fouille d'arbres, au sein de bases de données composées de documents semi-structurés (e.g. XML) est un sujet de recherche actuellement très actif. Ce processus trouve de nombreux intérêts dans le contexte de la fouille de données, comme par exemple la construction automatique d'un schéma médiateur à partir de schémas XML, ou bien l'analyse des structures des sites Web afin d'étudier leur usage ou modifier leur contenu.

L'objectif de cette thèse est de proposer une méthode d'extraction d'arborescences ordonnées et étiquetées fréquentes. Cette approche est basée sur une représentation compacte des arborescences cherchant à diminuer la consommation de mémoire dans le processus de fouille. En particulier, nous présentons une nouvelle technique de génération d'arborescences candidates visant à réduire leur nombre. Par ailleurs, nous proposons différents algorithmes pour valider le support des arborescences candidates dans une base de données selon divers types de contraintes d'inclusion d'arbres : induite, incrustée et floue. Finalement nous appliquons nos algorithmes à des jeux de données synthétiques et réels, et nous présentons les résultats obtenus.

Mots-clés : Extraction de connaissances, Fouille de données, fouille d'arbres, XML, inclusion d'arbres, schéma médiateur, sous-arbres, énumération d'arborescences.