UNIVERSITE MONTPELLIER II
SCIENCES ET TECHNIQUES DU LANGUEDOC

THESE

Pour obtenir le grade de

DOCTEUR DE L'UNIVERSITE MONTPELLIER II

Discipline : INFORMATIQUE
Formation Doctorale : Informatique
Ecole Doctorale : Informatique

Présentée et soutenue publiquement

par

Pierre Alain Laur

le 30 août 2004 à 10h30

Données semi structurées : Découverte, Maintenance et Analyse de Tendances

JURY

M. Mohand-Said Hacid	**Rapporteur**
M. Dominique Laurent	**Rapporteur**
M. Pascal Poncelet	**Directeur de thèse**
Mme. Violaine Prince-Barbier	**Examinateur**
Mme. Maguelonne Teisseire	**Examinateur**

Remerciements

Je tiens tout d'abord à remercier Messieurs les professeurs Dominique Laurent et Mohand-Said Hacid, respectivement de l'Université Claude Bernard (Lyon I) et de Cergy-Pontoise, d'avoir accepté d'être rapporteurs de ce travail. Leurs remarques constructives ont été pour moi très enrichissantes.

Je remercie également Madame le professeur Violaine Prince-Barbier de l'Université des Sciences et Techniques du Languedoc (UM II) d'avoir accepté de siéger à ce jury et juger mon travail.

Je remercie tout particulièrement Pascal Poncelet et Maguelonne Teisseire, mes directeurs de thèse, qui m'ont permis de travailler au sein d'une équipe dynamique, dans laquelle leur rigueur scientifique et leur enthousiasme de chaque heure étaient toujours là pour me guider vers mes objectifs.

Je remercie Florent Masseglia pour son aide et ses nombreux conseils.

Une partie de ces travaux sur la définition de vues n'aurait pas pu aboutir sans une collaboration avec mon ami et collègue de bureau Xavier Baril, qu'il trouve ici l'expression de toute mon amitié.

Je remercie le Laboratoire d'Informatique de Robotique et de Microélectronique de Montpellier pour m'avoir fourni les ressources nécessaires pour mener à bien ce travail, et plus particulièrement les membres du département informatique du laboratoire, pour leur bonne humeur, leur enthousiasme qui ont contribué à rendre ce travail des plus agréable. Je remercie également le personnel administratif du laboratoire qui m'a toujours aidé quand j'en avais besoin.

Ces remerciements ne seraient pas complets si je ne citais pas l'équipe pédagogique informatique de l'Ecole Nationale Supérieure de Chimie de Montpellier au sein de laquelle j'ai effectué trois années de monitorat. Ces années m'ont permis de me former à l'enseignement et de participer activement au contenu de celui-ci.

Que Monsieur Olivier Cogis soit remercié pour la confiance qu'il m'a accordée lors d'une période difficile de mes études universitaires.

Que Nicolas Vidot, qui a toujours su répondre à mes innombrables questions techniques que ce soit en programmation ou dans bien d'autres domaines trouve dans ces quelques mots ma gratitude.

Merci à celles et ceux qui m'ont supporté et qui ont partagé avec moi un peu de ces années : tous mes amis (et ceux de mon canard), Thibaud, Claire…

Enfin, je remercie ma famille pour tous ce qu'elle a enduré et je dédie cette thèse à ma défunte grand-mère qui serait fière de son petit-fils.

Table des matières

Table des figures

Chapitre I - Introduction

Depuis ces dernières années, de nombreux travaux ont été réalisés pour extraire de la connaissance dans de grandes bases de données. Généralement connu sous le terme de Fouille de Données (*Data Mining*), l'objectif de ces approches consiste à appréhender le contenu des bases pour en faire émerger des modèles ou des motifs représentatifs qui peuvent servir à une prise de décision. La plupart des contributions existantes dans ce domaine concernent des données plates, i.e. des données dont la structure se résume à des types atomiques (entier, réel, …). Ainsi, par exemple, dans le contexte de la recherche de règles d'association, les données manipulées correspondent aux articles contenus dans les « paniers de la ménagère » et sont généralement représentés sous la forme d'entier. Depuis ces dernières années, notamment dans le cas du Web, de nombreux travaux ont également été réalisés pour permettre de représenter et stocker des informations de plus en plus complexes. Cette complexité concerne d'ailleurs aussi bien le contenu, i.e. le fond, que la forme. Il est en effet assez courant de retrouver des serveurs Web dont la structure des pages est complexe.
L'engouement autour de la fouille de données est dû à la quantité de données sans cesse croissante et à l'impossibilité de pouvoir facilement extraire de la connaissance. Aujourd'hui, nous nous retrouvons confronté au même problème mais dans le cas de données complexes.

Nous verrons par la suite qu'il existe déjà de nombreux domaines d'application qui nécessitent de retrouver dans de grandes bases de données semi structurées quelles sont les structures typiques qui existent. Cependant, avant de présenter quelles sont les problématiques associées à la recherche de structures fréquentes que nous aborderons au cours de ce mémoire, nous rappelons les principes généraux de l'extraction de connaissances et les principales techniques de fouilles de données.

1 L'extraction de connaissances dans les bases de données

Bien que le terme de fouille de données recouvre à lui seul la découverte de connaissances, il ne constitue qu'une seule des étapes du processus général de l'ECD [FaPi96] (Extraction de Connaissances dans les bases de Données), qui est défini comme un processus non trivial qui consiste à identifier, dans les données, des schémas nouveaux, valides, potentiellement utiles et surtout compréhensibles et utilisables. Ce processus comprend globalement trois phases :

- **Préparation des données** : l'objectif de cette phase consiste à sélectionner uniquement les données potentiellement utiles de la base. L'ensemble des données est ensuite soumis à un prétraitement, afin de gérer les données manquantes ou invalides (opération de nettoyage). L'étape suivante dans cette phase consiste à formater ces données, pour les rendre compréhensibles au processus de fouille de données (opérations de transformation et réduction).
- **Extraction** : en appliquant des techniques de fouilles de données, l'objectif de cette phase est de mettre en évidence des caractéristiques et des modèles contenus intrinsèquement et implicitement dans les données. Il s'agit également de proposer des modèles ou motifs représentatifs du contenu de la base.
- **Interprétation des résultats** : le but de cette dernière phase est d'interpréter la connaissance extraite lors de l'étape précédente, pour la rendre lisible et compréhensible par l'utilisateur et permettre ainsi de l'intégrer dans le processus de décision.

La Figure 1 représente le processus d'extraction de connaissances à partir de données.

Les techniques de fouille de données sont utilisées dans de nombreux domaines d'applications. Les exemples les plus courants sont les compagnies d'assurances, les compagnies bancaires (crédit, prédiction du marché, détection de fraudes), le marketing (comportement des consommateurs, « mailing » personnalisé), la recherche médicale (aide au diagnostic, au traitement, surveillance de la population sensible), les réseaux de communication (détection de situations alarmantes, prédiction d'incidents), l'analyse de données spatiales.

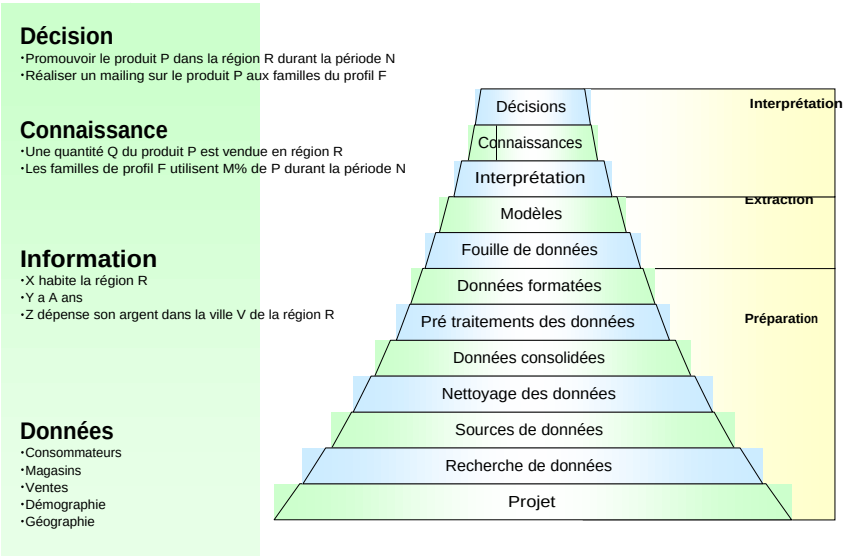


Figure 1 – Processus général d'extraction de connaissances à partir de données

Parmi les techniques les plus classiques, nous pouvons citer :

Les Règles d'Association
Le problème de recherche de règles d'association a fait l'objet de nombreux travaux ces dernières années [AgIm93, AgSr94, BrMo97, HoSw95, MaTo94, PaBa99, SaOm95, Toiv96, HaPe00, HaKa01]. Introduit dans [AgIm93] à partir du problème du panier de la ménagère (« *Market Basket Problem* »), il peut être résumé ainsi : étant donné une base de données de transactions (les paniers), chacune composée d'items (les produits achetés), la découverte d'associations consiste à chercher des ensembles d'items fréquemment liés dans une même transaction ainsi que des règles les combinant. Un exemple d'association pourrait révéler que « 75% des gens qui achètent de la bière, achètent également des couches ». Ce type de règles concerne un grand champ d'applications, telles que la conception de catalogues, la promotion des ventes, le suivi d'une clientèle, etc.

Les Motifs Séquentiels
Introduits dans [AgSr95], les motifs séquentiels peuvent être vus comme une extension de la notion de règles d'association intégrant diverses contraintes temporelles. La recherche de motifs consiste à extraire des ensembles d'items couramment associés sur une période de temps bien spécifiée. En fait, cette recherche met en évidence des associations inter transactions, contrairement à celle des règles d'association qui extrait des combinaisons intra transactions. Dans ce contexte, et contrairement aux règles d'association, l'identification des individus ou objets au cours du temps est indispensable afin de pouvoir suivre leur comportement. Par exemple, des motifs séquentiels peuvent montrer que « 60% des gens qui achètent une télévision, achètent un magnétoscope dans les deux ans qui suivent ». Ce problème, posé à l'origine par souci marketing, intéresse à présent des domaines aussi variés que les télécommunications (détection de fraudes), la finance ou encore la médecine (identification des symptômes précédant les maladies).

Les Dépendances Fonctionnelles
L'extraction de dépendances fonctionnelles à partir de données existantes est un problème étudié depuis de nombreuses années mais qui a été abordé récemment avec une « vision » fouille de données. Les résultats obtenus par ces dernières approches sont très efficaces [KaMa92, HuKa98, LoPe00]. La découverte de dépendances fonctionnelles est un outil d'aide à la décision à la fois pour l'administrateur de la base, les développeurs d'application, les 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.

La Classification
La classification, appelée également induction supervisée, consiste à analyser de nouvelles données et à les affecter, en fonction de leurs caractéristiques ou attributs, à telle ou telle classe prédéfinie [BrFr84, WeKu91]. Bien connus en apprentissage, les problèmes de classification intéressent également la communauté Base de

Données qui s'appuie sur l'intuition suivante : « plus les volumes de données traités sont importants, meilleure devrait être la précision du modèle de classification » [ShAg96]. Les techniques de classification sont par exemple utilisées lors d'opérations de « mailing » pour cibler la bonne population et éviter ainsi un nombre trop important de non réponse. De la même manière, cette démarche peut permettre de déterminer, pour une banque, si un prêt peut être accordé, en fonction de la classe d'appartenance d'un client.

La Segmentation
La technique de segmentation ressemble à celle de la classification, mais diffère dans le sens où il n'existe pas de classes prédéfinies [JaDu88] : l'objectif est de grouper des enregistrements qui semblent similaires dans une même classe. De nombreux algorithmes efficaces ont été proposés pour optimiser les performances et la qualité des classes obtenues dans de grandes bases de données [EsKr96, GuRa98a, GuRa98b]. Les applications concernées incluent notamment la segmentation de marché ou encore la segmentation démographique en identifiant par exemple des caractéristiques communes entre populations.

Les Séries Chronologiques
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 [AgFa93, RaFa94]. Cette technique permet par exemple d'identifier des sociétés qui présentent des séquences similaires d'expansion, ou encore de découvrir des stocks qui fluctuent de la même manière.

2 Problématiques abordées au cours du mémoire

Comme nous venons de le voir, il existe un grand nombre de techniques pour aider l'utilisateur à extraire à partir des données, des connaissances qui pourront s'avérer utiles pour l'aider dans ses prises de décisions. Cependant, notamment avec le développement de l'Internet, de nouveaux domaines d'applications émergent pour lesquels les techniques actuelles ne sont plus adaptées. C'est dans ce contexte que s'inscrit ce mémoire.

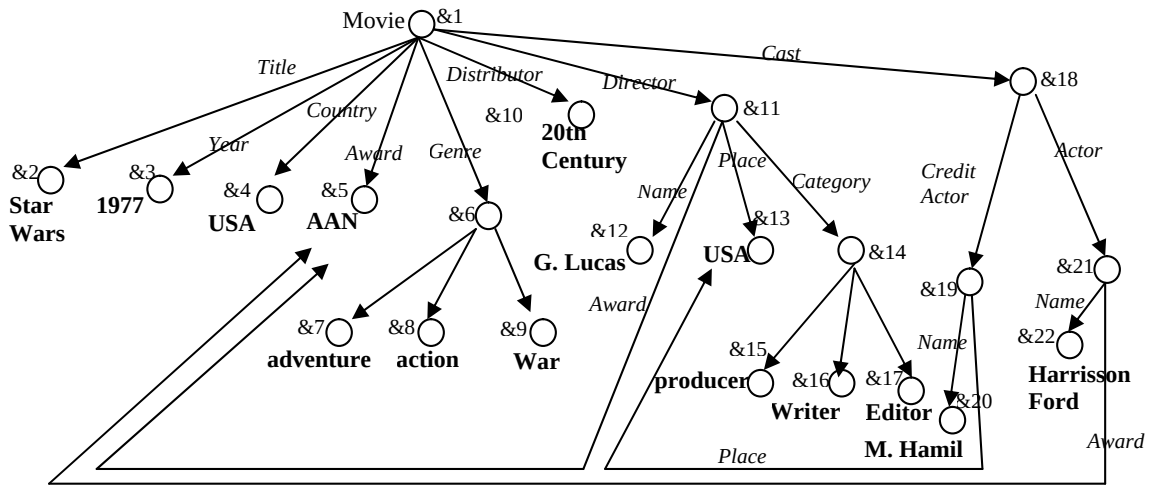


Figure 2 – Un exemple de document semi structuré (le film Star Wars)

De plus en plus de documents semi structurés, comme les fichiers HTML, Latex, Bibtex ou SGML sont désormais disponibles sur le Web. Ce type de données intervient généralement quand la source d'information n'impose pas une structure rigide ou lorsque les données sont issues de sources hétérogènes. L'exemple suivant illustre une application classique de données semi structurées.

Exemple 1 :

La Figure 2, extrait de la base de données des films imbd, illustre une partie d'un document semi structuré concernant le film Star Wars. Chaque cercle avec le numéro associé (&i) représente un sous document (e.g. un fichier HTML) et son identificateur (e.g. une URL). Les flèches et leur labels représentent des sous documents références avec leur rôle associé.

Contrairement aux données complètement non structurées comme les images ou les sons, les données semi structurées possèdent une structure pour savoir, par exemple, le type de questions qu'un utilisateur va pouvoir exprimer sans avoir à faire une analyse complète d'un document ou bien comment l'information est réellement représentée dans un document. Ainsi, la structure d'un document sur une personne permet de savoir que cette personne possède différents champs comme *Nom*, *Adresse*, *Famille* et que l'*Adresse* est un sous document qui possède lui-même des champs *Rue* et *Code Postal*. Le fait de connaître ce type d'information offre à l'utilisateur une aide pour décider si la source contient le type d'information qu'il recherche uniquement à partir des structures, de la même manière qu'il pourrait le faire pour un document au travers de sa table des matières.
Contrairement aux données structurées (comme celles des bases de données objet ou relationnel), les documents semi structurés n'ont pas de schémas ou de classes prédéfinis et chaque document contient donc sa propre structure. Dans le cas d'une base de données cinématographiques par exemple, certains films possèdent des acteurs qui eux-mêmes possèdent des oscars mais il existe des films pour lesquels aucun acteur ne dispose d'un oscar. Cette irrégularité structurelle n'implique cependant pas qu'il n'existe aucune similarité structurelle parmi les documents. Il est même fréquent de constater que des objets qui décrivent le même type d'information possèdent souvent des structures similaires. Par exemple, tous les films possèdent au moins un *titre*, une *année* et un *réalisateur*. Dans ce cadre, il est judicieux de penser qu'une requête sur la structure de ces documents peut devenir aussi importante qu'une requête sur les données [WaLi99].

La recherche de structures récurrentes ou typiques intéresse de nombreux domaines d'applications et concerne, par exemple, la bioinformatique, le Web sémantique ou le Web Usage Mining. La liste ci-dessous illustre quelques unes des applications possibles issues de la recherche de telles sous structures [AsAb02, MaHo99, MiSh01a, Work97, WaLi99, Zaki02] :

- **« carte » pour interroger/naviguer dans des données sources**
  L'une des limitations actuelles de l'interrogation ou de la navigation dans des documents semi structurés, comme ceux que l'on peut trouver sur le Web ou dans les bibliothèques digitales, est l'impression pour l'utilisateur d'être « perdu dans l'hyperespace » (*lost-in-hyperspace*) dans la mesure où il ne dispose pas d'information sur le schéma externe. Si l'utilisateur souhaite exprimer une requête, avec des langages comme WebSQL [MeMi96] par exemple, qui correspondent à des structures associées aux sources, il est indispensable de découvrir auparavant comment les informations sont représentées au sein de ces sources. Cette tâche peut être vue comme la découverte de structures typiques de documents.

- **Information générale sur le contenu**
  Très souvent, dans un premier temps, les utilisateurs ne cherchent pas quelque chose de spécifique mais préfèrent avoir une information générale sur le contenu des sources d'informations. Une bonne méthode pour aider l'utilisateur dans sa recherche est de lui proposer un aperçu de la structure des sources, de la même manière qu'il le ferait avec une table des matières. Ceci peut être réalisé en recherchant la structure de chaque document s'il y en a peu ou bien en recherchant les structures typiques dans le cas d'un grand nombre de documents. Bien entendu, la recherche de telles structures pouvant être assez fréquente, il devient alors indispensable de stocker les résultats dans une base de données qui pourra être interrogée par les utilisateurs. A partir des structures examinées, l'utilisateur pourra ainsi interroger ou naviguer dans les documents qui l'intéressent. La recherche de structures typiques peut aider à réaliser ce type de base de données.

- **Filtrer l'information**
  Dans le contexte de la recherche d'information, il est indispensable également de détecter les occurrences ou les régions de textes utiles. Pour cela, une approche classique consiste à donner au système deux jeux de données : un exemple de documents positifs et un exemple de documents négatifs. L'une des difficultés de tels systèmes est que pour être efficace, il est indispensable d'apprendre à partir de grands jeux de données. A l'heure actuelle, la plupart des exemples positifs sont recherchés à la main mais cela n'est plus possible pour une grande quantité de documents. La recherche de structures typiques offre une solution à ce type de problèmes en générant automatiquement de grands exemples d'apprentissage.

- **Classification de documents basée sur la structure**
  En considérant le point de vue semi structuré, chaque référence d'un document correspond à un sujet précis et les sujets d'un document peuvent eux mêmes être représentés par une structure d'arbre. Bien

entendu le sujet d'un sous document est alors relatif à celui de son « super document ». L'obtention de ces différentes catégories peut alors aider à classer les documents à partir des structures extraites.

- **Une aide à la construction d'index et de vues**
  Pour améliorer la recherche d'information, il est possible de construire des index ou des vues sur les structures fréquemment accédées. Par exemple, si nous savons que 98% des documents sur les films possèdent un champ *titre* et si nous savons que, la plupart du temps, les documents sont retrouvés en utilisant ce champ, alors indexer par rapport à *titre* (e.g. à l'aide d'un B-tree, d'une table de hachage ou d'une liste inversée) permet d'accélérer la recherche d'information.

- **Découvrir des motifs d'intérêt ou des accès réguliers**
  Découvrir le comportement et les goûts des utilisateurs sur le Web peut par exemple permettre de réorganiser dynamiquement un site. Nous pouvons considérer que les pages Web accédées au cours des différentes sessions correspondent à des structures typiques d'une collection de documents. Chaque document est enraciné sur une page d'entrée d'une session. En extrayant les structures typiques nous disposons d'une analyse plus fine du parcours des utilisateurs que celles que proposent les outils classiques d'analyse.

Au cours de ce mémoire, nous nous intéressons à la recherche de telles structures typiques. Ainsi, nous considérons, dans un contexte de fouille de données, le problème d'extraction suivant : étant donné une collection de documents, nous souhaitons rechercher les structures typiques qui interviennent dans un nombre minimal, spécifié par l'utilisateur, de documents. La recherche de structure typique peut être considérée également comme la recherche de sous arbres imbriqués fréquents si nous considérons que les documents manipulés peuvent être exprimés sous la forme d'arbre. Ainsi, au cours de ce mémoire nous utiliserons indifféremment ces deux interprétations.

L'idée générale, sous jacente au mémoire, est d'étudier si les approches basées sur la recherche de motifs séquentiels sont adaptables ou utilisables pour extraire des structures typiques dans un ensemble de données semi structurées.

Nous verrons, au cours de ce mémoire, qu'après avoir défini les types de données que nous recherchons, il est effectivement envisageable de réécrire nos graphes manipulés sous la forme de séquences. Cependant, étant donné les domaines d'applications visés, les contraintes imposées font, comme nous le montrerons, que les approches pour les motifs séquentiels ne sont pas adaptées. L'une des raisons principale de cette inaptitude est liée au fait que ces algorithmes sont définis pour manipuler des données plates et qu'il est évident que les structures issues des graphes sont complexes. Nous proposons ainsi une nouvelle approche appelée PSP_{tree} qui est inspirée des motifs séquentiels mais qui est étendue à la prise en compte de données complexes. De manière à étendre notre proposition à un plus grand nombre de domaines d'applications, nous généralisons notre problématique de manière à supprimer certaines contraintes liées principalement à la profondeur des composantes des structures. La proposition appelée PSP_{tree}–GENERALISE étend la précédente et offre à l'utilisateur de nouveaux types de structures typiques.

La seconde problématique est associée au fait que, surtout dans le cas du Web, les données sources ne sont pas statiques et qu'il est important de maintenir la connaissance extraite lors de l'étape préalable pour garantir que celle-ci est toujours représentative du contenu des bases associées. L'idée sous jacente est de montrer qu'il est réellement plus efficace de rechercher les connaissances en ayant effectué au préalable une extraction que de recommencer à zéro. Nous verrons au cours de ce mémoire qu'il est possible d'étendre l'algorithme PSP_{tree} de manière à lui faire conserver le minimum d'information indispensable (via la notion de bordure négative) pour optimiser l'extraction lors de mises à jour. Nous montrerons également comment tirer efficacement partie de cette information pour maintenir la connaissance extraite et proposerons pour résoudre les différentes opérations de mise à jour des algorithmes adaptés.

Les différents algorithmes proposés ont été intégrés dans un système nommé AUSMS (Automatic Update Schema Mining System). Ce système a été utilisé dans différents domaines d'applications : analyse du comportement d'utilisateurs sur des serveurs Web et aide à la sélection de vues dans le cas de données semi structurées.

3 Organisation du mémoire

Le mémoire est organisé de la manière suivante.

Le Chapitre II présente en détail les problématiques abordées au cours de ce mémoire ainsi que les travaux associés. Nous présentons ainsi successivement la problématique de la recherche de structures typiques dans de grandes bases de données et celle de la maintenance des connaissances extraites. De manière à pouvoir comparer notre approche, un panorama des techniques et approches existantes est proposé. Les contributions étant inspirées des algorithmes définis pour les règles d'association et les motifs séquentiels, nous présentons également les principaux travaux dans ces domaines. A l'issue de ce chapitre, une discussion est proposée au cours de laquelle nous revenons sur les points forts et faibles des approches et montrons les lacunes auxquelles nous souhaitons répondre.

Le Chapitre III concerne notre proposition à la problématique de la recherche de structures typiques. Celle-ci est tout d'abord basée sur une réécriture de la base de données sous la forme de séquences. Nous montrons que, même si nous disposons de séquences, les algorithmes de recherche de motifs séquentiels ne peuvent pas être utilisés pour répondre à notre problématique qui nécessite de nouvelles approches. Nous présentons deux nouvelles approches ainsi que les expériences menées. Ces dernières ont deux objectifs : valider nos propositions et étudier les performances des solutions mises en oeuvre. Pour valider l'approche, nous utilisons des données issues du Web afin de rechercher des structures typiques. Ce chapitre se conclue par une discussion au cours de laquelle nous réexaminerons l'utilisation des motifs séquentiels pour résoudre la problématique de la recherche de sous arbres.

Dans le Chapitre IV, nous décrivons notre contribution pour répondre à la problématique de la maintenance des connaissances extraites. L'approche proposée est basée sur l'utilisation de la bordure négative obtenue lors de l'extraction pour minimiser la mise à jour des connaissances lors de la modification des données sources. Au cours de ce chapitre, nous présentons quelques expériences menées afin de valider l'approche. Enfin, comme dans les chapitres précédents, une discussion vient clore le chapitre.

Le Chapitre V concerne le système AUSMS (Automatic Update Schema Mining System) dont le but est de proposer un environnement de découverte et d'extraction de connaissances pour des données semi structurées depuis la récupération des informations jusqu'à la mise à jour des connaissances extraites. Comme nous le verrons au cours de ce chapitre les principes généraux sont assez similaires au processus d'extraction de connaissances que nous avons présenté dans la section 1. Nous illustrons l'utilisation de notre système sur deux types d'applications. La première concerne le Web Usage Mining, i.e. l'analyse du comportement d'utilisateurs sur un ou plusieurs serveurs Web. Dans ce cadre, nous montrons que les propositions des chapitres précédents peuvent être utilisées pour offrir une analyse plus fine du comportement des utilisateurs en examinant les parcours contigus et en nous focalisant sur les différents points d'entrée sur les serveurs. Nous montrons également qu'il devient possible d'offrir au responsable d'un site une analyse des tendances dans le parcours de ses clients. Pour cela, nous présentons les problématiques du Web Usage Mining ainsi que les travaux associés et nous montrons les extensions apportées à AUSMS. Certaines expériences menées sur des sites différents (Laboratoire, Ecole, société de e-commerce) illustrent l'utilisation de notre système. Le second type d'application concerne l'aide à la sélection de vues dans des données semi structurées. Comme pour le Web Usage Mining, après un rappel de la problématique et des travaux existants, nous montrons les extensions et les expériences menées. Enfin, une discussion est proposée à la fin du chapitre.

Le Chapitre VI concerne la conclusion et les travaux futurs associés aux propositions. Après un rappel des différentes contributions, nous présentons les extensions possibles des travaux menés.

Chapitre II - Problématiques et travaux antérieurs

Nous avons vu dans le chapitre précédent que les algorithmes classiques n'étaient pas adaptés à la recherche de données non structurées sous forme « plate ». Dans cette section, nous présentons les problématiques que nous abordons et qui concernent d'une part la recherche de structures typiques dans des bases de données semi structurées et d'autre part la maintenance des connaissances extraites lors de l'étape précédente quand les sources sont modifiées. Dans le premier cas, il devient indispensable de modifier les algorithmes classiques dans la mesure où ils s'avèrent peu adaptés aux types de données que nous souhaitons manipuler. Pour cela, nous devons, préciser plus exactement le type de connaissances que nous souhaitons obtenir. En effet, étant donné qu'il existe de très nombreux modèles pour représenter les données semi structurées, il est important de bien préciser la manière dont nous souhaitons les traiter. Dans le second cas, comme nous l'avons aperçu dans le chapitre précédent, les données manipulées, et encore plus dans le cas du Web, évoluent constamment. Etant donné le temps nécessaire à une extraction de connaissances, il semble judicieux de tenir compte des connaissances préalablement obtenues, pour éviter lors de chaque mise à jour des données sources de recommencer le traitement global d'extraction.

Le chapitre est organisé de la manière suivante. Dans la section 1, nous présentons plus en détail les problématiques abordées au cours de ce mémoire. Ainsi, nous examinons successivement, la problématique de l'extraction de connaissances qui consiste dans notre cas à extraire les sous arbres qui apparaissent suffisamment fréquemment dans une base de données semi structurées, puis celle de la maintenance des connaissances extraites. La section 2 présente un aperçu des travaux associés aux problématiques abordées. Enfin, en conclusion de ce chapitre, nous proposons une discussion au cours de laquelle, nous présentons les limites des approches existantes pour répondre à notre problématique et la nécessité de définir une approche spécifique pour la manipulation de données semi structurées.

1 Présentation des problématiques étudiées

1.1 Extraction de structures typiques

Comme les modèles de bases de données semi structurées (XML [W3C03], OEM [AbQu97], …), nous adoptons la classe d'arbres ordonnés et étiquetés suivante.
Un *arbre* est un graphe connecté acyclique et une *forêt* est un graphe acyclique. Une forêt est donc une collection d'arbres où chaque arbre est un composant connecté de la forêt. Un *arbre enraciné* est un arbre dans lequel un nœud particulier est appelé la racine. Un *arbre étiqueté* est un arbre où chaque nœud est associé avec une étiquette. Un *arbre ordonné* est un arbre enraciné dans lequel les fils de chaque nœud sont ordonnés, i.e. si un nœud a k fils, nous pouvons les désigner comme premier fils, second fils et ainsi de suite jusqu'au $k^{\text{ième}}$ fils. Etant donné les types de jeux de données qui sont les plus classiques dans un contexte de fouille de données, nous considérons deux types d'ordre. Les fils d'un nœud de l'arbre sont soit ordonnés par ordre lexicographique par rapport à leur étiquette (« les fils d'un nœud sont alors vus comme un ensemble de fils »), i.e. l'ordre des fils n'est pas important pour l'application, soit ordonnés en fonction de l'application (« les fils d'un nœuds sont alors vus comme une liste de fils »). Nous considérons dans la suite que les arbres manipulés sont ordonnés, étiquetés et enracinés. En effet, ce type d'arbre s'applique à de nombreux domaines comme par exemple l'analyse des pages d'un site Web ou l'analyse de fichiers logs de transactions.

Dans un premier temps, nous précisons les « liens de parenté » des nœuds dans un arbre.

Définition 1 :

Soit x un nœud dans un arbre enraciné T de racine r. Chaque nœud sur le chemin unique de r à x est appelé un *ancêtre* de x et est noté par $y \leq_l x$, où l est la longueur du chemin d'y à x. Si y est un ancêtre de x alors x est un *descendant* de y. Chaque nœud est à la fois un ancêtre et un descendant de lui-même. Si $y \leq_1 x$, i.e. y est un ancêtre immédiat, alors y est appelé le *parent* de x. Nous disons que deux nœuds x et y sont *frères* s'ils ont le même parent.

Ainsi, nous pouvons définir plus formellement les arbres étiquetés, enracinés et ordonnés utilisés.

Définition 2 :

Soit T un arbre tel que $T=(N,B)$ où N représente l'ensemble des nœuds étiquetés et B l'ensemble des branches. La taille de T, notée $|T|$, correspond au nombre de nœuds dans T. Chaque nœud possède un numéro défini qui correspond à sa position lors d'un parcours en profondeur d'abord (parcours en pré-ordre) de l'arbre. Nous utilisons la notation n_i pour référencer le $i^{\text{ième}}$ nœud en fonction de la numérotation ($i=0...|T|-1$). En outre, il existe une fonction $I : N \rightarrow \{\otimes, \oplus, \perp\}$ qui affecte à chaque nœud $x \in N$ un indicateur sur l'ordre des fils. Si les fils sont ordonnés par l'application $I(x)= \otimes$; si les fils sont uniquement ordonnés par ordre lexicographique $I(x)= \oplus$ et si x ne contient pas de fils alors $I(x) = \perp$. Les étiquettes de chaque nœud sont issues de l'ensemble fini d'étiquettes $L=\{l, l_0, l_1, ..\}$, i.e. l'étiquette du nœud de nombre i est donnée par une fonction $l : N \rightarrow L$ qui fait correspondre à n_i l'étiquette associée $l(n_i) = y \in L$. Chaque nœud dans T est donc identifié par son numéro et son étiquette et possède un indicateur sur l'ordre de ses fils. Chaque branche, $b=(n_x,n_y) \in B$ est une paire ordonnée de nœuds où n_x est le parent de n_y.

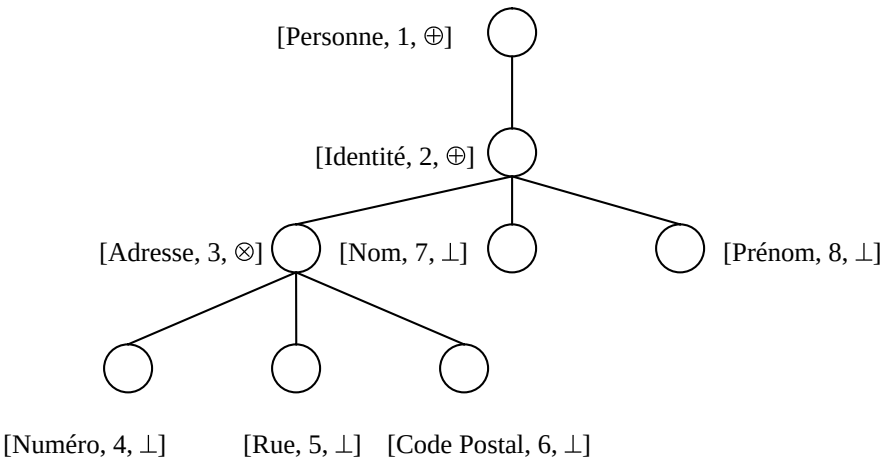


Figure 3 – Un exemple d'arbre ordonné, enraciné et étiqueté

Exemple 2 :

Considérons l'arbre représenté par la Figure 3. La racine de l'arbre possède le label Personne, il s'agit du premier nœud parcouru lors d'un parcours en profondeur d'abord (valeur 1), les fils du nœud (Identité) ne sont pas ordonnés donc l'indicateur vaut ⊕. Considérons le nœud de label Adresse. Etant donné que ses fils sont ordonnés par l'application, nous avons comme indicateur d'ordre ⊗. Enfin, considérons le dernier nœud de l'arbre en bas à droite. Son label est Numéro, son numéro lors du parcours est 4 et comme il s'agit d'une feuille de l'arbre, son indicateur d'ordre vaut ⊥. Le parent de ce nœud est Adresse et ses frères sont Rue et Code Postal.

Nous nous intéressons maintenant à la notion d'arbre imbriqué. Pour qu'un arbre soit imbriqué dans un autre arbre, il est nécessaire que tous les nœuds du sous arbre soit contenus dans les nœuds de l'arbre. Mais il faut également que les branches qui apparaissent dans le sous arbre possèdent deux sommets qui sont dans le même chemin de la racine à une feuille de l'arbre. Plus formellement, nous avons :

Définition 3 :

Soit $T = (N,B)$ un arbre étiqueté, ordonné et enraciné. Soit $S=(N_s,B_s)$ un arbre étiqueté, ordonné et enraciné. S est un *sous arbre imbriqué* de T, noté $S \preceq T$, si et seulement si :

- $N_s \subseteq N$
- $B = (n_x, n_y) \in B_s$ si et seulement si $n_y \preceq_1 n_x$, i.e. n_x est un parent dans T.

Si $S \preceq T$, nous disons également que T contient S. Un (sous) arbre de taille k est aussi appelé un k-(sous)arbre.

La définition précédente est bien entendu généralisable à la notion de sous forêt. Aussi par la suite nous utiliserons indifféremment la notion d'arbre ou de forêt.

Définition 4 :

Soit DB une base de données d'arbres, i.e. de forêts, et soit le sous arbre $S \preceq T$ pour chaque $T \in DB$. Soit $\{t_1, t_2, ..., t_n\}$ les nœuds dans T avec $|T|=n$ et soit $\{s_1, s2, ..., s_m\}$ les nœuds dans S avec $|S|=m$. S possède une *étiquette de correspondance* $\{t_{i1}, t_{i2}, ..., t_{im}\}$ si et seulement si :

- $l(s_k) = l(t_{ik})$ pour $k = 1, ..., m$
- la branche $b(s_j,s_k)$ est dans S si et seulement si t_{ij} est un parent de t_{ik} dans T.

Les conditions de la définition précisent que toutes les étiquettes des nœuds dans S ont une correspondance dans T et que la topologie des nœuds de correspondance dans T est la même que dans S. Une étiquette de correspondance est unique pour chaque occurrence de S dans T.

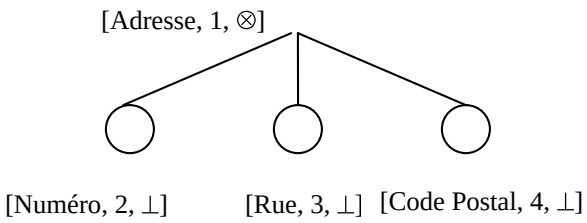


Figure 4 – Un exemple de sous arbre non imbriqué

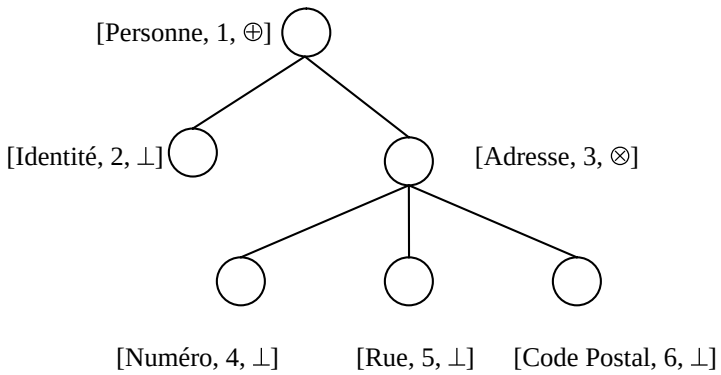


Figure 5 – Un exemple de sous arbre non imbriqué

Exemple 3 :

Considérons le sous arbre S de la Figure 6. Le sous arbre de cette figure est imbriqué dans l'arbre T de la Figure 3 car tous les nœuds de S sont inclus dans ceux de T, i.e. $N_s \subseteq N$ et les branches de l'arbre S respectent bien la topologie des nœuds de correspondance dans T. Par contre cette condition n'est pas vraie dans le cas des sous arbres S' et S'' de la Figure 4 et de la Figure 5. En effet, nous voyons que les nœuds ne sont pas à la

même hauteur dans l'arbre. Dans le cas de la Figure 4, nous pouvons constater que le nœud Adresse n'est pas au même niveau qu'Adresse dans la Figure 3. Dans le cas de la Figure 5, le lien de Parenté entre Adresse et Identité n'est pas respecté.

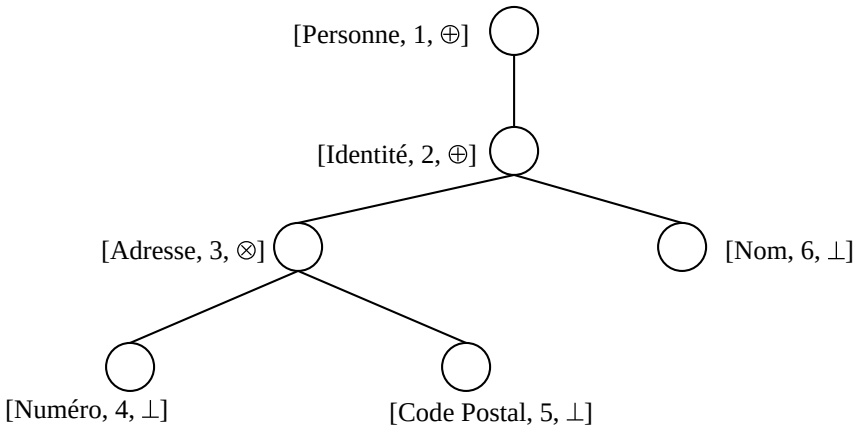


Figure 6 – Un exemple de sous arbre imbriqué

Considérons, à présent, comment le nombre d'occurrences des sous arbres dans la base de données *DB* est pris en compte.

Définition 5 :

Soit $\delta_T(S)$ le nombre d'occurrences du sous arbre *S* dans un arbre *T*. Soit $d_T(S) = 1$ si $\delta_T(S) > 0$ et $d_T(S) = 0$ si $\delta_T(S) = 0$. Le *support* d'un sous arbre *S* dans la base de donnée *DB* est défini par $support(S) = \Sigma_{T \in DB} d_T(S)$, i.e. le nombre d'arbres dans *DB* qui contiennent au moins une occurrence de *S*. Un sous arbre *S* est dit *fréquent* si son support est supérieur ou égal à une valeur de support minimal (*minSupp*) spécifiée par l'utilisateur. Nous notons L^k l'ensemble de tous les sous arbres fréquents de taille *k*.

Définition 6 :

Soit une base de données *DB*, la recherche des sous arbres fréquents ou de structures typiques, consiste à découvrir l'ensemble L^{DB} de tous les sous arbres fréquents de la base, i.e. des sous arbres dont le support est supérieur à *minSupp*.

De manière à faciliter l'expression des différents arbres, nous utilisons la notion de représentation d'arbre définie par :

Définition 7 :

Soit $T = (N, B)$ un arbre. Une *représentation d'arbre* est telle que :

- Si l'indicateur d'un nœud $x \in N$ vaut $\perp$ alors la représentation d'arbre vaut $x : \perp$.
- Soit un nœud $x \in N$ dont l'indicateur sur l'ordre des fils vaut $\oplus$ et soit $\{y_1, y_2, .., y_n\} \in N$ les fils du nœud *x*, alors la représentation d'arbre vaut $x : \{y_1, y_2, \ldots, y_n\}$.
- Soit un nœud $x \in N$ dont l'indicateur sur l'ordre des fils vaut $\otimes$ et soit $\{y_1, y_2, .., y_n\} \in N$ les fils du nœud *x*, alors la représentation d'arbre vaut $x : [y_1, y_2, \ldots, y_n]$.

Exemple 4 :

Considérons l'arbre T de la Figure 3. Sa représentation d'arbre associée est la suivante : Personne : {Identité : {Adresse : < Numéro : ⊥, Rue : ⊥, Code Postal : ⊥>, Nom : ⊥, Prénom : ⊥}.

Pour illustrer la problématique de recherche de sous arbres fréquents dans une base de données d'arbres, considérons l'exemple suivant.

Arbre_id	Arbre
T_1	Personne : {identite : {adresse : ⊥ , nom : ⊥}}
T_2	Personne : {identite : {adresse : <numero : ⊥, rue : ⊥, codepostal : ⊥>, compagnie : ⊥, directeur : <prenom : ⊥, nom : ⊥>, nom : ⊥}}
T_3	Personne : {identite : { adresse : <numero : ⊥, rue : ⊥, codepostal : ⊥>}, id : ⊥}
T_4	Personne : {identite : {adresse : ⊥, compagnie : ⊥, nom : ⊥}}
T_5	Personne : {identité : { adresse : ⊥, nom : ⊥ }}
T_6	Personne : {identite : { adresse : <numero : ⊥, rue : ⊥, codepostal : ⊥>, directeur : <nom : ⊥, prenom : ⊥>, nom : ⊥}}

Figure 7 – Une base de données exemple

Exemple 5 :

Supposons que la valeur du support minimal spécifiée par l'utilisateur soit de 50%, c'est-à-dire que pour être fréquent un sous arbre doit apparaître dans au moins trois arbres de la Figure 7. Les seuls sous arbres fréquents dans DB sont les suivants : Personne : {identite : {adresse : ⊥ , nom : ⊥ }} et Personne : {identite : {adresse : <numero : ⊥, rue : ⊥, codepostal : ⊥ >}}. Le premier est contenu dans T_1 mais également dans T_4 et T_5. Le second est contenu dans T_2, T_3 et T_6. Par contre, Personne : {identite :{ adresse :<numero : ⊥, rue : ⊥, codepostal : ⊥ >, directeur :<nom : ⊥, prenom : ⊥ >, nom : ⊥}} est contenu dans T_2 et T_6 mais n'est pas fréquent puisque le nombre d'occurrences de ce sous arbre est inférieur à la contrainte de support minimal.

1.2 Maintenance des structures fréquentes extraites

Dans cette section, nous considérons l'évolution des sources de données. Soit *DB* une base de données. Soit *db* la base de données incrément où de nouvelles informations sont ajoutées ou supprimées (*db* est décomposable en *db+* pour les ajouts et *db-* pour les suppressions). Soit *U*, la base de données mise à jour contenant tous les arbres des bases *DB* et *db*. Soit L^{DB}, l'ensemble de tous les sous arbres fréquents dans *DB*. Le problème de la maintenance des connaissances consiste à déterminer les sous arbres fréquents dans *U*, noté L^U, en tenant compte le plus possible des connaissances extraites précédemment de manière à éviter de relancer des algorithmes d'extraction sur la nouvelle base qui intègre les données mises à jour.

De manière à illustrer la problématique de la maintenance des structures typiques après modifications des données sources, considérons l'exemple suivant.

Exemple 6 :

Considérons l'ajout de la base db composée d'un seul nouvel arbre T_7 = {Personne : {identité : {adresse : ⊥ , nom : ⊥}}} à la base DB de la Figure 7 .Avec une même valeur de support minimal de 50%, un sous arbre, pour être fréquent, doit maintenant apparaître dans 4 transactions. Ainsi le sous arbre Personne : {identite : {adresse : ⊥ , nom : ⊥ }} reste fréquent car il apparaît dans T_1, T_4, T_5 et T_7 alors que Personne : {identite : {adresse : <numero : ⊥, rue : ⊥, codepostal : ⊥ >}} n'est plus fréquent car supporté uniquement par T_2, T_3 et T_6.

2 Aperçu des travaux antérieurs

Dans cette section, nous présentons les travaux antérieurs liés aux problématiques précédentes. Cependant, comme ils sont à l'origine de nombreuses approches et qu'ils sont proches de notre problématique, nous présentons dans un premier temps les travaux menés autour des motifs séquentiels. Nous nous intéressons, dans la section 2.2, aux algorithmes proposés pour rechercher des sous arbres fréquents. Enfin, nous présentons, au cours de la section 2.3, les différentes approches existantes pour maintenir la connaissance extraite par des algorithmes de fouille de données.

2.1 Motifs séquentiels

Avant de présenter les définitions, les propriétés et les principaux travaux existants dans le domaine des motifs séquentiels, nous effectuons d'abord un rappel de la problématique des règles d'association qui sont à l'origine des motifs séquentiels.

2.1.1 Rappel sur les règles d'associations

Formellement, le problème des règles d'association est présenté dans [AgIm93] de la façon suivante : soit $I = \{ i_1 i_2 \ldots i_k\}$ un ensemble d'items. Soit *DB* un ensemble de transactions, tel que chaque transaction *T*, dotée d'un identifiant noté (*TID*), soit consituée d'un ensemble d'items, appelés *itemset* vérifiant $T \subseteq D$. Une transaction *T* supporte *X*, un ensemble d'items de *I*, si elle contient tous les items de *X*, i.e. si $X \subseteq T$. Une règle d'association est une implication de la forme $X \rightarrow Y$, où $X \subseteq I$, $Y \subseteq I$ et $X \cap Y = \varnothing$. La confiance *c* dans la règle établit la probabilité qu'une transaction *T* supportant l'item *X*, supporte aussi l'item *Y*. Le support d'une règle $X \rightarrow Y$ correspond au pourcentage de toutes les transactions de *DB* supportant $X \cup Y$.

Ainsi à partir d'une base de transactions *DB*, le problème consiste à extraire toutes les règles d'association dont le support est supérieur à un support minimal spécifié par l'utilisateur (*minSupp*) et dont la confiance est supérieure à *minConf* (confiance spécifiée également par l'utilisateur). Cette tâche est généralement divisée en deux étapes :

1. Rechercher dans un premier temps tous les itemsets fréquents. Si la base possède *m* items, 2^m items sont alors potentiellement fréquents, d'où le besoin de méthodes efficaces pour limiter cette recherche exponentielle.

2. Générer à partir de ces itemsets fréquents, des règles dont la confiance est supérieure à *minConf*. Cette étape est relativement simple dans la mesure où il s'agit de conserver les règles du type $A/B \rightarrow B$ (avec $B \subset A$), ayant un taux de confiance suffisant.

Tous les algorithmes de recherche de règles d'association utilisent les propriétés suivantes de manière à optimiser la recherche de tous les ensembles fréquents.

Propriété 1 *(Support pour les sous-ensembles)*

Si $X \subseteq Y$ pour les itemsets *X* et *Y* alors $Support\ (X) \geq Support\ (Y)$ car toutes les transactions de *DB* qui supportent *X* supportent nécessairement *Y*.

Corollaire 1 *(Les sur-ensembles d'ensembles non fréquents ne sont pas fréquents)*

Si l'itemset *X* ne vérifie pas le support minimal dans *DB*, i.e. $Support\ (X) \leq minSupp$, alors tous les sur-ensembles *Y* de *A* ne seront pas fréquents car $Support\ (Y) \leq Support\ (X) \leq minSupp$ (C.f. Propriété 1).

Corollaire 2 *(Les sous-ensembles d'ensembles fréquents sont fréquents)*

Si l'itemset *Y* est fréquent dans *DB*, i.e $Support\ (Y) \geq minSupp$, alors tout sous-ensemble *X* de *Y* est fréquent dans *DB* car $Support\ (X) \geq Support\ (Y) \geq minSupp$ (C.f. Propriété 1). En particulier si $A = \{i_1, i_2, \ldots i_k\}$ est fréquent, tous les (*k-1*)-sous-ensembles sont fréquents. L'inverse n'est pas vrai.

Depuis ces dernières années de très nombreux travaux se sont intéressés à cette problématique et plus particulièrement à la génération des itemsets fréquents. La plupart des approches existantes sont basées sur l'approche *générer élaguer* définie dans l'algorithme Apriori [AgSr94].

Algorithm AlgoGenerique

Input : Un support minimal (*minSupp*) et une base de données *DB*
Output : L'ensemble L^{DB} des itemsets ayant une fréquence d'apparitions supérieure à minSupp.

1 : $k = 1$; $L^{DB} = \emptyset$// les itemsets fréquents
2 : C_1 = {{i}/ i ∈ ensemble des items de DB} ;
3 : **For each** $d \in DB$ **do** *CompterSupport* (C_1,*minSupp*,*d*) ; **enddo**
4: $L^1 = \{c \in C_1 / support(c) \geq minSupp\}$;
5 : **While** ($L^k \neq \emptyset$) **do**
6 : *genererCandidats* (C_k) ;
7 : **For each** $d \in DB$ **do** *CompterSupport* (C_k, *minSupp*, *d*) ; **enddo**
8 : $L^k = \{c \in C_k / support\ (c) \geq minSupp\}$;
9 : $L^{DB} \leftarrow L^{DB} \cup L^k$
10 : k += 1;
12 : **endWhile**
12 : **Return** L^{DB}

La méthode *générer élaguer* fonctionne de la manière suivante : puisque les fréquents ont entre eux des propriétés particulières, relatives à l'inclusion et puisque nous savons qu'il existe une taille maximale de fréquents (le fréquent de plus grande taille dans la base), il faut procéder par étapes (les étapes correspondant aux nombres de passes sur la base de données). Pour cela, l'algorithme détermine dans un premier temps les fréquents de taille 1 (les items fréquents qui apparaissent un nombre de fois supérieur au support minimal). A partir de ces fréquents des candidats de taille 2 (à partir des fréquents de taille 1) sont générés et les fréquents de taille 2 sont déterminés en effectuant une passe sur la base de données et en comparant leur nombre d'apparitions par rapport au support minimal. L'algorithme réitère la génération de candidats et la passe sur la base en étendant à chaque fois le nombre d'items contenu dans l'itemset. Il se termine lorsque plus aucun fréquent n'est trouvé.

Les travaux menés ces dernières années se sont d'abord intéressés à minimiser le nombre de passes sur la base. Nous pouvons citer, par exemple, l'algorithme Partition [SaOm95] qui n'effectue que deux passes sur la base pour déterminer les fréquents en divisant la base de données de manière à ce qu'elle tienne en mémoire. L'algorithme DIC [BrMo97] divise également la base mais effectue des recherches d'itemsets de différentes tailles pendant le parcours sur la base. L'approche de Toivonen [Toiv96] est basée sur une technique d'échantillonnage et de bordure négative (nous revenons sur cette notion dans la section 2.3).
De nouvelles approches plus récentes [HaPe00, HaKi01] considèrent que le goulot d'étranglement des processus d'extraction de type Apriori réside au niveau de la génération des candidats. Leurs études montrent, en effet, que pour 10^4 items fréquents découverts, l'algorithme Apriori doit tester 10^7 candidats de taille 2 sur la base de données. De plus pour découvrir des itemsets fréquents de taille 100, cet algorithme pourra tester (et donc générer) jusqu'à 10^{30} candidats au total. Les approches proposées consistent donc à contourner le problème de la génération des candidats en la supprimant grâce à une structure de FP-tree. Cette structure pour être mise en place, demande tout d'abord une passe sur la base de données afin de collecter les items fréquents. La suite de l'algorithme consiste alors à construire l'arbre qui est une transformation de la base d'origine limité aux items fréquents et à parcourir cet arbre pour rechercher les itemsets fréquents. Le lecteur intéressé par les différentes solutions proposées pour répondre à la problématique des règles d'association peut se reporter à [HaKa01]. Dans cet ouvrage, d'autres extensions pour les données fortement corélées sont proposées.

2.1.2 Définitions et propriétés des motifs séquentiels

Le problème de la recherche de séquences dans une base de données de transactions est présenté dans [AgSr95] de la façon suivante (nous gardons, au niveau des définitions, les concepts de clients et d'achats, cependant nous adapterons par la suite la recherche de séquences à d'autres types de données) :

Définition 8 :

Une *transaction*[1] constitue, pour un client *C*, l'ensemble des items achetés par *C* à une même date. Dans une base de données client, une transaction s'écrit sous la forme d'un ensemble : id-client, id-date, itemset. Un *itemset* est un ensemble d'items non vide noté ($i_1 i_2$ ... i_k) où i_k est un *item* (il s'agit de la représentation d'une

[1] Nous considérons ici le terme de transaction dans le sens d'une transaction financière et non pas celui d'une transaction dans une base de données.

transaction non datée). Une séquence est une liste ordonnée, non vide, d'itemsets notée $<s_1\ s_2\ ...\ s_n>$ où s_j est un itemset (une séquence est donc une suite de transactions qui apporte une relation d'ordre entre les transactions). Une séquence de données est une séquence représentant les achats d'un client. Soit T_1, T_2, ..., T_n les transactions d'un client, ordonnées par dates d'achat croissantes et soit *itemset* (T_i) l'ensemble des items correspondant à T_i, alors la séquence de données de ce client est $<itemset(T_1)\ itemset(T_2)\ ...\ itemset\ (T_n)>$.

Exemple 7 :

Soit C un client et S = <(3) (4 5) (8)>, la séquence de données représentant les achats de ce client. S peut être interprétée par « C a acheté l'item 3, puis en même temps les items 4 et 5, et enfin l'item 8 ».

Définition 9 :

Soit $s_1 = <a_1 a_2\ ...\ a_n>$ et $s_2 = <b_1 b_2\ ...\ b_m>$ deux séquences de données. s_1 est *incluse* dans s_2 (notée $s_1 \preceq s_2$) si et seulement si il existe $i_1 < i_2 < \ldots\ i_n$ des entiers tels que $a_1 \subseteq b_{i1}$, $a_2 \subseteq b_{i2}$, $\ldots\ a_n \subseteq b_{in}$.

Exemple 8 :

La séquence $s_1 = <(3)\ (4\ 5)\ (8)>$ est incluse dans la séquence $s_2 = <(7)\ (3\ 8)\ (9)\ (4\ 5\ 6)\ (8)>$ (i.e. $s_1 \preceq s_2$) car $(3) \subseteq (3\ 8)$, $(4\ 5) \subseteq (4\ 5\ 6)$ et $(8) \subseteq (8)$. En revanche $<(3)\ (5)>$ n'est pas incluse dans $<(3\ 5)>$ et vice versa.

Définition 10 :

Un client *supporte* une séquence *s* (fait partie du support pour *s*) si *s* est incluse dans la séquence de données de ce client. Le support d'une séquence *s* est calculé comme étant le pourcentage des clients qui supportent *s*. Soit *minSupp*, le support minimal spécifié par l'utilisateur. Une séquence qui vérifie le support minimal (i.e. dont le support minimal est supérieur à *minSupp*) est une *séquence fréquente*.

Remarque : Une séquence de données n'est prise en compte qu'une seule fois pour calculer le support d'une séquence fréquente, i.e. elle peut présenter plusieurs fois le même comportement, le processus de recherche de séquences considère qu'elle produit ce comportement sans tenir compte du nombre de ses apparitions dans la séquence de données.

Définition 11 :

Soit une base de données *DB*, l'ensemble L^{DB} des séquences fréquentes maximales (également notées motifs séquentiels) est constitué de toutes les séquences fréquentes telles que pour chaque *s* dans L^{DB}, *s* n'est incluse dans aucune autre séquence de L^{DB}.

Le problème de la recherche de séquences maximales (sequential patterns dans [AgSr95]) consiste donc à trouver l'ensemble L^{DB} de la définition précédente. Les deux propriétés suivantes considèrent le cas des sous-ensembles par rapport aux calculs du support et de l'inclusion.

Propriété 2 :

Soit s_1 et s_2, deux séquences, si s_1 est incluse dans s_2 (i.e. $s_1 \preceq s_2$) alors $support\ (s_1) \geq support\ (s_2)$.

Propriété 3 :

Soit s_1 une séquence non fréquente. Quelle que soit s_2 telle que $s_1 \preceq s_2$, s_2 est une séquence non fréquente.

La première propriété se justifie par le fait que toute séquence de données *d* dans *DB* supportant s_2 supporte obligatoirement s_1 (l'inverse ne se vérifie pas). La seconde propriété est une conséquence de la précédente. En effet, d'après celle-ci, $support\ (s_2) \leq support\ (s_1) < minSupp$, donc s_2 n'est pas fréquente.

Client	Date	Items
C_1	01/01/2003	20 60
C_1	02/02/2003	20
C_1	04/02/2003	30
C_1	18/02/2003	80 90
C_2	11/01/2003	10
C_2	12/01/2003	30
C_2	29/01/2003	40 60 70
C_3	05/01/2003	30 50 70
C_3	12/02/2003	10 20
C_4	06/02/2003	20 30
C_4	07/02/2003	40 70
C_4	08/02/2003	90

Figure 8 – Une base de données exemple

Pour illustrer la problématique de la recherche des motifs séquentiels, considérons l'exemple suivant. Soit la base de données *DB* illustrée par la Figure 8. Avec un support minimal de 50% (i.e. pour qu'une séquence soit retenue, il faut que deux clients dans la base de données supportent cette séquence), les séquences fréquentes sont alors les suivantes : <(20) (90)>, <(30) (90)> et <(30) (40 70). Les deux premières font partie du support pour C_1 et C_3, alors que la dernière apparaît dans les séquences de données des clients C_2 et C_4.

2.1.3 Les travaux autour des motifs séquentiels

Elaborés en premier lieu, les algorithmes de recherche de règles d'association connaissent de grandes difficultés d'adaptation aux problèmes d'extraction de motifs séquentiels. En effet, si le problème de la recherche de règles d'association est proche de celui des motifs séquentiels (il est à son origine), les études dans ce sens montrent que, lorsque l'adaptation est possible, c'est au prix de temps de réponses inacceptables (C.f. [AgSr95]). Plusieurs approches, destinées à présenter une solution nouvelle au problème de l'extraction de motifs ont été proposées depuis la définition du problème dans [AgSr95]. Le problème de l'extraction de motifs est proche de celui défini par [WaCh94,RiFl98], situé dans la découverte de similarités dans des bases de données de séquences génétiques pour lesquelles les séquences sont des caractères consécutifs séparés par un nombre variable de caractères parasites. Dans la définition du problème cependant, une séquence est considérée comme une liste ordonnée d'ensemble de caractères et non pas comme une liste de caractères. Si l'approche proposée par [AgSr95] se révèle très différente, c'est également en raison de la taille des instances traitées. En effet, les travaux de [WaCh94] sont des algorithmes basés sur des suffix tree qui travaillent principalement en mémoire centrale et révèlent une complexité en mémoire en O(*n*) avec *n* la somme des tailles de toutes les séquences de la base. Il est impossible, dans le cadre des motifs séquentiels d'accepter une complexité en mémoire qui dépend de la taille de la base car nous travaillons sur des instances de très grande taille et la mémoire volatile n'est jamais aussi importante que la mémoire permanente.

Pour répondre à la problématique initiale de nombreux travaux ont été réalisés dont l'un des précurseurs, l'algorithme GSP [SrAg96], a été développé dans le cadre du projet Quest d'IBM. Nous présentons dans la suite de cette section l'algorithme GSP ainsi que les principes utilisés pour générer et stocker efficacement les candidats. Associés à la recherche de motifs, de nombreux travaux ont été réalisés ces dernières années pour prendre en compte diverses contraintes. Le lecteur intéressé par ces travaux peut, par exemple, se reporter à : [GaRa02] et [WaHa04].

L'approche pionnière : l'algorithme GSP

GSP (Generalized Sequential Patterns) [SrAg96] est un algorithme basé sur la méthode *générer élaguer* mise en place depuis Apriori et destiné à effectuer un nombre de passes raisonnable sur la base de données. Cependant, les éléments manipulés n'étant plus des itemsets mais des séquences, la génération des candidats dans GSP est réalisée de la façon suivante : soit *k* la longueur des candidats à générer, plusieurs cas peuvent se présenter.

- *k=1*. Pour générer les fréquents de taille 1, GSP énumère tous les items de la base et détermine, en une passe, ceux qui ont une fréquence supérieure au support.

- *k=2*. A partir des items fréquents, GSP génère les candidats de taille 2 de la façon suivante : pour tout couple *x,y* dans l'ensemble des 1-séquences fréquentes (les items fréquents), alors si *x=y*, le 2-candidat *<(x) (y)>* est généré et si $x \neq y$, alors les 2-candidats *<(x) (y)>* et *<(x y)>* sont générés.
- $k > 2$. C_k est obtenu par autojointure sur L_{k-1}. La relation jointure (s_1, s_2) s'établit si la sous séquence obtenue en supprimant le premier élément de s_1 est la même que la sous séquence obtenue en supprimant le dernier élément de s_2. La séquence candidate obtenue par jointure (s_1, s_2) est la séquence s_1 étendue avec le dernier item de s_2. L'item ajouté fait partie du dernier itemset s'il était dans un itemset de taille supérieure à 1 dans s_2 et devient un nouvel itemset s'il était dans un itemset de taille 1 dans s_2.

<(1 2) (3)>	<(1 2) (3) >
< (2) (3 4)>	<(2) (3) (5)>
<(1 2) (3 4)>	<(1 2) (3) (5)>

Figure 9 – Deux exemples de jointure entre candidats dans GSP

Items	2-candidats	
<(1)>	<(1 2)>	<(1)(1)>
<(2)>	<(1 3)>	<(1)(2)>
<(3)>	<(1 4)>	<(1)(3)>
<(4)>		<(1)(4)>
	<(2 1)>	<(2)(1)>
	<(2 3)>	<(2)(2)>
	<(2 4)>	<(2)(3)>
		<(2)(4)>
	<(3 1)>	<(3)(1)>
	<(3 2)>	<(3)(2)>
	<(3 4)>	<(3)(3)>
		<(3)(4)>
	<(4 1)>	<(4)(1)>
	<(4 2)>	<(4)(2)>
	<(4 3)>	<(4)(3)>
		<(4)(4)>

3-fréquents	4-candidats
<(1 2) (3)>	<(1 2) (3 4)>
<(1 2) (4)>	<(1 2) (3) (5)>
<(1) (3 4)>	
<(1 3) (5)>	
<(2) (3 4)>	
<(2) (3) (5)>	

Figure 10 – Génération de candidats dans la méthode *générer élaguer*

Exemple 9 :

La Figure 10 (tableau de gauche) illustre la génération des candidats de taille 2 à partir des fréquents de taille 1. La Figure 9 illustre la façon dont la jointure est effectuée entre des séquences candidates. La Figure 10 (tableau de droite) illustre la génération des 4-candidats (colonne de droite) obtenus à partir d'un ensemble de 3-fréquents (colonne de gauche). Nous remarquons que la jointure (<(1 2) (3)>, <(2) (3 4)>) produit la séquence candidate de taille 4 <(1 2) (3 4)> et que jointure (<(1 2) (3)>, <(2) (3) (5))>) produit le candidat <(1 2) (3) (5)>. Les autres séquences ne peuvent pas apparaître dans la jointure. La séquence <(1 2) (4)> ne peut pas faire partie de la relation jointure car il n'y a aucune séquence de la forme <(2) (4 x)>.

Pour évaluer le support des candidats en fonction d'une séquence de données, GSP utilise une structure d'arbres de hachage destinée à effectuer un tri sommaire des candidats.

- Les candidats sont stockés en fonction de leur préfixe. Pour ajouter un candidat dans l'arbre des séquences candidates, GSP parcourt ce candidat et effectue la descente correspondante dans l'arbre. En

arrivant sur une feuille, GSP ajoute ce candidat à la feuille et si la feuille dépasse la taille maximale alors elle est scindée en deux nouvelles feuilles dans lesquelles les candidats sont répartis.

- Pour trouver quelles séquences candidates sont incluses dans une séquence de données, GSP parcourt l'arbre en appliquant une fonction de hachage sur chaque item de la séquence de données. Quand une feuille est atteinte, elle contient des candidats potentiels pour la séquence de données. Cet ensemble de séquences candidates est constitué de candidats inclus dans la séquence de données et de candidats « parasites ». Un algorithme supplémentaire est alors utilisé pour analyser ces séquences candidates et déterminer celles qui sont réellement incluses dans la séquence.

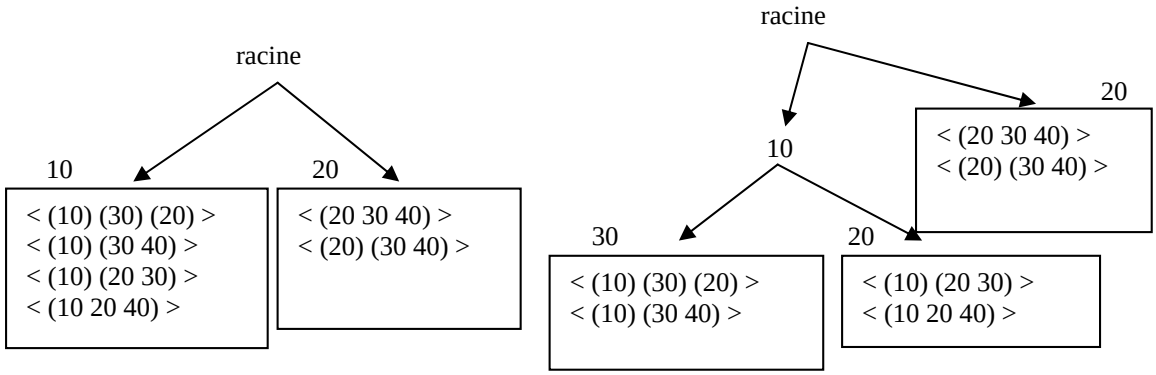


Figure 11 – La structure hash-tree de GSP

Exemple 10 :

La Figure 11 illustre la façon dont GSP gère la structure de hach-tree. Les deux arbres servent à conserver les mêmes candidats mais les feuilles de l'arbre de gauche peuvent contenir plus de deux séquences candidates alors que dans l'arbre de droite seulement deux séquences candidates peuvent être stockées dans la même feuille. Nous remarquons que la feuille qui porte l'étiquette 20 dans l'arbre de gauche (le deuxième fils de la racine) est utilisée pour stocker les séquences candidates < (20 30 40)>et <(20) (30 40)>. Quand GSP atteint cette feuille, il n'a aucun moyen de savoir si c'est la séquence < (20 30 40)> ou bien la séquence <(20) (30 40)> qui l'a conduit jusqu'à cette feuille. C'est pour cette raison que GSP doit tester les séquences candidates contenues dans les feuilles atteintes afin de savoir quels supports incrémenter.

Les approches proposant une nouvelle représentation des données : PSP, SPAM et SPADE

L'approche GSP, outre le fait qu'elle ait posé la problématique a eu l'avantage de prouver qu'il était nécessaire d'avoir une structure efficace pour représenter les candidats. En effet, étant donné le nombre de candidats générés (il est beaucoup plus important que dans le cas des règles d'association car il faut prendre en compte le fait qu'il existe plusieurs itemsets différents dans les séquences), il devient indispensable d'avoir une structure permettant de retrouver rapidement les candidats inclus dans une séquence. Un certain nombre de travaux ont été réalisés après GSP pour améliorer la représentation des données. L'un des précurseurs, PSP propose une nouvelle structure d'arbre préfixé pour stocker les candidats et les séquences. Ultérieurement, SPADE a proposé d'utiliser une structure d'arbre préfixé mais associée à une vision verticale de la base de données. Enfin, plus récemment, une nouvelle approche SPAM a été proposée et utilise une représentation sous la forme de bitmap pour manipuler les candidats. Dans la suite de cette section nous décrivons successivement les approches PSP, SPADE et SPAM.

L'algorithme PSP

Dans [MaPo99a, Mass02], les auteurs proposent un nouvel algorithme, appelé PSP, qui est basé sur l'approche *générer élaguer* mais qui utilise une nouvelle organisation des séquences candidates pour trouver plus facilement l'ensemble des candidats inclus dans une séquence de données. En effet, la structure de hachage utilisée dans GSP consiste à mettre en commun des préfixes sans faire de distinction entre deux items du même itemset et deux items avec changement d'itemsets. De ce fait, l'algorithme se voit contraint, lorsqu'il extrait des séquences candidates des feuilles, de tester à nouveau ces séquences pour rechercher celles qui sont incluses dans la séquence de données. La structure de données dans PSP est une structure d'arbre préfixé qui a l'avantage de factoriser les séquences candidates selon leur préfixe et tient compte des éventuels changement d'itemsets. La différence principale par rapport à la structure utilisée dans [SrAg96], vient du fait que chaque chemin de la

racine vers une feuille de l'arbre représente une séquence complète et une seule. Ainsi, pour des séquences candidates de longueur *k* (i.e. les *k*-candidats), la profondeur de l'arbre est exactement égale à *k*. D'autres caractéristiques de cette structure contribuent à la distinguer de l'arbre de hachage utilisé par [SrAg96] et plus particulièrement le fait que les candidats et les séquences fréquentes de longueur (0, …, *k*) sont gérés par une structure unique.

La structure préfixée est gérée de la manière suivante. Au premier niveau (k=1), chaque branche issue de la racine relie celle-ci à une feuille qui représente un item. Chacune de ces feuilles contient l'item et son support (le nombre de ses apparitions dans la base). Ce support est calculé par un algorithme comptant le nombre d'occurrences de l'item dans la base et dans des séquences distinctes.

Figure 12 – Phase de pruning sur les 1-itemsets

Exemple 11 :

L'arbre représenté à gauche de la Figure 12 illustre l'état de la structure après l'évaluation du support pour chaque item. Considérons que pour qu'une séquence soit retenue elle doit apparaître dans au moins deux séquences de données. L'arbre de droite de cette même figure représente la structure contenant uniquement les items fréquents. Considérons la feuille contenant le sommet 50 dans l'arbre de gauche, son support est de 1 séquence. Cet item ayant un support inférieur aux support minimum spécifié par l'utilisateur (2 séquences), il est éliminé des séquences candidates lors de la phase d'élagage.

Pour les niveaux suivants (k>1), chaque nœud de l'arbre vers une feuille représente une séquence. Pour distinguer les itemsets à l'intérieur d'une séquence (ex. (10 20) et (10) (20)) les fils d'un nœud sont séparés en deux catégories : « Same Transaction » et « Other Transaction ».
En outre, chaque feuille possède un identifiant de la dernière transaction ayant participé à l'incrémentation de son support (*idLast*). Il se peut, en effet, que l'algorithme de vérification des candidats passe plusieurs fois par une même feuille quand le chemin menant de la racine vers cette feuille est inclus plusieurs fois dans la séquence de données considérée. Dans un tel cas, la séquence candidate *n* fois incluse ne peut être considérée comme n fois présente dans la base uniquement à cause de cette séquence de données. L'algorithme de vérification des candidats peut ainsi savoir si la séquence de données qu'il teste a déjà participé à l'incrémentation du support de la feuille considérée.

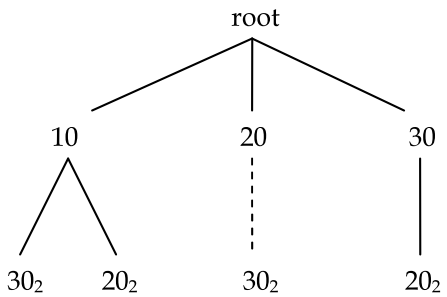


Figure 13 – Séquences fréquentes de taille 1 et 2

Exemple 12 :

L'arbre da la Figure 13 représente les séquences fréquentes de taille 1 et 2. Une branche en traits pleins marque le début d'un nouvel itemset dans la séquence (dans la séquence < (10) (30) >, il y a une branche en trait plein entre les items 10 et 30) et une branche en traits pointillés entre deux items signifie que ces items font partie de

la même transaction (< (20 30) >). Les séquences fréquentes représeentées sont donc : {< (10) (30) >, < (10) (20) >, < (20) (30) >, < (30) (20) >}.

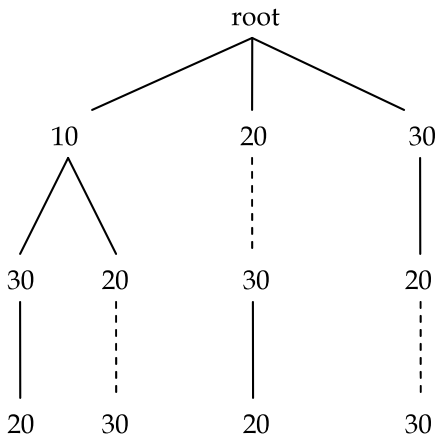


Figure 14 – Les 3-candidats obtenus

Exemple 13 :

L'arbre représenté par la Figure 14 illustre la façon dont les k-candidats et les l-sequences fréquentes (avec l compris entre 1 et k-1) sont simultament gérées par la structure. Cet arbre est obtenu après la génération des candidats de taille 3 à partir de l'arbre représenté dans la Figure 13. Les séquences fréquentes obtenues à partir de cet exemple sont < (10) (30) (20) >, < (10) (20 30) > et < (30) (20 30) >.

La génération des candidats de niveau 1 et 2 est similaire à celle de GSP. Par contre pour les niveaux supérieurs (*k>2*), l'algorithme tire parti de la structure construite de la manière suivante. Pour chacune des feuilles *l* de l'arbre, l'algorithme recherche à la racine l'item *x* représenté par *l*. Ensuite il étend la feuille *l* en construisant pour cette feuille une copie des fils de *x*. A cette étape de la génération, PSP applique un filtrage destiné à ne pas générer de séquences dont nous savons à l'avance qu'elles ne sont pas fréquentes. Pour cela, il considère *F*, l'ensemble des fils de *x*. Pour chaque *f* dans *F*, si *f* n'est pas frère de *l* alors il est inutile d'étendre *l* avec *f*. En effet, nous savons que si *f* n'est pas frère de *l*, alors soit *p* le père de *l*, (*p,f*) n'est pas fréquent et donc (*p,l,f*) non plus.

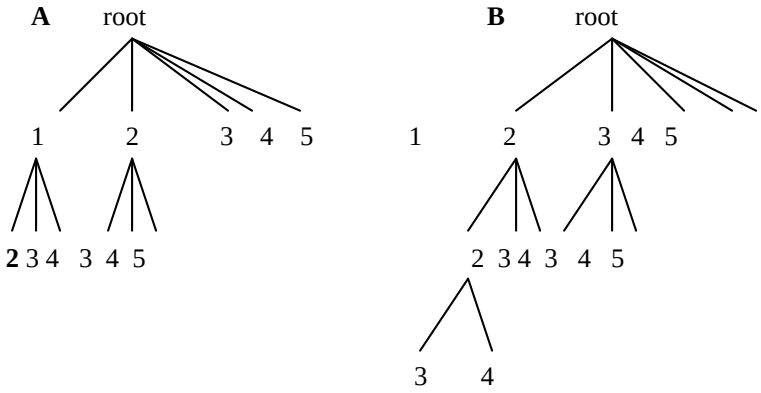


Figure 15 – Un candidat non fréquent détecté à l'avance

Exemple 14 :

La Figure 15 représente un arbre avant (arbre A) et après (arbre B) la génération des candidats de taille 3. La feuille représentant l'item 2 (en gras dans l'arbre A) est étendue (dans l'arbre B) uniquement avec les items 3 et 4 (pourtant 5 est un fils de 2 à la racine). En effet, 5 n'est pas frère de 2 (en gras dans l'arbre A), ce qui signifie que <(1) (5)> n'est pas une séquence fréquente. Ainsi, <(1) (2) (5)> ne peut être déterminée fréquente et il est donc inutile de générer ce candidat.

L'algorithme SPADE

SPADE, présenté dans [Zaki00, Zaki01], se classe dans la catégorie des approches qui cherchent à réduire l'espace des solutions en regroupant les motifs séquentiels par catégorie. Pour SPADE, comme pour PSP, les motifs fréquents présentent des préfixes communs, qui permettent de décomposer le problème en sous problèmes qui seront traités en mémoire. Le calcul des fréquents de taille 2, i.e. L^2, passe par une inversion de la base qui la transforme d'un format vertical vers un format horizontal. L'auteur considère que cette opération peut être simplifiée si la base peut être chargée en mémoire vive. SPADE gère les candidats et les séquences fréquentes à l'aide de classes d'équivalence comme suit : deux *k*-séquences appartiennent à la même classe si elles présentent un préfixe commun de taille (*k-1*). Plus formellement, soit $P_{k-1}(\alpha)$ la séquence de taille *k-1* qui préfixe la séquence α. Comme α est fréquente, i.e. $P_{k-1}(\alpha) \in L^{k-1}$ avec L^{k-1} les fréquents de taille *k-1*. Une classe d'équivalence est définie de la façon suivante :

$$[s \in L^{k-1} a] = \{ \alpha \in L^k \mid P_{k-1}(\alpha) = s\}$$

Chacune des clases d'équivalence contient alors deux types d'éléments : $[s.l_1] < (x)>$ ou bien $[s.l_2]=<x>$, selon que l'item *x* appartient ou pas à la même transaction que le dernier item de *s*.

Les candidats sont ensuite générés selon trois critères :

- Autojointure ($[s.l_1]$ X $[s.l_1]$)
- Autojointure ($[s.l_2]$ X $[s.l_2]$)
- Jointure ($[s.l_1]$ X $[s.l_2]$)

Le reste de l'algorithme concernant le comptage du support pour les candidats générés, repose sur la réécriture préalable de la base de données. En effet, la transformation consiste à associer à chaque *k*-séquence l'ensemble des couples (client, itemset) qui lui correspondent dans la base. L'exemple suivant illustre le résultat de cette transformation, et la façon dont la table obtenue est utilisée lors du calcul du support.

Client	Itemset	Items
C_1	10	A B
C_1	20	B
C_1	30	A B
C_2	20	A C
C_2	30	AB C
C_2	50	B
C_3	10	A
C_3	30	B
C_3	40	A
C_4	30	A B
C_4	40	A
C_4	50	B

Figure 16 – Une base de données exemple pour SPADE

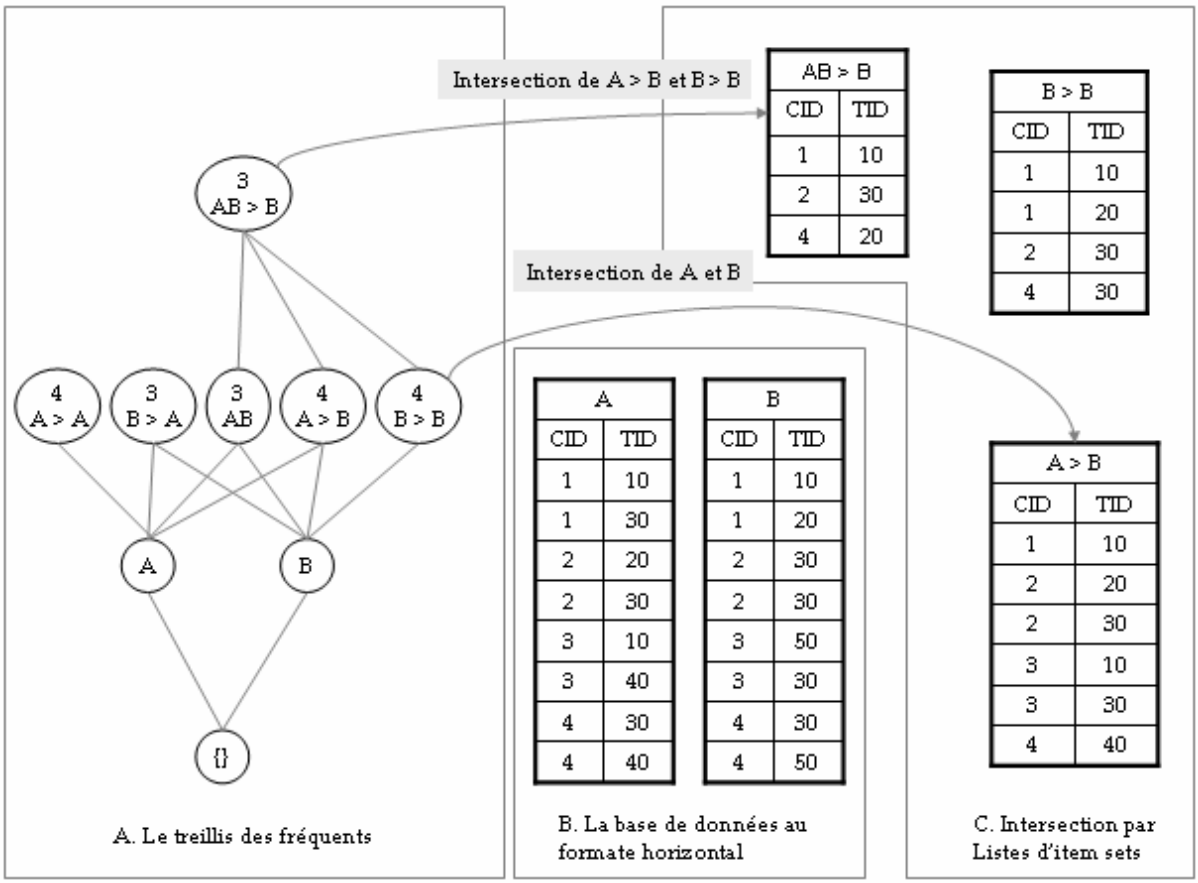


Figure 17 – Intersections de listes d'itemsets dans SPADE, avec la base de données de la Figure 16

Exemple 15 :

La Figure 16 représente une base de données selon le format vertical classique. Après transformation selon les besoins de l'algorithme SPADE, la base de données DBspade est alors décrite dans le cadre « B » de la Figure 17. Une fois la base de données ainsi transformée, l'algorithme peut alors accéder aux supports des candidats de taille 2, grâce aux listes d'itemsets et clients supportant les items fréquents, en procédant à une intersection de ces listes. Considérons l'enchaînement « A>B » qui signifie « A est suivi par B » ou encore < (A) (B)>. En utilisant un algorithme d'intersection des listes de A et de B, SPADE peut déduire la liste d'itemsets de <(A) (B)>.

La façon dont SPADE gère les intersections est détaillée dans [Zaki01].

L'algorithme SPAM

Dans [ArGe02], les auteurs proposent SPAM (Sequential PAttern Mining) pour rechercher des motifs séquentiels.

SPAM considère l'existence d'un ordre lexicographique (noté $\preceq$) sur les items de la base de données. Un arbre de séquence respectant cette notion d'ordre lexicographique est donc construit. Ainsi, tous les fils d'un nœud *n* dans l'arbre sont des nœuds *n'* tel que $n \preceq n'$ et quelque soit le nœud *m* de l'arbre si $n' \preceq m$ alors $n \preceq m$.
La branche qui relie *m* et *n* est telle que la séquence de *n* est une extension de la séquence de *m*. Cette extension est obtenue de deux manières : soit par *S-extension* soit par *I- extension*. Une *S-extension* est obtenue en ajoutant une nouvelle transaction constituée d'un seul élément à la séquence de son père (elle correspond à « *other transaction* » dans PSP). Une *I-extension* est obtenue en ajoutant un item au dernier ensemble d'items de la séquence de son père, tel que cet item soit plus grand que n'importe lequel des items de ce dernier ensemble (elle correspond à « *same transaction* » dans PSP).

Customer ID (CID)	TID	Itemset
1	1	{a, b, d}
1	3	{b, c, d}
1	6	{b, c, d}
2	2	{b}
2	4	{a, b, c}
3	5	{a, b}
3	7	{b, c, d}

CID	Séquence
1	({a, b, d}, {b, c, d}, {b, c, d})
2	({b}, {a, b, c})
3	({a, b}, {b, c, d})

Figure 18 – Une base de données sous la forme CID/TID et son équivalence sous la forme de séquences

Exemple 16 :

Considérons la séquence s_a = ({a, b, c}, {a, b}) alors ({a, b, c}, {a, b}, {a}) est une S-extension de s_a et ({a, b, c}, {a, b, d}) est une I-extension de s_a.

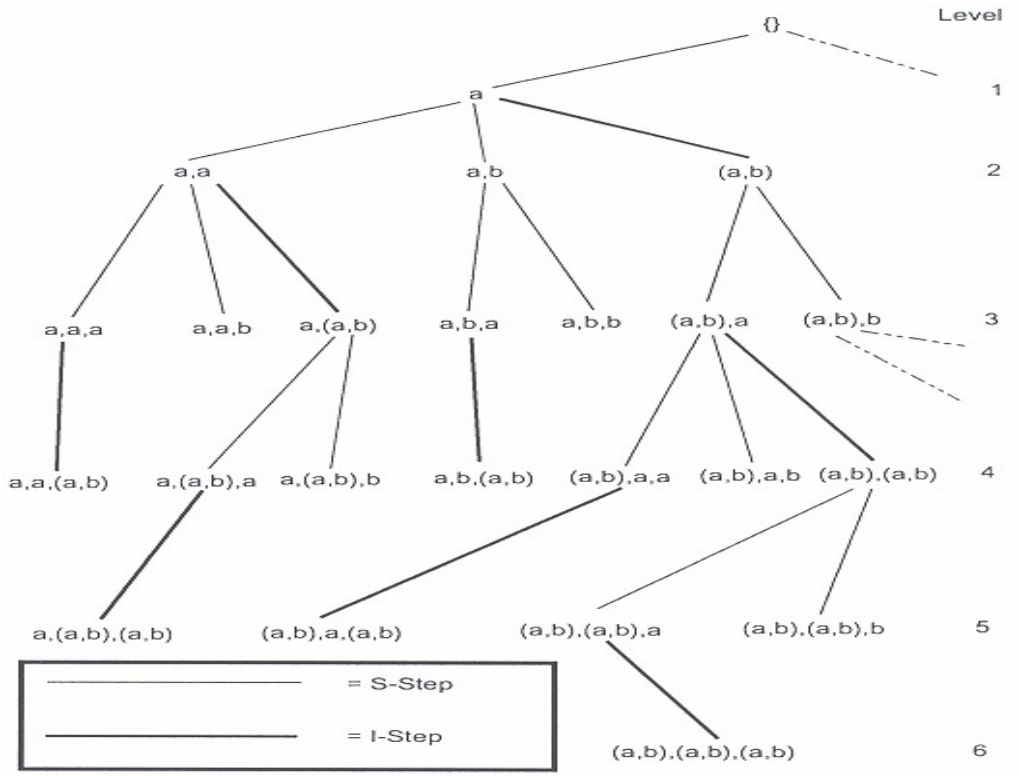


Figure 19 - L'arbre lexicographique des séquences

Chaque nœud de l'arbre peut générer des fils dont la séquence est une extension de type *S* ou *I* de la séquence de son père. Le processus de génération de séquences de type *S* est appelé le *S-step* et celui de type *I*, le *I-Step*. Il est ainsi possible d'associer à chaque nœud de l'arbre *n* deux ensembles : S_n, l'ensemble des items candidats pour le *S-step*, et I_n, l'ensemble des candidats pour le *I-Step*.

Exemple 17 :

La Figure 19 illustre un exemple d'arbre de séquences complet pour deux items a et b avec une taille de séquence maximale 3. La racine de l'arbre représente la séquence vide et chaque niveau inférieur à k contient toutes les k-séquences ordonnées dans l'ordre lexicographique. Les S-séquences précédent les I-séquences.

SPAM parcourt l'arbre des séquences en profondeur pour effectuer l'extraction des motifs. Pour chaque nœud *n* traversé, le support des nœuds générés dans le *S-step* et *le I-step* (par extension de la séquence de *n*) est compté. Si le support d'une séquence générée *s* est plus grand que le support minimal *minSupp*, la séquence est stockée et les étapes précédentes sont appliquées de manière récursive sur le nœud contenant *s*. Si aucune des séquences générées n'est fréquente alors l'algorithme s'arrête.

De manière à améliorer la recherche de séquences candidates incluses dans une séquence, les auteurs proposent une structure de données basée sur des vecteurs de bits. Un vecteur de bits vertical est créé pour chaque item de la base de données, et chaque vecteur de bits contient un bit correspondant à chaque transaction de la base de données. Si un item *i* apparaît dans une transaction *j* alors le bit correspondant du vecteur de bits est mis à *1*, autrement il a pour valeur *0*.

CID	TID	{a}	{b}	{c}	{d}
1	1	1	1	0	1
1	3	0	1	1	1
1	6	0	1	1	1
-	-	0	0	0	0
2	2	0	1	0	0
2	4	1	1	1	0
-	-	0	0	0	0
-	-	0	0	0	0
3	5	1	1	0	0
3	7	0	1	1	1
-	-	0	0	0	0
-	-	0	0	0	0

Figure 20 – Représentation de la base sous forme de vecteur de bits vertical

La Figure 20 représente la base de données de la Figure 18 sous la forme d'un vecteur de bits vertical. Considérons un vecteur de bits pour l'item *i* (noté *B (i)*) et un vecteur de bits pour l'item *j* (noté *B (j)*) alors le vecteur de bits pour *{i, j}* est obtenu via l'opérateur logique *AND* sur les deux bitmaps précédents : *B (i) & B (j)*.

({a}, {b})		{d}		({a}, {b, d})
0		1		0
1		1		1
1		1		1
0		0		0
0		0	résultat	0
0	&	0	→	0
0		0		0
0		0		0
0		0		0
1		1		1
0		0		0
0		0		0

Figure 21 – Un exemple de I-extension de *({a}, {b})* par *{d}*

Exemple 18 :

La Figure 21 représente un exemple de I-extension de la séquence ({a}, {b}) par la séquence {d}.

Soit un vecteur de bits image d'une séquence *B* (s_a) et un vecteur de bits image de l'item *i B (i)*. La *S-extension* de s_a par *i* est réalisée de la manière suivante. Un vecteur de bits est tout d'abord généré à partir de *B* (s_a). Les bits situés avant *k* pour chaque *CID* différents ont pour valeur 0 et les bits situés après *k* ont pour valeur 1 (*k* est le premier bit qui a pour valeur 1 dans *B* (s_a) et possède donc une valeur différente pour chaque *CID*). *k* représente en fait la première date à laquelle la séquence s_a est supportée dans chacune des différentes séquences associées aux *CID*.

Exemple 19 :

Considérons la Figure 21, les valeurs de k pour s_a *= ({a}) sont respectivement : 1,2 et 1.*

La *S-extension* est alors réalisée via l'opérateur logique AND entre ce vecteur de bits transformé *B'* (s_a) et *B (i)*. La Figure 22 illustre un exemple de *S-extension*.

({a})		({a}) transformé		{b}		({a}, {b})
1		0		1		0
0		1		1		1
0		1		1		1
0		1		0		0
0	S-step	0		1	résultat	0
1	→	0	&	1	→	0
0		1		0		0
0		1		0		0
1		0		1		0
0		1		1		1
0		1		0		0
0		1		0		0

Figure 22 – Un exemple de S-extension de *({a})* par *{b}*

Un parcours sur la base a permis lors de la construction du vecteur de bits de connaître le nombre de *TID* pour chaque *CID* donnée. Il est alors possible de partitionner le vecteur de bits, comme illustré Figure 18, en section où chaque section représente les *CID* d'une *TID* donnée. Le calcul du support pour une séquence donnée consiste alors à compter le nombre de sections non nulles dans la représentation par vecteur de bits verticale de cette séquence.

Sans génération de candidats : L'algorithme PrefixSpan

La méthode PrefixSpan, présentée par [Mort99, PeHa01] se base sur une étude attentive du nombre de candidats qu'un algorithme de recherche de motifs séquentiels peut avoir à produire afin de déterminer les séquences fréquentes. En effet, selon les auteurs, pour envisager d'utiliser un algorithme comme GSP, PSP ou SPADE, il faut s'attendre à devoir générer, uniquement pour la seconde passe, pas moins de $n^2+(nx(n-1)/2)$ candidats de taille 2 à partir des *n* items trouvés fréquents lors de la première passe. L'objectif des auteurs est alors de réduire le nombre de candidats générés. Pour parvenir à cet objectif, PrefixSpan propose d'analyser, comme dans PSP, les préfixes communs que présentent les séquences de données de la base à traiter. A partir de cette analyse, contrairement à PSP, l'algorithme construit des bases de données intermédiaires, qui sont des projections de la base d'origine déduites à partir des préfixes relevés. Ensuite dans chaque base obtenue, PrefixSpan cherche à faire croître la taille des motifs séquentiels découverts, en appliquant la même méthode de manière récursive. Deux sortes de projections sont alors mises en place pour réaliser cette méthode : la projection dite « niveau par niveau » et la « bi-projection ». Au final, les auteurs proposent une méthode d'indexation permettant de considérer plusieurs bases virtuelles à partir d'une seule, dans le cas où les bases générées ne pourraient être maintenues en mémoire en raison de leurs tailles.

PrefixSpan fonctionne avec une écriture de la base différente de celle utilisée par GSP. En effet, l'algorithme requiert un format qui présente sur une ligne le numéro de client suivi de toutes ses transactions sous forme de séquence de données. Ce format nécessite une réécriture de la base de données avant de procéder à l'étape de fouille de données.

Considérons l'exemple suivant qui illustre l'algorithme et la structure utilisée.

Client	Séquence
C_1	<(a) (a b c) (a c) (d) (c f)>
C_2	<(a d) (c) (b c) (a e)>
C_3	<(e f) (a b) (d f) (c) (b)>
C_4	<(e) (g) (a f) (c) (b) (c)>

Figure 23 – DBSpan, une Base de Données exemple pour PrefixSpan

Préfixe	Base projetée (suffixes)	Motifs séquentiels
<a>	<(abc)(ac)(d)(cf)>, <(_d)(c)(bc)(ae)>, <_b)(df)(c)(b)>, <(_f)(c)(b)(c)>	<a>, <(a)(a)>,<(a)(b)>, <(a)(bc)>, <(a)(bc)(a)>, <(a)(b)(a)>, <(a)(b)(c)>, <(ab)>, <(ab)(c)>, <(ab)(d)>, <(ab)(f)>,<(ab)(d)(c)>, <(a)(c)(a)>, <(a)(c)(b)>,<(a)(c)(c)>,<(a)(d)>,<(a)(d)(c)>, <(a)(f)>
<b>	<(_c)(ac)(d)(cf)>, <(_c)(ae)>, <(df)(c)(b)>, <(c)>	<b>, <(b)(a)>, <(b)(c)>, <(bc)>, <(bc)(a)>, <(b)(d)>, <(b)(d)(c)>, <(b)(f)>
<c>	<(ac)(d(cf)>,<(bc)(ae)>,<b>, <(b)(c)>	<c>,<(c)(a)>, <(c) (b)>, <(c) (c)>
<d>	<(cf)>, <(c)(bc)(ae)>, <(_f)(cb)>	<d>,<(d)(b)>,<(d)(c)>,<(d)(c)(b)>
<e>	<(_f)(ab)(df)(c)(b)>, <(af)(c)(b)(c)>	<(e)>, <(e)(a)>, <(e)(a)(b)>, <(e)(a)(c)>, <(e)(a)(c)(b)>,<(e)(b)>,<(e)(b)(c)>,<(e)(c)>, <(e)(c)(b)>, <(e)(f)>, <(e)(f)(b)>, <(e)(f)(c)>, <(e)(f)(c)(b)>
<f>	<(ab)(df)(c)(b)>, <(c)(b)(c)>	<f>, <(f)(b)>, <(f)(b)(c)>,<(f)(c)>,<(f)(c)(b)>

Figure 24 – Résultat de PrefixSpan sur la base de données précédente

Exemple 20 :

Considérons la base de données DBSpan exemple de la Figure 23. La méthode de projection préfixée permet de procéder à une extraction de motifs séquentiels avec un support minimal de deux clients en appliquant les étapes suivantes :

- ***Etape 1 : trouver les items fréquents**. Pour cela, une passe sur la base de données va permettre de collecter le nombre de séquences supportant chaque item rencontré, et donc d'évaluer le support des items de la base. Les items trouvés sont (sous la forme <item > : support) : <a>:4, <b>:4, <c>:4, <d>: 3, <e>:3, <f>:3.*
- ***Etape 2 : diviser l'espace de recherche**. L'espace de recherche complet peut être divisé en six sous-ensembles puisqu'il y a six préfixes de taille 1 dans la base (i.e. les six items fréquents). Ces sous-ensembles seront : (1) les motifs séquentiels ayant pour préfixe <a>, (2) ceux ayant pour préfixes <b>, … et (6) ceux ayant pour préfixe <f>.*
- ***Etape 3 : trouver les sous-ensembles de motifs séquentiels**. Les sous-ensembles de motifs séquentiels peuvent être trouvés en construisant les projections préfixées des bases obtenues et en appliquant à nouveau l'algorithme de manière récursive. Les bases ainsi projetées et les motifs obtenus seront alors donnés dans la Figure 24.*

Le processus de découverte des motifs séquentiels, sur les bases projetées, peut alors se dérouler de la manière suivante :
Tout d'abord PrefixSpan cherche les sous séquences de données ayant pour préfixe <a>. Ainsi, seules les séquences contenant <a> sont à prendre en compte. De plus dans chacune de ces séquences, seul le suffixe doit être considéré. Par exemple avec la séquence <(e f) (a b) (d f) (c) (b)>, seule la sous séquence (le suffixe) < (_ b) (d f) (c) (b)> sera prise en compte (dans cette séquence, le caractère « _ » signifie que le préfixe était contenu dans le même itemset que « b »). Les séquences de DBSpan qui contiennent <a> sont alors projetées pour former $DBSpan_{<a>}$ *qui contient quatre suffixes < (a b c) (a c) (d) (c f)>, <(_ d) (c) (b c) (a e)>, <(_ b) (d f) (c) (b)> et <(_ f) (c) (b) (c)>. Une passe sur* $DBSpan_{<a>}$ *permet alors d'obtenir les motifs séquentiels de longueur 2 ayant <a> pour préfixe commun : <(a)(a)>:2, <(a)(b)>:4, <(a b)>:2, <(a) (c)>:4, <(a)(d)>:2 et <(a)(f)>:2.*
De façon récursive, toutes les séquences ayant pour préfixe <a> peuvent être partitionnées en 6 sous-ensembles : (1) celles qui ont pour préfixe < (a) (a)>, (2) celles qui ont pour préfixe <(a) (b)> ... et (6) celles qui ont pour préfixe <(a)(f)>. Ces motifs peuvent alors former de nouvelles bases projetées et chacune de ces bases peut alors être utilisée en entrée de l'algorithme toujours de manière récursive.
$DBSpan_{<(a)(a)>}$ *qui contient la projection des séquences ayant pour préfixe <(a) (a)>, contient une seule sous séquence (suffixe) : <(_ b c) (a c) (d) (c f)>. Comme aucune autre séquence fréquente ne peut être générée à partir d'une seule séquence (le support de 1 étant la borne inférieure), le processus quitte cette branche, pour remonter d'un niveau dans la récursivité.*
$DBSpan_{<(a)(b)>}$ *contient trois suffixes : <(_ c) (a c) (d) (c f)>, <(_ c) (a)> et <c>. En fonctionnant de manière récursive sur* $DBSpan_{<(a)(b)>}$*, le processus va trouver quatre motifs séquentiels : <(_ c)>, <(_ c) (a)>, <a> et <c>. Après remplacement du préfixe générique (« _ ») par le préfixe qui a conduit à cette base projetée (« <(a) (b)> »), nous obtenons les motifs suivants : <(a) (b c)>, <(a) (b c) (a)>, <(a) (b) (a)> et <(a) (b) (c)>.*
Le processus continue alors pour les bases projetées sur < (a b)>, <(a) (c)>, <(a) (d)> et <(a) (f)>. Ensuite les autres items fréquents (b, c, d, e et f) seront examinés en tant que préfixe de sous séquences de la base pour construire respectivement $DBSpan_{<b>}$, $DBSpan_{<c>}$, ... $DBSpan_{<f>}$ *et les analyser de manière récursive.*

2.2 Approches concernant la recherche de structures typiques

Dans cette section, nous nous intéressons aux approches voisines de notre problématique et qui concernent donc l'extraction de structures typiques dans de grandes bases de données d'objets semi structurés.

L'approche pionnière

Dans [WaLi98, WaLi99], les auteurs s'intéressent à la recherche de structures typiques dans des documents. Ils proposent une extension des travaux précédents liés à la recherche de transactions imbriquées [WaLi97a, WaLi97b] dans laquelle les données semi structurées, i.e. les documents dans ce cas, sont représentées par un graphe étiqueté *OEM*. Ce modèle permet de représenter de manière simple ces données, chaque objet du graphe *OEM* est composé d'un identifiant et d'une valeur. L'identifiant d'un objet *o*, noté *&o*, est défini de manière unique. La valeur d'un objet *o*, notée *val (&o)*, est soit un document atomique (entier, chaîne de caractères, …), soit une liste de documents de la forme $<l_1 :\&o_1,..., l_p : \&o_p>$, soit un ensemble de documents de la forme $\{l_1 : \&o_1,..., l_p : \&o_p\}$. En fait, les $\&o_i$ représentent les identifiants des sous documents et les l_i des étiquettes décrivant le rôle de ces sous documents. L'ordre dans les ensembles de documents n'a aucune importance contrairement aux séquences de documents. La répétition de sous documents est autorisée dans les ensembles et les séquences. La Figure 25 illustre trois objets *&1, &20, &24* contenant des informations sur trois clubs de football.

Dans ce contexte, une base de données de transactions est une collection de documents sur laquelle la recherche de structure typique va être effectuée. Dans les objets de la Figure 25, une base de transactions peut être vue comme une collection de documents « Club ». Les auteurs proposent également d'utiliser deux symboles spéciaux : '?' pour le caractère joker et '⊥' pour un schéma vide.

De manière à définir une notion de séquence appropriée au contexte, les auteurs proposent la notion de *tree-expression* définie de la manière suivante :

Définition 12 :

Quelque soit le label, *l** peut prendre pour valeur indifféremment *l* ou le caractère joker *?*. Soit te_i, $1 \leq i \leq p$, les *tree-expression* des documents étudiés sont définis par :

- ⊥ est la *tree-expression* de n'importe quel document.

- Si *val (&o) = {l_1 :&o_1,..., l_p : &o_p}* et *{i_1,..., i_k}* est un sous ensemble de *{1,..., p}, k>0, {li^*_1 :te_{i1},..., li^*_k : te_{ik}}* est la *tree-expression* du document *o*.

- Si *val (&o) = <l_1 :&o_1,..., l_p : &o_p>* et *<i_1,..., i_k>* est un sous ensemble de *<1,..., p>, k>0, <li^*_1 :te_{i1},..., li^*_k : te_{ik}>* est la *tree-expression* du document *o*.

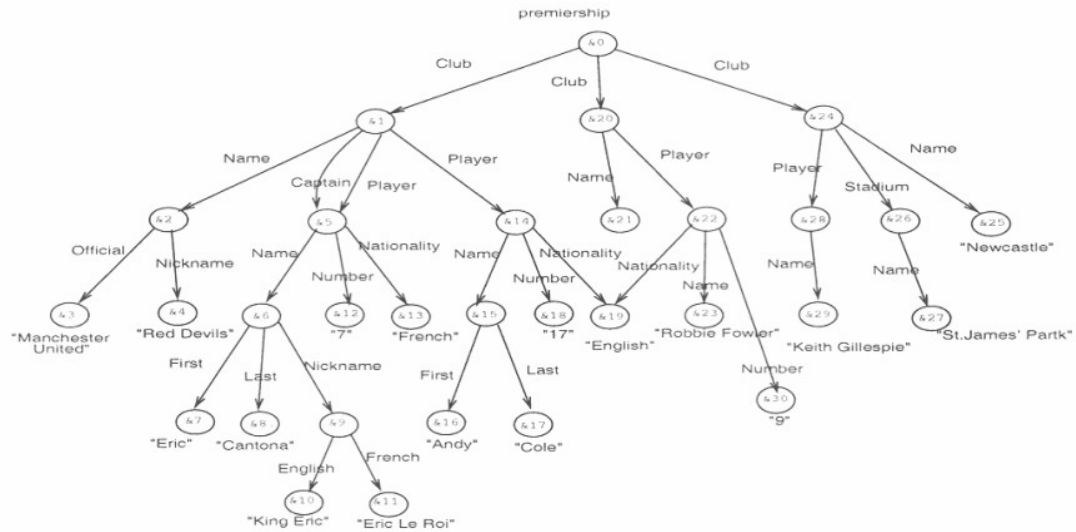


Figure 25 - Un graphe OEM

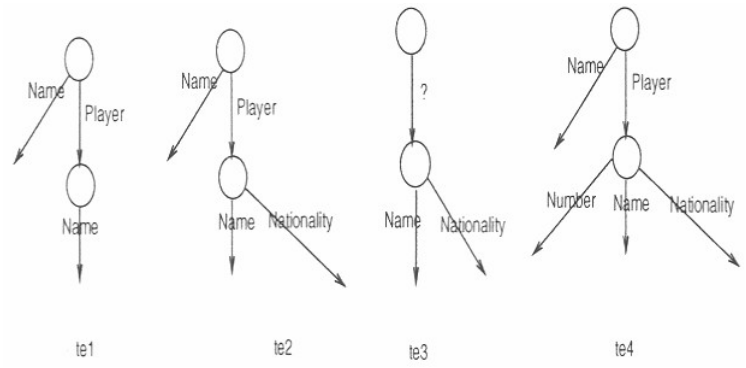


Figure 26 - Des tree-expressions du document concernant les clubs de football

Exemple 21 :

Soit la base de données de transactions {&1, &20, &24} des trois clubs de football. te_1 = {Player : {Name :⊥}, Name :⊥ } est une tree-expression de &1, &20, &24, tout comme l'est la même tree-expression dans laquelle nous aurions remplacé Player par ?. te_2 = {Player :{Name : ⊥, Nationality :⊥}, Name :⊥} est une tree-expression de &1 et &20, mais pas de &24. te_3 = { ?:{Name : ⊥, Nationality : ⊥}} est commune à &1,&20,&24. La Figure 26 illustre une représentation sous la forme d'arbre de te_1, te_2 et te_3.

Pour résoudre le problème lié aux cycles dans des documents, i.e. pour ne considérer que des graphes acycliques, les auteurs proposent de pouvoir nommer une feuille par un symbole spécial $\perp_i$, $i > 0$. $\perp_i$ représente l'alias utilisé pour représenter le nœud situé *i* nœuds au dessus de cette feuille dans la tree-expression. La Figure 27 illustre, à gauche, un document contenant un cycle et, à droite, la tree-expression correspondante.

De manière à pouvoir comparer entr'elles des tree-expressions, les auteurs définissent la notion de *weaker than* :

Définition 13 :

Weaker than est une fonction définie par :

- $\perp$ est *weaker than* toutes les trees-expressions.
- $\perp$ est *weaker than* elle-même.
- $\{l_1: te_1, ..., l_p: te_p\}$ est *weaker than* $\{l'_1: te'_1, ..., l'_q: te'_q\}$ si pour tout $1 \leq i \leq p$, te_i est *weaker than* certaines te'_{ji} telles que soit $l_i = l'_{ji}$ ou $l_i = ?$.
- $<l_1: te_1, ..., l_p: te_p>$ est *weaker than* $<l'_1: te'_1, ..., l'_q: te'_q>$ si pour tout $1 \leq i \leq p$, te_i est *weaker than* certaines te'_{ji} telles que soit $l_i = l'_{ji}$ ou $l_i = ?$.

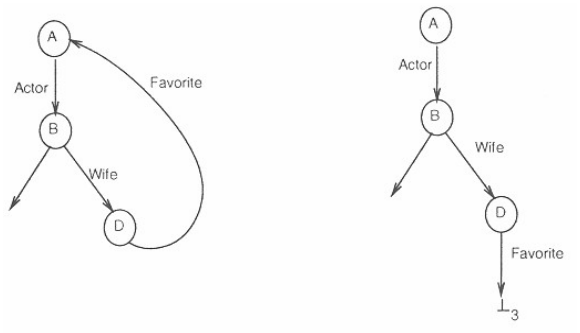


Figure 27 - Extension de la notion de tree-expression pour représenter les cycles

Intuitivement, si te_1 est *weaker than* te_2 toutes les informations structurelles contenues dans te_1 existent dans te_2. Tous ces éléments permettent de poser la problématique de la recherche d'informations structurelles en terme de tree-expression. Soit une tree-expression *te*, le support de *te* est le nombre de *documents racines d* (les parties de schéma à partir desquelles l'extraction est effectuée) tels que *te* est *weaker than d*. La problématique consiste donc à trouver toutes les tree-expressions te_i telles que le support de chaque te_i soit supérieur à une valeur *minSupp* fournie par l'utilisateur. Parmi ces *te* fréquentes nous ne retournerons que celles dites fréquentes maximales. Une tree-expression *te* est fréquente maximale si elle est fréquente et s'il n'existe aucune autre tree-expression fréquente *te'* telle que la relation *te weaker than te'* soit vraie.

Exemple 22 :

Dans la Figure 25, &1, &20, &24 sont des documents racines. Considérons les tree-expressions te_1, te_2, te_3 et te_4 de la Figure 26. Le support de te_1 et de te_3 est de 3, le support de te_2 et te_4 est de 2. Si minSupp a pour valeur 3 alors seuls te_1 et te_3 sont fréquentes maximales. Par contre, si minSupp vaut 2 toutes ces tree-expressions sont fréquentes mais seule te_4 est fréquente maximale.

Le principe général utilisé par les auteurs pour résoudre cette problématique est du type *générer élaguer* que nous avons vu dans les sections précédentes. Cependant pour prendre en compte la particularité des structures étudiées, i.e. non plates, les auteurs introduisent la notion de *k-tree-expression* comme métrique représentant la taille des candidats.

Définition 14 :

Une *k-tree-expression* est représentée par une séquence de *k* chemins de la forme $[T, l_1^{ji}, ..., l_n^{jn}, \perp]$ où $[T, l_1^{ji}, ..., l_n^{jn}, \perp_i]$, telle que les l_i sont les étiquettes sur un chemin simple de *G* commençant à un *document racine*. Les *ji* représentent les « *superscripts* » pour les étiquettes l_i, et aucun p_i n'est préfixe d'un autre. La dernière étiquette l_n ne peut être un caractère joker comme pour les tree-expressions. $[T, l_1^{ji}, ..., l_n^{jn}, \perp]$ où $[T, l_1^{ji}, ..., l_n^{jn}, \perp_i]$ sont appelés *path-expression*. Une *k-tree-expression* est donc une séquence $p_1, ..., p_k$ de *k-path-expressions*.

L'utilisation du superscript est nécessaire pour la résolution du problème suivant : soit *{l : &a, l : &b}* un document, *{l : ⊥, l :⊥}* est une tree-expression légale de ce document mais elle ne peut pas être construite à partir du seul chemin *[T, l,⊥]*. Les auteurs utilisent le superscript pour lever cette ambiguïté en générant deux chemins *[T, l_1,⊥]* et *[T, l_2,⊥]*.

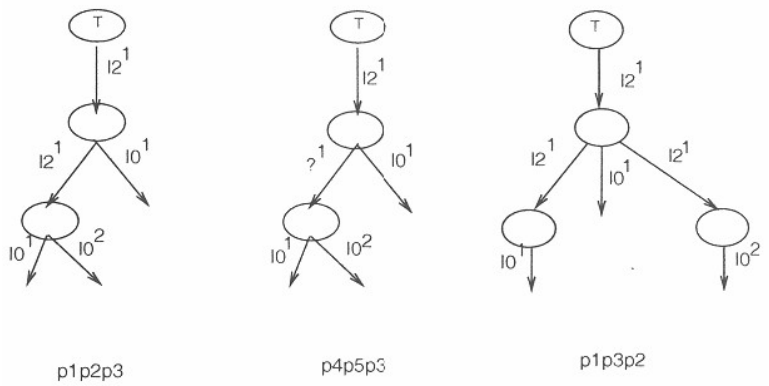


Figure 28 - Exemples de tree-expressions

Dans la Figure 28, te_1 peut être construite par la séquence $p_1p_2p_3$ et te_2 par la séquence $p_4p_5p_3$, où p_i, sont les path-expressions suivantes :

$p_1 = [T, l_2^1, l_2^1, l_0^1, \perp]$,
$p_2 = [T, l_2^1, l_2^1, l_0^2, \perp]$,
$p_3 = [T, l_2^1, l_0^1, \perp]$,
$p_4 = [T, l_2^1, ?^1, l_0^1, \perp]$,
$p_5 = [T, l_2^1, ?^1, l_0^2, \perp]$.

L'algorithme utilisé pour rechercher les tree-expressions se divise en trois phases : Calcul de F_1, Calcul des F_k, et Calcul des path-expressions maximales. L'algorithme est basé de manière similaire aux motifs séquentiels sur le théorème suivant :

Théorème 1 :

Soit p_i des path-expressions. Chaque *k-tree-expression* fréquente $p_1,..., p_k$ est construite par deux *k-1-tree-expressions* fréquentes $p_1,..., p_{k-2}, p_{k-1}$ et $p_1,..., p_{k-2}p_k$.

Phase I : calcul de F_1

L'objectif de cette phase est de trouver toutes les *1-tree-expressions* fréquentes. Ces expressions dépendent uniquement des étiquettes l_i, les superscripts ne sont pas utiles dans ce cas, le support d'une *1-tree-expression* est noté *sup* $(l_1,..., l_n,\perp)$ ou *sup* $(l_1,..., l_n, \perp_i)$. Il représente le nombre de documents racines pour lesquels il existe dans *G* un chemin simple de la forme $l_1,..., l_n$.

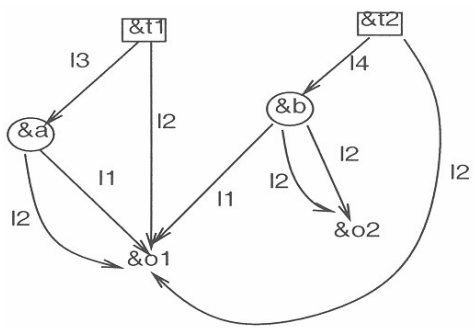


Figure 29 - Un exemple de graphe OEM

path-expression	supportée par	tree-expression représentée
p_1 : [T, l_2^1,⊥]	$\&t_1$, $\&t_2$	< l_2 : ⊥ >
p_2 : [T, l_2^2,⊥] (élaguée)	$\&t_1$, $\&t_2$	< l_2 : ⊥ >
p_3 : [T, ?^{1}, l_1^1,⊥]	$\&t_1$, $\&t_2$	< ? : {l_1 : ⊥} >
p_4 : [T, ?^{1}, l_2^1,⊥]	$\&t_1$, $\&t_2$	< ? : {l_2 : ⊥} >
p_5 : [T, ?^{1}, l_2^2,⊥] (élaguée)	$\&t_1$, $\&t_2$	< ? : {l_2 : ⊥} >
p_6 : [T, ?^{2}, l_1^1,⊥] (élaguée)	$\&t_1$, $\&t_2$	< ? : {l_1 : ⊥} >
P_7 : [T, ?^{2}, l_2^1,⊥] (élaguée)	$\&t_1$, $\&t_2$	< ? : {l_2 : ⊥} >
P_8 : [T, ?^{2}, l_2^2,⊥] (élaguée)	$\&t_1$, $\&t_2$	< ? : {l_2 : ⊥} >

Figure 30 - F_1

Exemple 23 :

Soit le graphe OEM de la Figure 29 contenant deux documents racines $\&t_1$ et $\&t_2$: val ($\&t1$) = <l_3 :$\&a$, l_2 :$\&o_1$> et val ($\&t_2$) = <l_4 :$\&b$, l_2 :$\&o_1$> avec val ($\&a$) = {l_1 :$\&o_1$, l_2 :$\&o_1$} et val ($\&b$) = {l_1 :$\&o_1$, l_2 :$\&o_2$, l_2 :$\&o_2$}. $\&o_1$ et $\&o_2$ sont des documents atomiques. Un nœud inscrit dans un rectangle représente une séquence de documents et un nœud inscrit dans un cercle représente un ensemble de documents. Pour une valeur de minSupp de 2, durant la phase I, 8 path-expressions p_1,..., p_8 sont fréquentes. Ces 8 path-expressions sont représentées dans la Figure 30. Par exemple, sup (l_2, ⊥) = 2 car chaque document racine possède un chemin simple étiqueté l_2.

A partir des deux path-expressions les autres path-expressions fréquentes sont générées de la même manière.

Phase II : calcul de F_k

Pour représenter l'espace de recherche les auteurs construisent un (k-1) arbre des candidats qui représente F_1,..., F_{k-1} (ensembles de *1,.., (k-1)* tree-expressions fréquentes). Cet arbre, noté $\prod_{k-1}$ est de profondeur *k-1*. Chaque nœud non racine représente une path-expression fréquente p_i de F_1. Chaque chemin depuis la racine jusqu'à une feuille de longueur $j \leq k\text{-}1$ est appelé j-candidate-path, il représente une j-tree-expression fréquente de F_j. Ainsi, chaque nœud feuille v de $\prod_{k-1}$ représente la path-expression en v et le candidate-path se terminant en v. Pour commencer $\prod_1$ a un fils pour chaque path-expression de F_1.

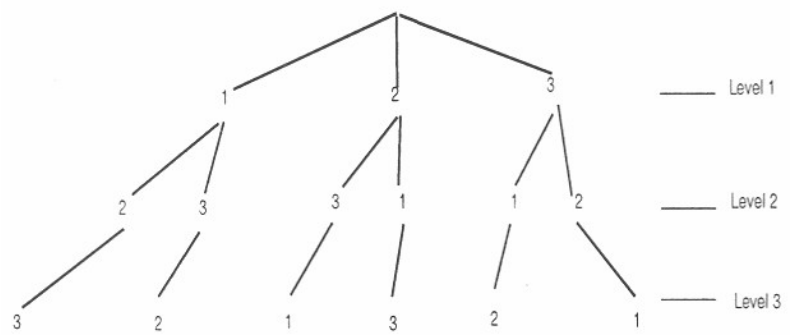


Figure 31 - $\prod_1$, $\prod_2$ et $\prod_3$

Pour générer $\prod_k$ à partir de $\prod_{k-1}$, pour chaque couple de paire ,..., p_{k-2}, p_{k-1} et p_1,..., $p_{k-2}p_k$ satisfaisant les conditions du théorème représenté par les nœuds feuilles l et l', un nouveau fils sous l est créé pour représenter le k-candidate-path p_1,..., $p_{k-2}p_{k-1}p_k$. Cette opération est appelée extension de l par l'. La Figure 31 illustre $\prod_1$, $\prod_2$ et $\prod_3$ générés par 3 path-expressions fréquentes p_1, p_2, p_3, avec la condition que toutes les 2-tree-expressions et 3-tree-expressions soit fréquentes. Pour compter le support des k-candidate-paths il suffit de parcourir tous les documents racines. La hiérarchie de chaque document racine t est utilisée en parcourant $\prod_k$ et le support de tous

les *k*-candidate-paths est incrémenté s'ils représentent une tree-expression de *t*. L'utilisation de la structure permet de déterminer plus rapidement si le *k*-candidate-path est une tree-expression du document *t*. En effet, pour un candidat de niveau *k*, pour $j < k$, si $p_1...p_j$ n'est pas une tree-expression de *t* il est inutile de poursuivre cette branche de l'arbre. L'élagage des candidats non fréquents revient à supprimer les nœuds feuilles de niveau *k* qui ont un support inférieur à *minSupp*. De manière à réduire l'espace de recherche, les auteurs proposent trois stratégies basées sur le fait qu'une tree-expression est construite de façon naturelle ou non.

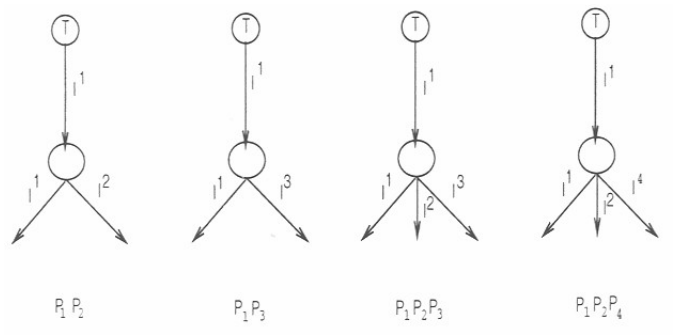


Figure 32 - Construction naturelle et non naturelle

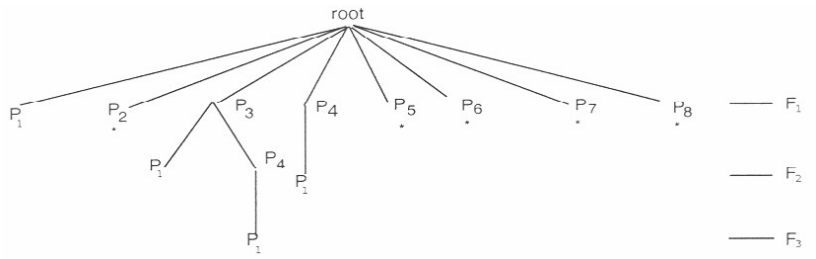
Une tree-expression est naturelle si pour chaque nœud non feuille avec *k* branches ayant la même étiquette, les superscripts sont *1,.., k* de gauche à droite, autrement la tree-expression est non naturelle. Par exemple dans la Figure 32, $p_1p_2p_3$ et p_1p_2 sont naturelles, mais p_1p_3 est non naturelle. Par contre $p_1p_2p_3$ qui est naturelle est construite à partir de p_1p_2 naturelle et p_1p_3 non naturelle. La première stratégie consiste à dire qu'un nœud feuille est étendu seulement s'il représente un candidate-path naturel. Après avoir réalisé toutes les extensions des éléments de $\prod_{k-1}$, les nœuds feuilles qui représentent des (*k-1*)-candidate-paths non naturels peuvent être élagués. Les auteurs nomment un *candidate-path super-unatural* si parmi les nœuds de cette tree-expression, les superscripts pour des branches étiquetées de manière identiques ne sont pas dans l'ordre. Etant donné que l'extension d'un *super-unatural candidate-path* génère toujours un *super-unatural candidate-path*, la seconde stratégie est la suivante : un extension est effectuée seulement si elle ne génère pas un *super-unatural candidate-path*. Enfin la dernière stratégie indique que si une extension de *l* génère un *candidate-path* fréquent alors on peut élaguer la tree-expression représentée par *l*. Cette dernière stratégie n'est à utiliser que pour obtenir les tree-expressions fréquentes maximales.

Exemple 24 :

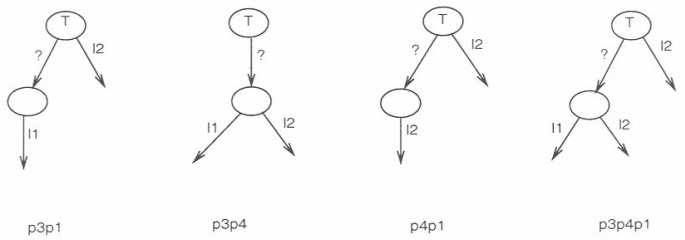
Considérons l'application des stratégies sur les éléments de la Figure 29. La Figure 33(a) illustre $\prod_1$, $\prod_2$ *et* $\prod_3$ *qui correspondent respectivement aux niveaux 1,2 et 3.3 2-candidate-paths fréquents et 1 3-candidate-path fréquents sont générés. Les candidate-paths non inclus dans* $\prod_i$*, sont soit non fréquents soit élagués par une des stratégies. Ainsi par exemple, tous les 2-candidate-paths* p_1p_i*, pour* $2 \leq i \leq 8$*, sont non fréquents. Les 1-candidate-paths* p_2*,* p_5*,* p_6*,* p_7*,* p_8 *marqués par une étoile dans la Figure 33 (a) ne sont pas étendus car ils sont non naturels. La Figure 33 (b) et (c) montre les ensembles* F_2 *et* F_3 *et leurs représentations sous forme de tree-expressions.*

Phase III : calcul des path-expressions maximales

Le résultat de la phase précédente est la structure $\prod_k$. L'objectif de cette phase est d'éliminer les tree-expressions fréquentes non maximales. Ces éléments sont obtenus en utilisant l'observation suivante : quelque soit $i > j$, aucune *i*-tree-expression ne peut être *weaker than* une *j*-tree-expression. Il suffit alors de parcourir les F_i*, 1..k* et de tenir compte de cette observation pour construire l'ensemble *M* des tree-expressions maximales. Dans la Figure 33, la seule tree-expression maximale est $p_3p_4\,p_1$ ou $< ? : \{l_1 : \perp, l_2 : \perp\}, l_2 : \perp >$.



(a) $\prod_1$, $\prod_2$ et $\prod_3$



(b) 2-tree-expressions et 3-tree-expressions

séquence de path-expressions	supportée par	tree-expression représentée
		F_2
p_3p_1	$\&t_1$, $\&t_2$	< ? : {l_1 : ⊥}, l_2 : ⊥ >
p_3p_4	$\&t_1$, $\&t_2$	< ? : {l_1 : ⊥, l_2 : ⊥ } >
p_4p_1	$\&t_1$, $\&t_2$	< ? : {l_2 : ⊥}, l_2 : ⊥ >
		F_3
p_3p_4 p_1	$\&t_1$, $\&t_2$	< ? : {l_1 : ⊥, l_2 : ⊥ }, l_2 : ⊥ >

(c) F_2 et F_3

Figure 33 - Fouille sur le graphe OEM

Autres approches sur la recherche d'arbres

De nombreux travaux se sont intéressés ces dernières années à la problématique de l'extraction de sous arbres. Les principales extensions ont été réalisées soit au travers d'une nouvelle structure de représentation, soit en proposant de nouvelles techniques de gestion des candidats.

Vers de nouvelles structures de représentation

L'approche proposée dans [MiSh01a] est assez similaire de celle proposée dans [WaLi98, WaLi99] et plus généralement à la méthode *générer élaguer*. La différence essentielle par rapport à l'approche précédente est la mise en œuvre d'une nouvelle structure de représentation, i.e. le Tag Pattern Tree, qui est en fait une variante de graphe OEM. Dans [Mish01b], les auteurs proposent de nouvelles optimisations liées à la gestion de ce type de graphe.
Dans [CoHo00], un algorithme est proposé pour extraire des motifs d'un arbre dirigé. La technique utilisée est celle de l'induction basée sur les graphes et l'utilisation de l'algorithme Subdue [CoHo00]. Subdue est un système d'apprentissage relationnel utilisé pour la recherche de sous structures qui apparaissent de manières répétitives dans la représentation par des graphes d'une base de données. Subdue commence par rechercher des sous structures qui permettent de compresser au mieux les graphes de la base de données en utilisant le principe de la description de longueur minimale (MDL) définit dans [Riss89]. Après avoir découvert la première sous structure, Subdue compresse le graphe et réitère le même processus. Une des caractéristiques importantes de Subdue est la possibilité d'utiliser des connaissances sur le domaine étudié sous la forme de sous structures

prédéfinies. Pour cela Subdue utilise un graphe étiqueté. Une sous structure est donc un sous graphe du graphe d'entrée construit à partir des informations contenues dans la base de données. Pour générer ces sous structures l'algorithme part d'un simple vecteur et l'étend petit à petit à la manière d'algorithme de type *générer élaguer*. La partie intéressante de cette approche réside dans sa capacité à effectuer des comparaisons presque exactes, c'est-à-dire de pouvoir comparer deux graphes g_1 et g_2 et de déterminer s'ils sont presque identiques à une métrique de distance près. [MaHo99], propose une extension de l'algorithme Subdue via un nouveau système de contraction du graphe d'entrée par les sous structures découvertes.

Dans [Zaki02], l'auteur propose deux algorithmes *TreeMinerH* et *TreeMinerV* pour la recherche d'arbres fréquents dans une forêt, i.e. une collection d'arbres où chaque arbre est un composant connecté de la forêt. En plus des étiquettes liées à chaque nœud, les auteurs utilisent une numérotation en profondeur d'abord pour numéroter les nœuds de *T* depuis *1* jusqu'à |*T*|. Le nœud ayant pour numéro *i* est noté n_i. Soit $T(n_i)$ le sous arbre ayant pour racine le nœud n_i et soit n_r la feuille la plus à droite dans $T(n_i)$. Les auteurs appellent *scope* du nœud n_i l'intervalle *[i, r]*.

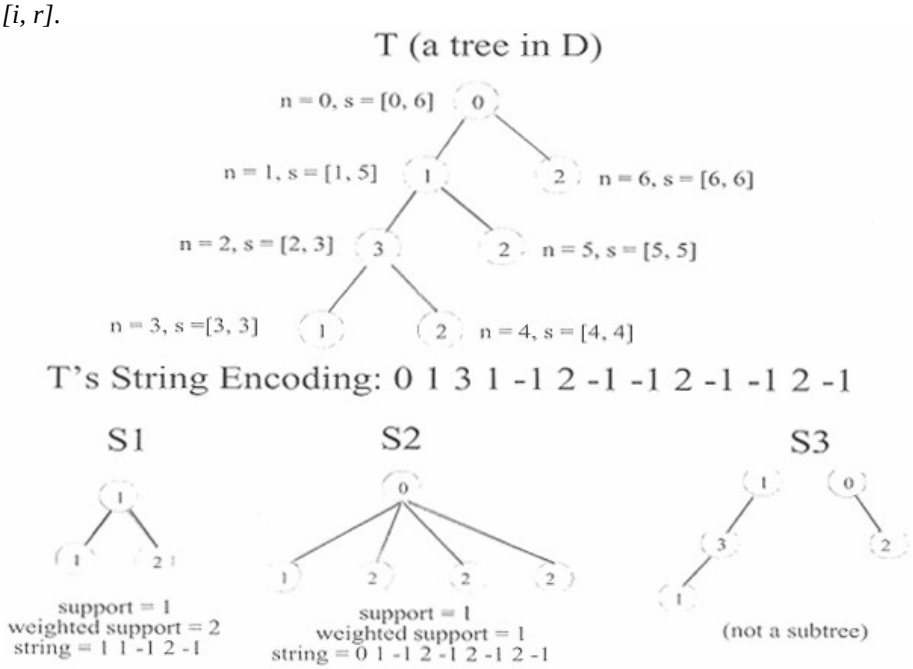


Figure 34 - Un exemple d'arbre, de sous arbres et leurs supports

Soit *D* une base de données d'arbres (i.e. une forêt), soit *minSupp* un support donné, le problème consiste donc à trouver tous les sous arbres *f* de *D* qui ont une valeur de support, i.e. le nombre d'arbres *d* de *D* pour lesquels *f* est sous arbre de *d*, supérieure à *minSupp*. Les auteurs nomment *weighted support* le nombre total d'occurrence de *f* dans *D*. Chaque occurrence de *f* peut être identifiée par son match label qui est défini par l'ensemble des positions qui correspondent dans *d* pour chaque nœud de *f*. Les sous arbres fréquents de taille *k* sont notés F_k.

Exemple 25 :

La Figure 34 montre un exemple d'arbre T de D. Sur la figure, nous pouvons voir pour chaque nœud sa numérotation et son scope. Ainsi l'arbre S_1 est un sous arbre de T ; il a un support de 1 et son weighted support vaut 2 car le nœud 2 apparaît aux positions n = 4 et n = 5. S_1 a deux matches label 134 et 135. S_2 est aussi un sous arbre de T, son match label est 03456. Cependant le motif S_3 n'est pas un sous arbre car il est déconnecté.

Les auteurs pour pouvoir générer de manière efficace, compter et manipuler les arbres utilisent une représentation d'arbre sous la forme de chaîne de caractères. Pour calculer la chaîne de caractères d'un arbre *T* noté T_s. Initialement T_s est vide, il suffit d'effectuer un parcours en profondeur depuis la racine en ajoutant l'étiquette *x* de chaque nœud à T_s. A chaque fois que nous retournons en arrière depuis un fils jusqu'à son parent l'étiquette unique -1 est ajoutée. Dans la Figure 34, le codage est représenté sous forme de chaîne de caractères de *T* et de ses sous arbres.
Pour systématiser la génération des candidats, les auteurs utilisent le concept des classes d'équivalences [Zaki00, Zaki01] en étendant ce concept à la notion de sous arbre. Soit *X*, *Y* deux sous arbres de taille *k* et X_s et Y_s leur

chaîne de caractères correspondante. Soit la fonction $p\ (X_s, i)$ qui renvoie la chaîne de caractères *préfixe* de X_s jusqu'au $i^{ème}$ nœud. On peut alors définir une relation d'équivalence θ_i entre les sous arbres de taille k X et Y de la façon suivante : $X\ \theta_i\ Y \Leftrightarrow p\ (X_s, i) = p\ (Y_s, i)$. Les sous arbres X et Y ont donc un préfixe commun de longueur i. Pour une génération efficace de candidats, nous nous intéressons au cas particulier où $i = k - 1$. Les auteurs proposent un lemme qui implique qu'un élément valide X peut être attaché uniquement sur des nœuds qui se trouvent sur un chemin allant de la racine jusqu'à la feuille la plus à droite de P.

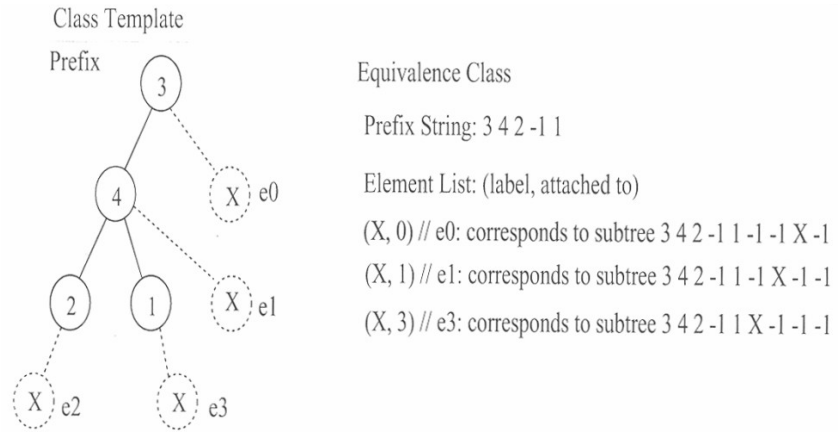


Figure 35 - Exemple de classe d'équivalence

Exemple 26 :

La Figure 35 représente un template pour des sous arbres de tailles 5 avec le même sous arbre préfixe P de taille 4 dont la chaîne de caractères est P_s = 3 4 2 -1 1. Dans cet exemple X est une étiquette arbitraire. Les trois positions valides pour une extension de P sont e_0, e_1 et e_3 car dans ce cas tous les sous arbres ont le même préfixe. Par contre la position e_2 n'est pas valide car P_{se2} = 3 4 2 X. La Figure 35 illustre aussi la méthode formelle qu'utilisent les auteurs pour stocker les classes d'équivalences ; elle consiste en la chaîne préfixe représentant la classe, une liste d'éléments. Chaque élément est une paire (X, p) où X est l'étiquette utilisée pour l'extension et p la position du nœud à étendre.

L'extension d'une classe est alors réalisée de la manière suivante. Soit P une classe préfixe dont la chaîne de caractère équivalente est P_s, soit (x, i) et (y, j) deux des éléments de la classe à étendre. Soit X_s la chaîne de caractères correspondant à x et soit Y_s la chaîne de caractères correspondant à y. Un opérateur de jointure (x) est défini sur ces deux éléments *noté* $(x, i)\ (x)\ (y, j)$:

Cas I - $(i = j)$: Si P_s <> *vide*, ajouter (y, j) et $(y, j+1)$ dans la nouvelle classe $[X]$.
Si P_s = *vide*, ajouter $(y, j+1)$ à la nouvelle classe $[X]$.

Cas II - $(i > j)$: ajouter (y, j) à la nouvelle classe $[X]$.

Cas III - $(i < j)$: aucun nouveau candidat n'est possible dans ce cas.

Alors tous les $(k-1)$ sous arbres de préfixe P de taille $k-1$ sont énumérés en appliquant l'opérateur de jointure sur chaque paire non ordonnée d'éléments (x, i) et (y, j).

Exemple 27 :

La Figure 36 illustre les trois cas possibles d'extension des classes. La classe préfixe P = (1 2) contient deux éléments (3, 1) et (4, 0). Le résultat de l'auto jointure (3, 1) (X) (3, 1) produit les candidats (3, 1) et (3, 2) représentés par les deux premiers sous arbres de la classe de gauche. Dans la jointure de (3, 1) (X) (4, 0) c'est le cas II qui s'applique. Le seul candidat possible est (4, 0), il est ajouté à la classe $P_{(3, 1)}$ représenté par le troisième sous arbre de la classe de gauche. La jointure (4, 0) (X) (3, 1) illustre le cas III, aucun nouveau candidat n'est généré. Enfin l'auto jointure (4, 0) (X) (4, 0) ajoute l'élément (4, 0) et (4, 1) à la classe $P_{(4, 0)}$ représentée par les deux sous arbres de la classe de droite.

Les différences essentielles entre TreeMinerH et TreeMinerV sont liées à la représentation des données et au parcours de recherche. TreeMinerH emploie une méthode de recherche par niveau itérative (en largeur d'abord) pour obtenir les sous arbres fréquents assez similaire à celle de GSP ou de PSP. TreeMinerV quand à elle est basée sur une représentation verticale et un parcours en profondeur d'abord de manière similaire à SPADE.

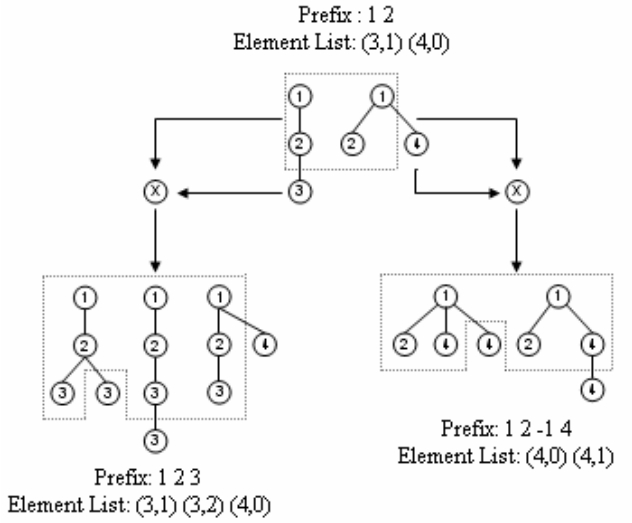


Figure 36 - Exemple de génération de candidats

Optimisation de la gestion des candidats

Dans [AsAb02], les auteurs proposent également de représenter les données semi structurées sous la forme d'un arbre étiqueté, ordonné. Une notion importante de leur approche est la représentation canonique d'un arbre étiqueté ordonné. Un arbre étiqueté ordonnée T de taille $k \geq 1$ (k = nombre de nœuds) est en forme normale si l'ensemble des nœuds de T noté $V_T = \{1,..., k\}$ et tous les éléments de V_T sont numérotés par un pré ordre transversal. Si un arbre étiqueté ordonné T est en forme normale alors sa racine notée $v_0 = 1$ et la feuille la plus à droite est $v_{k-1} = k$. Les auteurs proposent ainsi une fonction permettant la comparaison entre deux arbres T_1 et T_2 en forme normale pour détecter l'inclusion entre ces deux arbres. Cette fonction permettra de compter le support de Ti dans une base de données D constituée d'arbres étiquetés ordonnés.
De manière générale l'algorithme effectue une recherche par niveau et utilise également la technique d'énumération définie dans [Baya98].

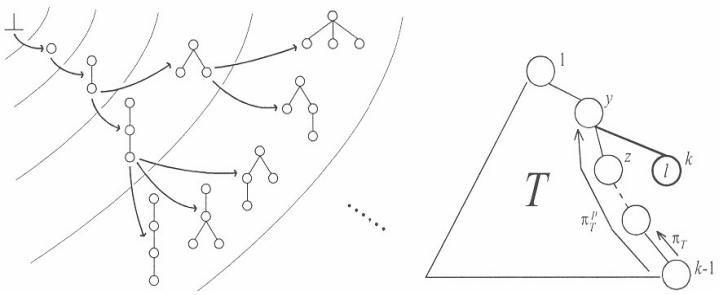


Figure 37 - L'extension par le nœud le plus à droite pour les arbres ordonnés

La Figure 37 à gauche illustre l'idée d'énumération de leur algorithme. Cette extension s'appuie sur le lemme suivant : Pour tous les $k \geq 2$, si S est un *(k-1)-pattern* alors n'importe laquelle des expansions de ce motif par le nœud le plus à droite T de S est aussi un *k-pattern*. De plus, Si T est un *k-pattern* alors il existe un unique *(k-1)-pattern* S tel que T soit l'expansion par le nœud le plus à droite de S.
Une méthode similaire est d'ailleurs utilisée dans [YaHa02]. La première partie du lemme propose une technique d'extension de motifs garantissant que l'arbre final est en forme normale, la seconde garantit l'unicité de l'arbre étendu. La Figure 37 à droite illustre la *(p, l)* extension de l'arbre T. l est une étiquette possible de L. p permet de repérer le nœud sur lequel l'extension est effectuée. Les auteurs peuvent alors construire un arbre d'énumération

pour *Te*, qui est le graphe acyclique *G*, où chaque nœud est un arbre ordonné *Te U {}*, et un arc existe dans *T* si et seulement si *T* est un successeur de *S*. Tous les nœuds de l'arbre T_1 sont considérés comme des successeurs de l'arbre vide de taille 0. Nous pouvons énumérer tous les arbres de *Te* en le parcourant par un algorithme de parcours en largeur ou en profondeur.

Dans [YaHa02], les auteurs utilisent une recherche par parcours en profondeur (Depth First Search – DFS) sans génération de candidats. La mise en place d'un ordre lexicographique et une méthode pour écrire la description d'un arbre (minimum DFS code) diminue considérablement le temps d'exécution d'un tel algorithme. L'approche gSpan proposée est en ce sens assez similaire à celle de PrefixSpan et est adaptée aux données manipulées.

Domaines proches de la problématique

De nombreux travaux, même s'ils ne traitent pas directement de la recherche de structures typiques sont suffisamment proches pour être examinés. Dans [HaFu95, SrAg95] des approches pour rechercher des règles d'association multi niveaux ont été proposées. Les auteurs supposent que pour rechercher de telles règles, ils disposent d'une base de données de transactions de clients et d'une taxonomie (hiérarchie is-a) entre les items achetés par les clients. Les travaux semblent proches de notre problématique dans la mesure où les éléments recherchés sont des ensembles d'items issus de différents niveaux de taxonomie. Mais ils restent assez différents, car ils s'intéressent uniquement à des données enrichies par une hiérarchie is-a alors que nous nous intéressons à des transactions dans lesquelles les données sont enrichies aussi bien par des hiérarchies « ensemble-de » que « liste-de ».

L'approche WARMR proposée dans [DeTo98] aborde la recherche de sous structures fréquentes dans des composants chimiques. Toutefois même si la problématique est similaire, ils considèrent que les sous structures recherchées sont exprimées sous la forme de requête DATALOG. De manière générale, les motifs autorisés sont spécifiés via un langage déclaratif (de la même manière que dans les approches basées sur la logique inductive (ILP)) et sont stockés dans un treillis. A partir d'un algorithme par niveau ils déterminent quelles sont les clauses les plus fréquentes et trouvent ainsi les structures fréquentes.
Enfin, proche de la problématique mais dans un contexte de recherche dans du texte, les auteurs de [BaGo96] présentent un algorithme pour la recherche d'expressions régulières sur des textes prétraités. Cette approche est intéressante car elle présente une méthode par automate pour la recherche d'un motif particulier (ou expression régulière) dans une chaîne longue (un texte). Leur approche permet de localiser ces motifs, mais aussi d'en compter le nombre d'occurrence à l'aide d'arbres préfixés et d'une méthode pour construire un automate qui permet de répondre à la problématique en un temps logarithmique (en moyenne).

2.3 Maintenance des connaissances

Il n'existe, à notre connaissance, aucun travaux de recherche pour maintenir la connaissance extraite lors de la recherche de sous arbres fréquents. Cependant, ces dernières années de nombreuses recherches ont été réalisées tout d'abord dans le cadre des règles d'association [AgPs95a, AyTa99, AyEl01, ChHa96, ChKa97, RaMo96, RaMo97, SaSr98, ThBo97] puis plus récemment pour la maintenance des motifs séquentiels. Au cours de cette section, nous nous intéressons plus particulièrement à ces derniers et présentons les principaux travaux existants.

Globalement, deux grandes tendances existent. La première est basée sur la notion de bordure négative introduite par [MaTo96] La seconde tendance ne considère que les séquences fréquentes et examine d'une part les nouvelles séquences qui peuvent intervenir lors de l'ajout d'items et d'autre part celles qui ne restent pas fréquentes dans le cas d'ajout de transaction ou bien de changement de valeur de support. Nous examinons, dans le reste de cette section les principaux travaux associés.

Approches utilisant la bordure négative : ISM et IUS

L'algorithme ISM, proposé par [PaZa99], est en fait une extension de l'algorithme SPADE, présenté dans la section 2.1.3.

La Figure 38 présente la base de données *DBspade* de la Figure 16 après une mise à jour.

A l'issue d'une première extraction avec l'algorithme SPADE sur DBspade, nous avons obtenu le treillis des séquences représenté dans la Figure 39. Le principe général d'ISM consiste à conserver, dans ce treillis, la bordure négative (*BN*) composée des *j*-candidats les plus bas de la hiérarchie d'inclusion qui n'ont pas été

retenus comme fréquents au cours de la passe sur la base. En d'autres termes, les membres de la bordure négative sont tels que : si $s \in BN$, il n'existe pas de s' tel que s' est fils de s et $s' \in BN$. La zone grisée de la Figure 39 représente la bordure négative associée à la base *DBspade*.

Client	Itemset	Items
C_1	10	A B
C_1	20	B
C_1	30	A B
C_1	**100**	**A C**
C_2	20	A C
C_2	30	A B C
C_2	50	B
C_3	10	A
C_3	30	B
C_3	40	A
C_3	**110**	**C**
C_3	**120**	**B**
C_4	30	A B
C_4	40	A
C_4	50	B
C_4	**140**	**C**

Figure 38 - *DBspade*, après la mise à jour

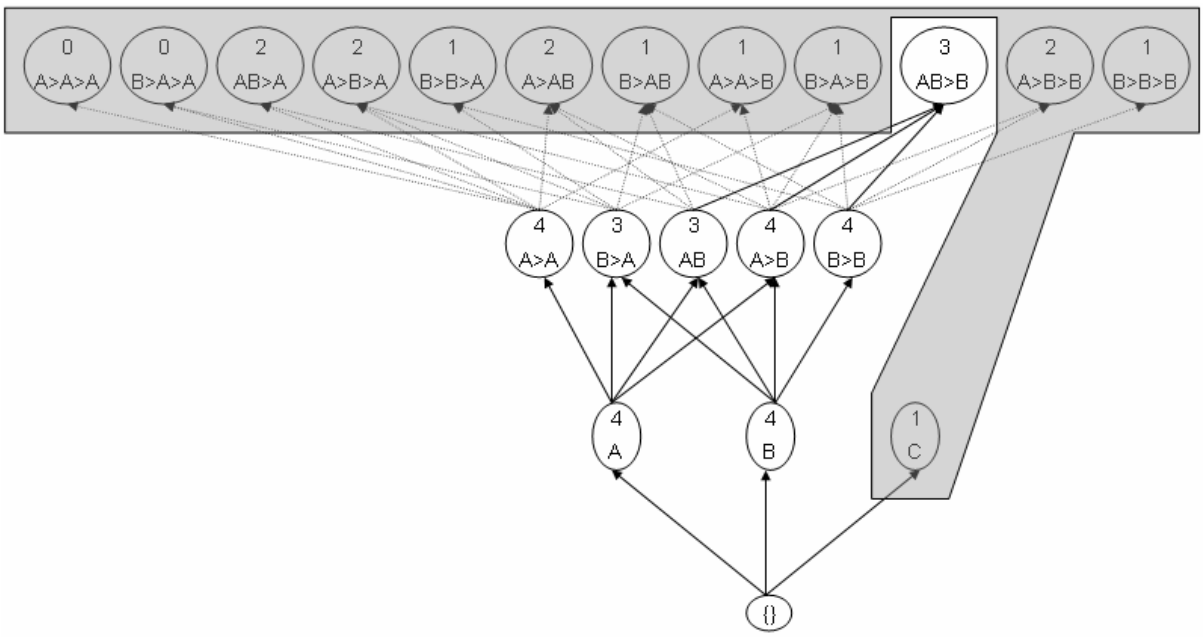


Figure 39 - La bordure négative, considérée par ISM après exécution de SPADE sur la base de données de la Figure 16

La première étape d'ISM, consiste à supprimer, à l'aide d'une passe sur la base, de l'ensemble des séquences fréquentes, celles qui ne le sont plus après la mise à jour. La bordure négative et le treillis sont à l'issue, de cette passe, mis à jour pour prendre en compte le nouveau support. Au cours de cette étape ISM détermine également les séquences de la bordure négative qui doivent migrer de *BN* vers *FS* (où *FS* désigne l'ensemble des séquences

fréquentes), i.e. l'ensemble des séquences qui deviennent fréquentes avec la nouvelle base. Enfin, l'ensemble des items fréquents à la fin de la passe est mis à jour pour insérer d'éventuels nouveaux items dans le treillis.

La seconde étape d'ISM, consiste à ré-examiner les nouveaux fréquents (les items ajoutés au treillis ou les séquences qui ont migrés de *BN* vers *FS*) un par un en reprenant le principe de génération de SPADE.

Exemple 28 :

Considérons l'item « C », de la base de données DBspade. Dans la base initiale, le support de cet item est de 1. Par contre, après mise à jour, sa valeur support est désormais de 4. Dans ce cas, l'item « C » migre de la bordure négative vers FS. De la même manière, les séquences < (A) (A) (B) > et < (A) (B) (B) > deviennent fréquentes et passent donc de BN vers FS. Toutes ces séquences sont découvertes après la première étape qui consiste à réévaluer le support de chaque séquence appartenant à la bordure négative dans le treillis.
La seconde étape consiste à générer les candidats en se limitant aux séquences qui viennent d'être ajoutés à FS. Par exemple, à partir des deux dernières séquences précédentes, nous pouvons générer le candidat < (A) (A) (B) (B) > qui sera, à l'issue de la passe sur la base de données, déterminé non fréquent et sera donc inséré dans la bordure négative.

Dans [ZhXu02a, ZhXu02b], les auteurs proposent l'algorithme IUS qui combine plusieurs notions : la bordure négative, la notion de suffixe et la notion de préfixe.

L'idée générale d'IUS est d'améliorer l'utilisation de la bordure négative en spécifiant un nouveau support minimal *Min_nbd_supp* qui permet de ne pas conserver la totalité de la bordure négative et ainsi d'en réduire son coût en terme d'espace. Cependant cette valeur de support n'est en fait définissable que de manière expérimentale.
L'algorithme IUS se divise globalement en deux étapes principales :

- Dans un premier temps, IUS utilise les fréquents et la bordure négative de *DB* et *db* comme candidats pour calculer de nouvelles séquences fréquentes ou membres de la bordure négative dans la base de données mise à jour *U*.

- Dans un deuxième temps, IUS combine les séquences fréquentes L^{DB} qui sont fréquentes dans *U* et non incluses dans L^{db} avec les éléments de L^{db} qui sont fréquents dans *U* et non inclus dans L^{DB}, afin de générer de nouveaux candidats dans U ainsi que dans la bordure négative. Le support des nouveaux candidats est alors calculé via une représentation préfixée et suffixée de la base.

Item	a	b	c	d	e	f	g	h	K
Support	4	3	2	2	1	1	1	1	1

Figure 40 - La base de données initiale

Item	a	F	B	d	c	e	g	h	k
Support	3	3	2	1	1	1	1	1	1

Figure 41 - db : la base de données incrémentale

Exemple 29 :

Considérons la base de données de la Figure 40. Considérons un support minimal de 50% et une valeur de Min_nbd_sup de 1. Pour des séquences de taille 1, nous avons : L_1^{DB} = {<a,4>,<b,3>,<c,2>,<d,2>}. Les autres éléments e, f, g, h et k sont donc éléments de la bordure négative : $NBD_1(DB)$ = {<e,1> ,<f,1>,<g,1>,<h,1>,<k,1>}. On obtient ainsi les ensembles L_2^{DB} = {<ab, 2>, <bc, 2>, <cd, 2>} et $NBD_2(DB)$ = {<ac, 1><ad, 1>, <bd, 1>} pour des séquences de taille 2 et L_3^{DB} = {<abc, 2>} et NBD_3 (DB) = {<bcd, 1>} pour les séquences de taille 3.
Considérons, à présent, la base incrémentale db de la Figure 41. Avec les mêmes paramètres que précédemment nous obtenons : L_1^{db} = {<a,3>,<b,2>,<f,3>} et $NBD_1(db)$ = {<c,1> ,<d,1>,<e,1>,<g,1>,<h,1>,<k,1>} pour le niveau 1, L_2^{db} = {<ab,2>,<bf,2>} et $NBD_2(db)$ = {<af,1>} pour le niveau 2, L_3^{db} = {<abf, 2>} pour le niveau

3. Ces ensembles permettent de calculer les candidats de la base de données mise à jour lors de la seconde phase de l'algorithme. L_1^U *= {<a,7>,<b,5>,<d,4>,<f,4>} et* $NBD_1(U)$ *= {<c,3> ,<e,2>,<g,2>,<h,2>,<k,2>} pour le niveau 1,* L_2^U *= {<ab,4>,<bd,4>} et* $NBD_2(U)$ *= {<ad,2>,<da,1>,<df,2>,<fd,1>} pour le niveau 2,* $NBD_3(U)$*= {<abd,3>} pour le niveau 3. On note que le nombre de transactions minimal pour être fréquent est de 4 (50% de 7).*

Enfin, même si l'approche KISP [LiLe98, LiLe03], n'est pas complètement basée sur la notion de bordure négative, elle en est suffisamment proche pour que nous la classions dans cette catégorie. En fait, la maintenance proposée par cet algorithme ne se situe pas comme les précédents dans sa capacité à prendre en compte les données issues d'une base *db* mais plutôt dans la possibilité de faire varier le support minimal d'une recherche de motifs séquentiels sur une base *DB* de données sans effectuer des calculs depuis zéro.

L'utilisation de KISP se divise en deux phases. Lors de la première recherche sur une base *DB*, il agit de la même manière que GSP. En plus, il génère la base fondamentale de connaissance *KB*. Elle est construite par un simple passage sur *DB* en comptant le support des l-séquences (où l est le nombre d'item de la séquence). Les résultats de ce calcul sont alors ajoutés à *KB* pour chaque passe effectuée par GSP. A la fin de l'exécution de GSP, *KB* contient toutes les séquences candidates (fréquentes ou non) dont le support a été calculé par GSP. *KB* garde la totalité des séquences pour deux raisons. La première réside dans le fait que si on diminue le support minimal alors certains candidats non fréquents initialement peuvent le devenir (comme dans la bordure négative). On pourrait alors obtenir directement ces motifs à partir de *KB* sans aucun accès à la base de données. Deuxièmement pour retrouver les motifs fréquents, le processus de fouilles de données de GSP compte le support de nombreux candidats même si ceux-ci se révèlent non fréquents par la suite. On pourra ainsi par la suite éviter d'effectuer les calculs de ces candidats en regardant directement le résultat dans *KB*. On obtiendra alors dans GSP une phase de comptage de support plus efficace et un arbre de hachage plus restreint étant donné que moins de candidats doivent être pris en considération. Une fois cette base générée, lors de futures recherches sur *DB* avec des supports différents, on va pouvoir réutiliser cette information stockée. Soit *sup_KB* le support pour lequel la *KB* a été calculée. L'utilisateur choisit un nouveau support *minsup*. Deux cas se présentent alors, soit *minsup* est plus grand que *sup_KB*, soit il est inférieur. Dans le premier cas la fouille de données se résume à regarder dans *KB* quels sont les motifs qui satisfont le nouveau support. *KB* ne subit alors aucune modification et aucun accès à *DB* n'est effectué. Dans le deuxième cas, il est nécessaire d'appliquer une extraction sur *DB* pour retrouver les nouveaux motifs qui ne sont pas contenus dans *KB*. La différence fondamentale entre KISP et GSP est que KISP à seulement besoin de compter les « nouveaux » candidats en les séparant des candidats déjà existants dans *KB*. Ensuite, les nouveaux candidats sont intégrés dans KB pour une extraction future et *sup_KB* prend pour valeur *minSupp*.

Ainsi au fur et à mesure des calculs, le contenu de la *KB* est étendu de manière incrémentale. L'efficacité en terme de réduction de comptage de support de *KB* va donc en s'accroissant. Les principes utilisés par KISP ressemblent plus ou moins à ceux de la bordure négative, la question de la taille de cette base de connaissance reste cependant entière, dans le cadre de motifs séquentiels longs et relativement fréquents elle aura une taille supérieure à celle de la bordure négative.

Approches n'utilisant pas la bordure négative : ISE et USSLMP

Contrairement aux approches précédentes, il existe deux méthodes qui ne tiennent pas compte de la bordure négative mais seulement des résultats obtenus pendant l'extraction pour minimiser les temps de recherches.

Ainsi, l'algorithme ISE [MaPo99a] permet de gérer l'ajout de transactions à des clients existants et l'ajout de nouveaux clients et de nouvelles transactions. Pour cela, ISE manipule trois ensembles de séquences. Tout d'abord, les séquences de *DB* qui peuvent devenir fréquentes si elles apparaissent dans db. Ensuite les séquences qui n'apparaissent pas sur *DB* mais qui sont fréquentes sur *db*. Enfin, les séquences fréquentes sur *U* qui n'appartiennent à aucun des deux ensembles précédents (i.e. supportées par au moins une séquence de données, partagée entre *DB* et *db*).

L'algorithme fonctionne de la manière suivante. Une première passe sur *db* permet d'identifier les items appartenant à *db*, ainsi que le nombre de leurs apparitions sur *db* et donc sur *U* (par addition avec cette information sur *DB*). A partir de cet ensemble d'items, ISE construit les séquences de taille 2 (L_1^{db} x L_1^{db}) qui apparaissent sur *db* (nouvelle passe sur *db*). Ces séquences forment l'ensemble *2-candExt*. Ces extensions sont alors vérifiées par une première passe sur *U* pour donner l'ensemble *2-freqExt*. ISE profite de cette passe pour associer aux sous-séquences de L^{DB} les items fréquents de L_1^{db}, et vérifier la validité de ces associations (i.e. une

sous-séquence de L^{DB} est-elle fréquente si on lui ajoute l'item i de L_1^{db} ?). Les associations validées sur *U* permettent d'obtenir l'ensemble *freqSeed* qui constitue la base des candidats qui seront désormais générés.
Les passes supplémentaires sont basées sur la méthode suivante : nous disposons de deux ensembles, *2-freqExt* et *freqSeed*. A partir de *2-freqExt* l'ensemble *3-candExt* est généré. A partir de ces ensembles *2-freqExt* et *freqSeed*, l'ensemble *candInc* est généré selon le principe suivant : soit *s1* appartient à *freqSeed* et *s2* appartient à *freqExt*, si le dernier item de *s1* est égal au premier de *s2*, alors on génére la séquence *s3* en supprimant le premier item de *s2* et en concaténant *s1* et *s2*, enfin on ajoute *s3* à *candInc*. Il faut alors vérifier ces deux ensembles (*candExt* et *candInc*) sur *U*.
Ce procédé est répété jusqu'à ce que l'on ne puisse plus générer d'extensions candidate dans *candExt* ou de séquences dans *candInc*. A la fin de ce processus, nous obtenons les séquences fréquentes sur *U* à partir des séquences de L^{DB} et des séquences maximales de *freqSeed* U *freqInc* U *freqExt*.

La Figure 42 illustre le fonctionnement général du processus d'ISE.

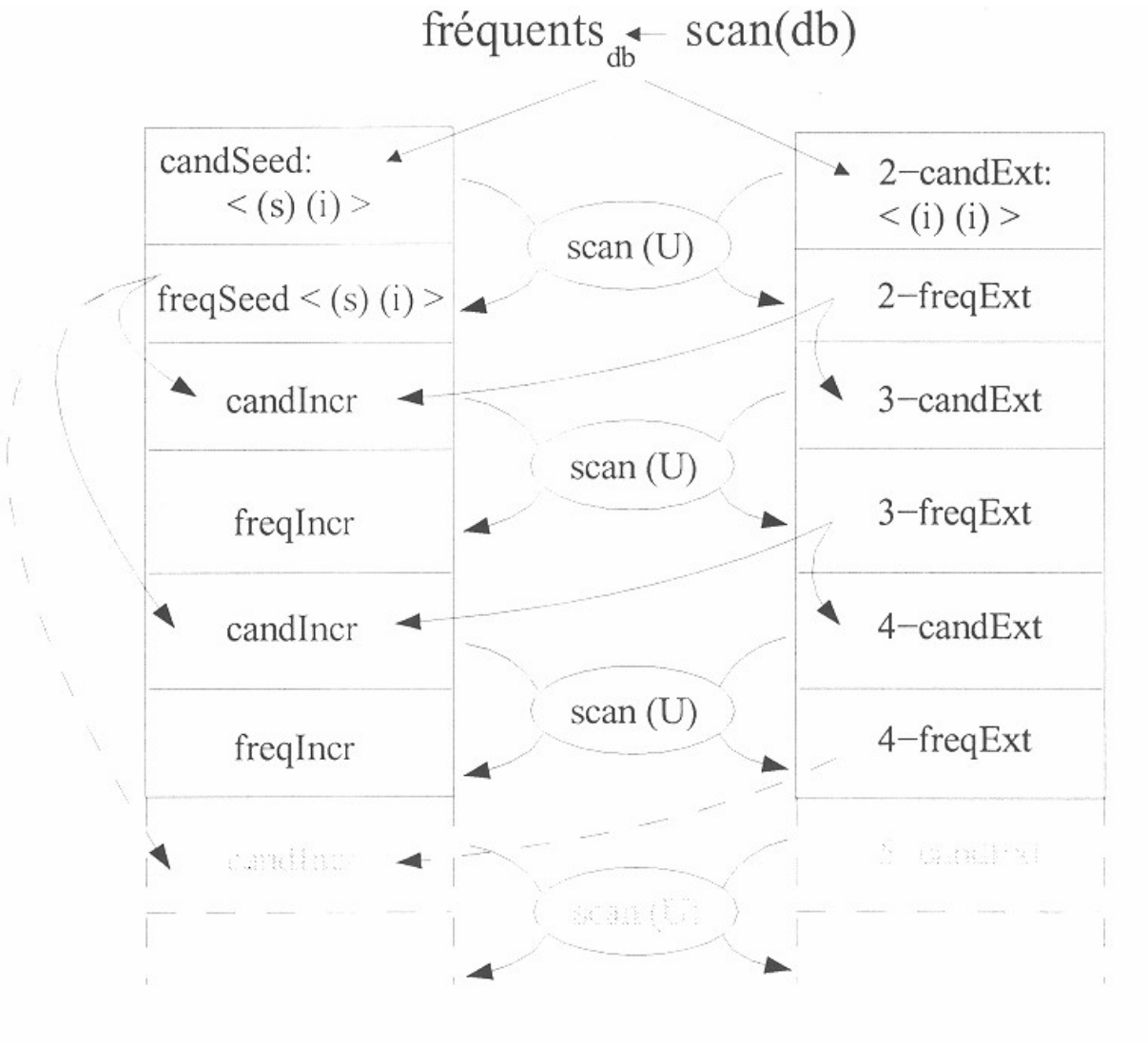


Figure 42 – L'approche ISE

Dans le cas de l'algorithme USSLP [YeCh01] le principe général consiste, dans un premier temps, à ajouter les nouvelles transactions issues de *db* dans la base initiale *DB*. Dans la nouvelle base *U*, les transactions obsolètes à effacer sont marquées.

CID	Séquence correspondante
1	*{a, b}* {a} {b} {b, c} {d}
2	{a} {b} {a, b, c}
3	*{a, b}* {a} {b} {c} {d}
4	{a,c} {b, c} {c}
5	{a, b, d}

Figure 43 - La base de données de transactions *U'* ordonnée selon l'identifiant du client

La deuxième phase de l'algorithme consiste à réordonner *U* selon l'identifiant du client (*CID*) pour former une base de données de transactions *U'*. La Figure 43 illustre une nouvelle base *U'* où les transactions ajoutées apparaissent en souligné et les transactions supprimées en italique. A partir de *U'*, une recherche de toutes les séquences fréquentes de taille 1 est réalisée. Puis chaque 1-séquence fréquente est encodée et la base de données *U'* est transformée (Cf. Figure 44 qui illustre pour un support de 60% la table de correspondance pour la base de la Figure 43). Si la 1-séquence était fréquente avant la mise à jour de la base de données, alors son code reste identique. Autrement, on affecte à la 1-séquence fréquente un code non utilisé dans la table de correspondante initiale. Finalement, on enlève de la table de correspondance initiale les 1-séquences qui ne sont plus fréquentes après la mise à jour.

Après avoir encodé chaque 1-séquence fréquente, *U*' est transformée et décomposée en une base de données mise à jour décomposée appelée *DU'*. Les méthodes de transformation sont similaires à celles utilisées par [AgSr95]. *DU'* est aussi divisée en deux parties l'une contenant les séquences des clients non changées, l'autre contenant les séquences des clients modifiées. Pendant cette décomposition les 1-séquences fréquentes de la partie des séquences des clients modifiées qui n'apparaissent pas dans les séquences des clients non changées sont mises dans l'ensemble B_{seed}. Par exemple dans la Figure 45, $B_{seed} = \{< \{E\} >\}$.
La cinquième phase de l'algorithme permet alors de générer toutes les séquences fréquentes à partir de la base de données décomposée mise à jour *DU'* et à partir du résultat, une recherche des motifs maximaux est réalisée.

Avant mise à jour		Après mise à jour	
Séquence fréquente de taille 1	Code	Séquence fréquente De taille 1	Code
{a}	A	{a}	A
{b}	B	{b}	B
{c}	C	{c}	C
{a, b}	D	{d}	E
		{b, c}	F

Figure 44 - La table de correspondance après mise à jour des transactions de la base de données

CID	Séquences des clients non changées
2	< {A} >, < {B}>, < {A} {B} {C} {F} >
4	< {A} {C} >, < {B} {C} {F} >, < {C} >
	Séquences des clients modifiées
1	*< {A} {B}*>, < {A} >, < {B} >, < {B} {C} {F} >, <{E} >
3	*< {A} {B} >*, < {A} >, < {B} >, < {C} >, < {E} >
5	< {A} {B} {E} >

Figure 45 - La base de données mise à jour *DU'* après la décomposition

3 Discussion

Dans ce chapitre, nous avons développé les problématiques étudiées ainsi que les différents travaux existants autour de ces thématiques. Dans cette discussion nous revenons d'abord sur les travaux existant autour de l'extraction de structures typiques en examinant leur adéquation par rapport à nos problématiques. Nous revenons ensuite sur le problème de la maintenance des connaissances extraites.

Examinons maintenant les travaux relatifs à la recherche de schémas ou plus exactement à la recherche de structures typiques telles que nous les avons définis. Parmi les travaux présentés, même si nos arbres possèdent des propriétés particulières (prise en compte des listes de et d'ensemble de), les algorithmes proposés par [WaLi99] se rapprochent le plus de notre problème dans la mesure où la prise en compte des niveaux d'imbrication est réalisée. Ce n'est pas le cas des autres approches, comme TreeMiner par exemple, qui s'intéressent plutôt à la recherche dans des forêts sans tenir compte du niveau d'imbrication des nœuds dans l'arbre. Revenons, par exemple, sur les schémas invalides (Figure 4 et Figure 5) et le schéma valide (Figure 6). En simplifiant ces schémas pour ne prendre en compte que la notion d'*ensemble de*, les approches à la Tree Miner considéreraient que ces trois schémas sont valides (il suffit d'examiner la Figure 34 pour s'en convaincre). Par contre dans le cas de [WaLi99], seul le schéma de la Figure 6 sera lui aussi considéré comme valide. Maintenant si l'on considère les différents opérateurs manipulés dans notre problématique, les stratégies d'élagage proposées par ce dernier ne sont plus adaptées dans la mesure où, avec la possibilité d'ordonner de manière différente, nous ne pouvons plus considérer de construction naturelle.

Revenons à présent sur la manière dont les algorithmes proposent de rechercher les structures typiques et, étant donné les nombreux travaux menés dans le domaine des motifs séquentiels et les nombreuses approches performantes (en temps de réponse), réexaminons notre problématique d'extraction sous ce point de vue. En effet, si nous considérons par exemple les approches de type TreeMiner, il est clair que la représentation proposée, sous la forme de chaîne de caractères, ressemble fortement à la description d'une séquence telle que celle définie en début de chapitre : un item correspond à un caractère et il existe un ordre total dans les items. Si nous examinons attentivement les propositions dans le cadre de la recherche de structures typiques, la plupart des propositions utilisent également des algorithmes assez similaires à ceux des motifs séquentiels : gSpan utilise la même philosophie que PrefixSpan mais en considérant que les données sont structurées sous la forme d'un arbre lexicographique lui-même représentant un parcours en profondeur des données ; TreeMiner est basée sur les classes d'équivalence déjà définies dans SPADE et la manière dont les tree-expressions sont gérées dans [WaLi99] est assez similaire à des approches de type *générer élaguer*.

De manière à tirer parti des avancées sur les algorithmes de recherche des motifs séquentiels, plusieurs questions se posent : est ce que les algorithmes sont utilisables tels quels pour répondre à notre problématique ? Si non quelles sont les améliorations à apporter à ces approches pour pouvoir conserver leur philosophie et ne pas dégrader leur efficacité ? Est-il possible de transformer les données pour qu'elles soient à même d'être directement utilisables par des algorithmes de motifs séquentiels. Enfin, il ne faut pas négliger que l'une des caractéristiques les plus importantes de l'extraction de motifs séquentiels se trouve dans la configuration des données. En effet, c'est en fonction de l'organisation des données que devrait être choisi l'algorithme d'extraction qui sera appliqué. Cette organisation peut se présenter de plusieurs façons : données denses ou creuses, corrélées ou pas, affichant beaucoup de séquences courtes ou peu de séquences très longues, comprenant un alphabet réduit ou vaste, etc. Malheureusement, aucune étude à ce jour (à notre connaissance) ne propose d'analyser la configuration des données et d'en déduire l'algorithme d'extraction le plus efficace. Ainsi, même si toutes les approches présentées ont montré leur intérêt dans de nombreux contextes applicatifs, il n'est pas démontré que telle ou telle approche est meilleure qu'une autre dans tous les cas. Par exemple, les algorithmes du type GSP ou PSP nécessitent une grande consommation de la mémoire pour stocker les différents candidats mais favorisent l'étape de comptage des candidats à l'aide de structures optimisées. Dans le cas des algorithmes comme SPAM ou SPADE, elles favorisent les coûts de stockage (projection de la base) aux dépens des coûts de calculs. Enfin, les approches comme PrefixSpan considèrent que la base de données peut tenir en mémoire et sont donc très efficaces pour de petites bases.

Si l'on considère à présent la maintenance des connaissances extraites, nous sommes confrontés au même type de problème. Quelle est aujourd'hui l'approche la plus efficace entre le stockage d'une bordure négative ou l'utilisation des connaissances extraites lors d'une fouille préalable ? La réponse est malheureusement « ni l'une ni l'autre » car elles dépendent complètement des types de données manipulées et des valeurs de supports utilisées. Ainsi, les approches basées sur les bordures négatives seront privilégiées si de nombreuses séquences sont stockées dans la bordure et évitent ainsi de repartir de zéro. Ces conditions sont donc fortement liées à la notion de support et le coût associé est la gestion d'une grande bordure négative. Les approches de la seconde

tendance, quand à elle, ne seront performantes que si les ajouts ne se font véritablement qu'en fin de transaction ou que le nombre de nouvelles transactions ne modifie pas trop les supports des séquences déjà extraites.
Dans ce cas, comment privilégier un type d'approche par rapport à l'autre ? La solution est sans doute d'essayer de proposer une approche en fonction des domaines d'application visés mais quelque soit ce choix, la contrainte de support minimal n'est toujours dépendante que de l'utilisateur final.

Enfin, pour finir cette discussion, nous pouvons remarquer que malgré les différentes propositions existantes pour rechercher des structures typiques, il n'existe pas, à l'heure actuelle, de travaux pour rechercher et maintenir la connaissance. Nous avons cependant vu en introduction de ce mémoire qu'étant donné les domaines d'applications visés, ce type d'approche devient de plus en plus indispensable.

Au cours des chapitres suivants, nous allons répondre à ces différentes questions au travers de différentes propositions adaptées à nos problématiques.

Chapitre III - Extraction de sous arbres fréquents

Le chapitre est organisé de la manière suivante. Dans un premier temps, nous décrivons une nouvelle manière de représenter les sous arbres stockés dans la base de données. Cette représentation étant basée sur une structure séquentielle nous prouvons que les deux représentations, i.e. un arbre et notre séquence, sont bijectives. Une première idée naïve, en examinant cette nouvelle représentation, pourrait être d'utiliser des algorithmes de type motifs séquentiels pour rechercher les sous arbres fréquents. Aussi dans la section 2 nous montrons les problèmes liés à l'utilisation de ces algorithmes. La section 3 propose une nouvelle approche basée sur la méthode *générer élaguer* pour rechercher les sous arbres fréquents. Appelée PSP_{tree}, elle utilise une structure d'arbre préfixée pour gérer efficacement les candidats. Dans la section 4, nous examinons une extension de la problématique initiale appelée recherche de sous arbres fréquents généralisés et nous proposons des algorithmes pour la résoudre. Les expérimentations menées avec ces algorithmes sont décrites dans la section 5. Enfin nous concluons ce chapitre par une discussion.

1 Vers une nouvelle représentation des sous arbres

De manière à résoudre efficacement la problématique de la recherche de sous arbres fréquents, il est indispensable de trouver une représentation de ceux-ci. Pour décrire un graphe *G*, de nombreuses représentations peuvent être utilisées [GoMi90]. Une manière classique de représenter des arbres étiquetés consiste souvent à utiliser des matrices d'adjacences. Cependant, cette approche nécessite une quantité d'informations nécessaires égale à N^2 où N représente le nombre de sommets du graphe. En outre, dans le cas de sommets peu denses, il existe une très importante perte d'espace et il devient donc préférable de décrire uniquement les termes non nuls de la matrice d'adjacence. De manière à ne stocker que les informations non nulles, nous utilisons donc une approche basée sur une liste ou plus particulièrement une séquence.

Nous considérons les séquences manipulées définies de la manière suivante.

Définition 15 :

Soit $N = \{n_1, n_2, \ldots n_n\}$ l'ensemble des nœuds d'un graphe $G = (N,B)$. Soit $I(n_i)$ et $prof(n_i)$ respectivement l'indicateur d'ordre et la position du nœud n_i dans le graphe *G*. Un élément est un ensemble de nœuds non vide noté $e = (\{I(n_1), n_1, prof(n_1)\} \{I(n_2), n_2, prof(n_2)\}, \ldots \{I(i_k), i_k, prof(i_k)\})$. Une séquence est une liste ordonnée, non vide, d'éléments notée $<e_1\ e_2\ \ldots\ e_n>$ où e_i est un élément.

Remarque : Par souci de lisibilité, nous utiliserons par la suite indifféremment la notation condensée $(I(n_i)n_{iprof(ni)})$ au lieu de $\{I(n_i), n_i, prof(n_i)\}$. Ainsi, l'élément $(\{\oplus, Personne, 1\})$ s'écrira en notation condensée $(\oplus Personne_1)$.

La transformation d'un arbre en une séquence est réalisée via l'algorithme *TreeToSequence* qui correspond à un parcours en profondeur d'abord de l'arbre et qui tient compte des niveaux d'imbrication.

Exemple 30 :

Considérons l'arbre ordonné, enraciné et étiqueté de la Figure 46. Sa transformation avec l'algorithme TreeToSequence est la suivante : $<(\{\oplus, Personne, 1\}) (\{\oplus, Identité, 2\}) (\{\otimes, Adresse, 3\}) (\{\perp, Numéro, 4\}) (\{\perp, Rue, 4\}) (\{\perp, CodePostal, 4\}) (\{\perp, Nom, 3\}) (\{\perp, Prénom, 3\})>$. *Soit, en utilisant la notation condensée,* $<(\oplus Personne_1) (\oplus Identité_2) (\otimes Adresse_3) (\perp Numéro_4) (\perp Rue_4) (\perp CodePostal_4) (\perp Nom_3) (\perp Prénom_3)>$.

Algorithm TreeToSequence

Input : Un graphe G (on note sa racine r)
Output : Une liste S ordonnée

1 : $S = (\{I(r),r,prof(r)\})$; // Ajout d'un élément à la séquence
2 : **If** (I(r) = $\oplus$ **and** $\forall$ $x_i \in Fils(r)$, $I(x_i) = \perp$ **then**
3 : $S = S \,\&\, (\{I(x_1),x_1,prof(x_1)\} \ldots \{I(x_n),x_n,prof(x_n)\})$
4 : **Else For each** $x \in Fils(r)$ **do**
5 : $S = S \,\&\, TreeToSequence(G,x)$ // & représente l'opérateur de concaténation
6 : **enddo**

Algorithme 1 -TreeToSequence

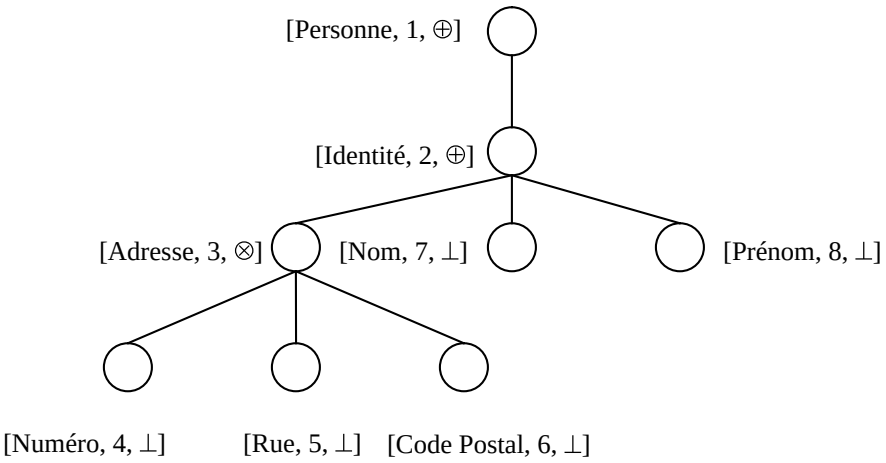


Figure 46 – Un exemple d'arbre ordonné, enraciné et étiqueté

Soit *G* un arbre, la séquence obtenue à partir de l'algorithme *TreeToSequence* possède les propriétés suivantes :

Propriété 4 *(unicité)* :

Chaque sommet et chaque arc de *G* apparaissent une seule fois dans *S*.

En effet, par construction, un nœud de l'arbre *G* n'est examiné qu'une seule fois pour construire la séquence. Dans le cas des arcs, ceux-ci sont pris en compte en tenant compte d'un nœud et du niveau de profondeur de ces fils garantissant ainsi que tous les arcs de *G* apparaissent également une seule fois.

Propriété 5 (*racine*) :

Il existe un sommet *x* dans *S* sans prédécesseur. Nous appelons ce sommet racine de *S* et le notons par *r*. Par construction ce sommet est toujours le premier dans *S*. Ainsi, pour tout autre sommet *x* de *S* différent de *r*, il existe un sommet prédécesseur *y* dans *S* représentant de l'arc *(y, x)*.

Propriété 6 (*arcs*) :

Soit un sommet *x* dans *S* différent de *r*. S'il existe dans *G* un arc *(x, y)* et un arc *(z, x)* alors *x* est un sommet successeur de *z* dans *S* et *x* est un sommet prédécesseur de *y* dans *S*.

Propriété 7 *(connexité) :*

Pour chaque sommet *x* dans *S* il existe un chemin et un seul (un sous-ensemble d'arcs) de *r* à *x*.

En parcourant la séquence, il n'existe qu'un nœud parent pour chaque nœud fils. Le nœud parent précède le nœud fils et possède une profondeur d'un niveau inférieur.

Nous considérons, à présent, la transformation inverse permettant, à partir d'une séquence *S*, d'obtenir le graphe initial *G*. De manière générale, l'algorithme *SequenceToTree* fonctionne de la manière suivante : l'arborescence est créée sous la forme d'une liste de sommets pointant sur une liste de fils. Dans un premier temps, la liste est créée à l'aide de tous les nœuds contenus dans la séquence. La racine de l'arbre, de profondeur 1, est considérée comme un cas particulier et nécessite de rechercher dans la séquence tous les nœuds dont la profondeur est égale à 2 (lignes 2-3). Ensuite, chaque élément est accédé et intégré en tant que fils du précédent si sa profondeur est supérieure d'un niveau (procédure *ajouterfils*), i.e. en fonction de *Propriété 6* et *Propriété 7*. Si la profondeur n'est pas supérieure, la procédure *rechercherpere* recherche dans les nœuds précédents quel est le nœud dont le niveau est immédiatement inférieur. Ainsi, à l'issue de l'algorithme, chaque nœud de l'arbre possédera un indicateur sur l'ordre de ses fils et la liste de tous ses fils.

Algorithm SequenceToTree

Input : Une liste S ordonnée
Output : Une arborescence G présentée comme une liste de sommets pointant sur une liste de fils

```
1 : ForEach e ∈ S do G+=e ; enddo // création de la liste
2 : r = FirstNode(S) ; // cas particulier de la racine
3 : ForEach e ∈ S do
4 :   if (prof(e)=prof (r)+1) then ajouterfils(e, r)endif
5 : enddo
6 : prec = r ; // cas des autres éléments de la séquence
7 : Foreach e ∈ S do
8 :   if (prof (e)=prof(prec)+1) then ajouterfils(e,prec)
9 :   else ajouterfils(e,rechercherpere(e)); endif
10 :    prec = e ;
11 : enddo
```

Algorithme 2 - SequenceToTree

Nœud	Ident	Fils
Personne	⊕	Identité
Identité	⊕	Adresse Nom Prénom
Adresse	⊗	Numéro Rue CodePostal
Numéro	⊥	⊥
Rue	⊥	⊥
CodePostal	⊥	⊥
Nom	⊥	⊥
Prenom	⊥	⊥

Figure 47 – La liste représentant l'arborescence

Exemple 31 :

Considérons la séquence obtenue précédemment : <(Personne$_1$) (Identité$_2$) (Adresse$_3$) (Numéro$_4$) (Rue$_4$) (CodePostal$_4$) (Nom$_3$) (Prénom$_3$)>. Dans un premier temps, la liste est créée avec tous les nœuds de la séquence (C.f. Figure 47). Etant donné que l'indicateur de la racine est un «ensemble-de » (i.e., ⊕), il est ajouté dans la structure de la liste. La racine étant de niveau 1, un premier parcours est effectué dans la séquence pour rechercher tous les fils de niveaux 2 (Identité). Ceux-ci sont alors ajoutés comme fils de Personne. L'algorithme se poursuit en recherchant à partir d'identité tous les nœuds d'un niveau supérieur (Adresse). Nom est rattaché

à Identité qui est le premier sommet de niveau immédiatement inférieur. L'algorithme se termine lorsque tous les nœuds de la séquence ont été analysés.

Lemme 1 :

La transformation *S = TreeToSequence (G, r)* qui transforme une arborescence *G* en une séquence *S* est une bijection dont l'ensemble de départ est l'ensemble des arborescences et l'ensemble d'arrivée est restreint à l'ensemble de toutes les séquences images des arborescences.

Preuve : Pour prouver que cette application est bijective il faut montrer qu'elle est surjective ce qui est trivial par définition de l'ensemble d'arrivée et qu'elle est injective. Pour cela considérons deux arborescences *A* et *B*. Pour montrer que la transformation *S=TreeToSequence (A, r)* est une bijection, il suffit de montrer que *B=SequenceToTree (S)* est l'arborescence identique à l'arborescence *A*. Soit LS_A la liste représentant l'arbre *A* définie telle que chaque nœud de l'arbre pointe vers un indicateur d'ordre et ses différents fils. Soit LS_B la liste obtenue après application de *SequenceToTree*. Les sommets de la liste apparaissent une seule fois dans *S* (propriété 6) et donc une seule fois dans LS_B (ligne 1 de l'algorithme *SequenceToTree*). Ainsi, les ensembles LS_A et LS_B sont égaux. Supposons maintenant qu'un sommet *y* appartienne aux ensembles $LS_A(x)$ et $LS_B(z)$ avec $x \neq z$. Ceci implique que le sommet *y* a deux pères soit dans l'arborescence *A* (ce qui est impossible par définition de l'arborescence) soit dans la liste *S* (ce qui est impossible par construction). Cette contradiction prouve que pour un sommet quelconque *x*, tous les sommets de l'ensemble $LS_A(x)$ sont dans l'ensemble $LS_B(x)$ et inversement, donc $LS_A(x)=LS_B(x)$. Donc $A = B$.

Arbre_id	Arbre
T_1	Personne : {identite : {adresse : ⊥ , nom : ⊥}}
T_2	Personne : {identite : {adresse : <numero : ⊥, rue : ⊥, codepostal : ⊥>, compagnie : ⊥, directeur : <prenom : ⊥, nom : ⊥>, nom : ⊥}}
T_3	Personne : {identite : { adresse : <numero : ⊥, rue : ⊥, codepostal : ⊥>}, nom : ⊥}
T_4	Personne : {identite : { adresse : <numero : ⊥>, compagnie : ⊥, nom : ⊥}}
T_5	Personne : {identité : { adresse : ⊥, nom : ⊥ }}
T_6	Personne : {identite : { adresse : <numero : ⊥, rue : ⊥, codepostal : ⊥>, directeur : < prenom : ⊥ , nom : ⊥, >, nom : ⊥}}

Figure 48 – Une Base de Données exemple

Arbre_id	Arbre
T_1	(⊕Personne$_1$) (⊕identite$_2$) (⊥adresse$_3$ ⊥nom$_3$)
T_2	(⊕Personne$_1$) (⊕identite$_2$) (⊗adresse$_3$) (⊥numero$_4$) (⊥rue$_4$) (⊥codepostal$_4$) (⊥compagnie$_3$) (⊗directeur$_3$) (⊥prenom$_4$) (⊥nom$_4$) (⊥nom$_3$)
T_3	(⊕Personne$_1$) (⊕identite$_2$) (⊗adresse$_3$) (⊥numero$_4$) (⊥rue$_4$) (⊥codepostal$_4$) (⊥nom$_3$)
T_4	(⊕Personne$_1$) (⊕identite$_2$) (⊗adresse$_3$) (⊥numero$_4$) (⊥compagnie$_3$) (⊥nom$_3$)
T_5	(⊕Personne$_1$) (⊕identite$_2$) (⊥adresse$_3$) (⊥nom$_3$)
T_6	(⊕Personne$_1$) (⊕identite$_2$) (⊗adresse$_3$) (⊥numero$_4$) (⊥rue$_4$) (⊥codepostal$_4$) (⊗directeur$_3$) (⊥prenom$_4$) (⊥nom$_4$) (⊥nom$_3$)

Figure 49 – La Base de Données après application de TreeToSequence

Exemple 32 :

Considérons la base DB de la Figure 48. Nous avons vu dans le chapitre précédent que pour une valeur de support minimal de 50%, le seul sous arbre fréquent est : Personne : {identite :{adresse :<numero : ⊥, rue : ⊥, codepostal : ⊥ >}, nom}. Considérons maintenant la traduction de la base de données de sous arbre par application de l'algorithme TreeToSequence (C.f. Figure 49), nous avons $<(\oplus Personne_1)\ (\oplus identité_2)\ (\otimes adresse_3)\ (\perp numéro_4)\ (\perp rue_4)\ (\perp codepostal_4)\ (\perp nom_3)>$.

2 Une première approche naïve : les algorithmes de motifs séquentiels

Une première approche naïve pour résoudre la problématique de la recherche des sous arbres fréquents, à partir de leur réécriture pourrait être d'utiliser des algorithmes de motifs séquentiels. Cependant, ce type d'approche possède des défauts majeurs qui ne permettent pas de l'utiliser. Les premiers problèmes sont liés au fait que les algorithmes de motifs sont dédiés à des structures plates et ne peuvent donc pas prendre en compte le niveau d'imbrication des sous arbres quelle que soit leur représentation. Le second problème est par contre lié à la génération des candidats qui est effectuée sans tenir compte du niveau d'imbrication des candidats.

Problématique de sous arbres frères

Le premier problème est lié à la représentation des arbres sous la forme de séquence et la manière dont un candidat est évalué par rapport à une séquence de données. Etant donné que les algorithmes de recherche de motifs séquentiels ne s'intéressent qu'à la présence (ou l'absence) de motifs récurrents, i.e. qu'ils ne s'intéressent qu'à des structures plates, ils ne peuvent donc pas prendre en compte le fait qu'un sous arbre peut être lié à un autre sous arbre. De manière à illustrer ce problème, considérons l'exemple suivant.

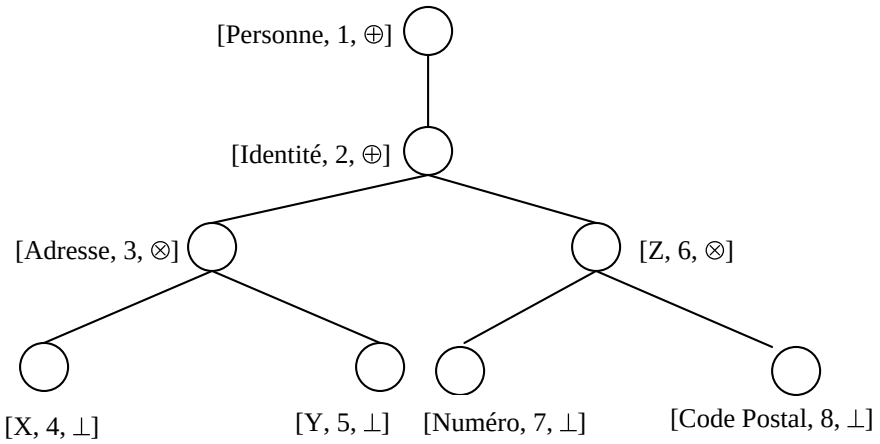


Figure 50 – L'arbre associé à la séquence S_1

Exemple 33 :

Considérons l'arbre représenté par la Figure 50. Sa représentation sous la forme de séquence est la suivante $S_1 = <(\oplus Personne_1)\ (\oplus identité_2)\ (\otimes adresse_3)\ (\perp X_4)\ (\perp Y_4)\ (\otimes Z_3)\ (\perp numéro_4)\ (\perp codepostal_4)>$. *Supposons que X, Y et Z ne soient pas fréquents, i.e. leur nombre d'occurrences dans la base de données est inférieur au support minimal. Supposons à présent qu'il existe une séquence de la forme :* $S_2 = <(\oplus Personne_1)\ (\oplus identité_2)\ (\otimes adresse_3)\ (\perp numéro_4)\ (\perp codepostal_4)\ (\perp nom_3)\ (\perp prénom_3)>$. *La Figure 51 illustre l'arbre associé à cette séquence. Nous pouvons constater sur les figures que les deux arbres ne sont pas inclus. Cependant, les algorithmes de parcours utilisés par les approches de recherche de motifs séquentiels montrent que* S_2 *est inclus dans* S_1. *En effet, ils déterminent que* $<(\oplus Personne_1)\ (\oplus identité_2)\ (\otimes adresse_3)\ (\perp numéro_4)\ (\perp codepostal_4)>$ *est inclus dans* S_1 *dans la mesure où ils ne peuvent pas tenir compte du fait que* $(\perp numéro_4)$ *et* $(\perp codepostal_4)$ *sont des fils de* $(\otimes Z_3)$.

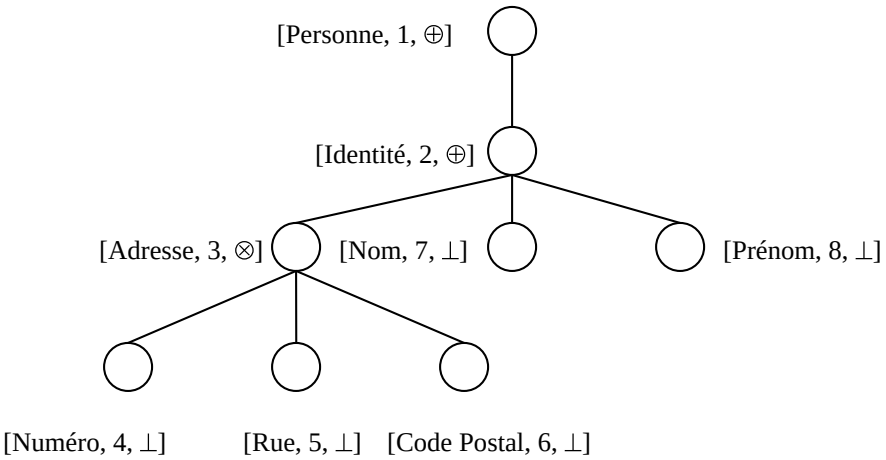


Figure 51 – L'arbre associé à la séquence S_2

Problématique du niveau d'imbrication

La problématique du niveau d'imbrication est un sous problème du précédent et concerne un même sous arbre. Toujours lié au problème des structures plates analysées par les algorithmes de motifs séquentiels, nous pouvons constater que ces derniers peuvent faire apparaître des éléments qui ne sont pas reliés entre eux dans la mesure où, à aucun moment, il n'y a de contrainte sur les successeurs d'un élément.

Exemple 34 :

Considérons la séquence S_1 = <(⊕Personne$_1$) (⊕identité$_2$) (⊥nom$_3$) (⊥prénom$_3$)>. Considérons à présent, la séquence S_2 = <(⊕Personne$_1$) (⊕Z$_2$) (⊥nom$_3$) (⊥prénom$_3$)>. L'application d'algorithme de motifs séquentiels trouve comme fréquent sur ces deux séquences : <(⊕Personne$_1$) (⊥nom$_3$) (⊥prénom$_3$)>. Ceci est contradictoire avec notre problématique dans la mesure où il n'existe pas de nœud dans l'arbre liant (⊕Personne$_1$) à (⊥nom$_3$) et (⊥prénom$_3$).

Problématique de la génération des candidats

L'un des derniers problèmes à l'utilisation des algorithmes de motifs séquentiels réside dans la génération des candidats. En effet, le principe général d'utilisation de ces algorithmes consiste à étendre d'un item à chaque passe les itemsets jugés fréquents dans la passe courante. En outre, ils ne tiennent pas compte du niveau d'imbrication, i.e. du fait qu'il existe une relation « ancêtre de ». En conservant ce principe, un trop grand nombre de candidats est généré de manière inutile car dans notre cas seules les extensions qui tiennent compte de la profondeur peuvent être réalisées.

Exemple 35 :

Considérons les éléments fréquents (⊕Personne$_1$), (⊕identité$_2$) et (⊥nom$_3$). En utilisant un algorithme classique de génération des candidats nous générons, par exemple, < (⊕Personne$_1$) (⊥nom$_3$)> qui est inutile dans la mesure où un tel arbre ne peut pas être fréquent par rapport à notre problématique.

3 Une nouvelle approche pour la recherche de sous arbres fréquents

Les limites précédentes nous ont donc amenés à proposer une nouvelle approche pour rechercher les sous arbres fréquents de la base de données. La méthode destinée à extraire les sous arbres maximaux repose sur le principe de la méthode *générer élaguer* décrite précédemment et dont nous redonnons l'algorithme ci-dessous. L'algorithme général procède donc par passes successives sur la base de données en alternant génération puis suppression des candidats non fréquents. Cet algorithme cesse quand aucun candidat généré n'est retenu.

Algorithm General

Input : Un support minimal (*minSupp*) et une base de données *DB*
Output : L'ensemble L^{DB} des éléments ayant une fréquence d'apparitions supérieure à *minSupp*.

1 : $k = 1$; $L^{DB} = \emptyset$// les éléments fréquents
2 : $C_1 = \{\{i\} / i \in$ *ensemble des éléments de DB*$\}$;
3 : **For each** $d \in DB$ **do** *CompterSupport* $(C_1, minSupp, d)$; **enddo**
4: $L^1 = \{c \in C_1 / support(c) \geq minSupp)$;
5 : **While** $(L^k \neq \emptyset)$ **do**
6 : *genererCandidats* (C_k) ;
7 : **For each** $d \in DB$ **do** *CompterSupport* $(C_k, minSupp, d)$; **enddo**
8 : $L^k = \{c \in C_k / support(c) \geq minSupp\}$;
9 : $L^{DB} \leftarrow L^{DB} \cup L^k$
10 : $k += 1$;
12 : **endWhile**
12 : **Return** L^{DB}

Algorithme 3 – L'algorithme général

A partir de l'algorithme général ainsi défini, nous proposons l'approche PSP_{tree} pour rechercher des sous arbres fréquents dans la base de données [LaMa00a, LaMa00b]. Cette approche, PSP_{tree}, est basée sur une structure d'arbre préfixée.

3.1 Utilisation d'une structure préfixée : l'algorithme PSP_{tree}

Dans le chapitre précédent, nous avons présenté l'algorithme PSP pour rechercher des motifs séquentiels. Nous proposons un nouvel algorithme basé sur la même structure de données, i.e. une structure préfixée, pour résoudre la problématique de la recherche de sous arbre dans une base de données.
L'idée initiale de PSP_{tree}, comme dans PSP, consiste à représenter les candidats dans une structure qui factorise les mêmes séquences candidates en fonction de leur préfixe. L'un des autres avantages de cette structure est que les séquences fréquentes y sont elles mêmes stockées facilitant ainsi la génération des candidats.

Dans un premier temps nous présentons la structure utilisée en montrant comment les éléments sont stockés par niveau. Nous nous intéressons ensuite à la génération des candidats et montrons comment utiliser efficacement la structure pour optimiser cette génération. Enfin, nous décrivons l'algorithme complet PSP_{tree}.

3.1.1 Description de la structure PrefixTree

Dans cette section, nous présentons la structure PrefixTree utilisée dans l'algorithme PSP_{tree}. L'avantage d'utiliser une structure préfixée réside dans le fait que les éléments manipulés, i.e. les sous arbres, possèdent des structures communes. Ainsi, en factorisant le stockage des structures communes, il devient possible d'optimiser l'espace de stockage des éléments candidats et fréquents. L'arbre préfixé ne stocke pas seulement les candidats mais permet de les générer rapidement.

Dans un premier temps, nous nous intéressons au niveau 1 de l'arbre préfixé qui consiste en fait à représenter tous les éléments de la base de données et nous montrons ensuite comment les éléments sont stockés dans les niveaux supérieurs. Etant donné que nous ne nous intéressons qu'aux sous arbres possédant une même racine, l'arbre préfixé est forcément déséquilibré, i.e. les extensions de niveau supérieur à 2, ne peuvent se faire que par rapport au seul nœud racine de l'arbre. Par contre, nous verrons que les extensions de niveau 2 seront indispensables pour faciliter la génération des candidats.

Considérons comment les éléments candidats de niveau 1 sont stockés dans la structure préfixée.

K=1 : Chaque branche issue de la racine relie celle-ci à une feuille qui représente un élément. Chacune de ces feuilles contient l'élément et son support (le nombre de ses apparitions dans la base). Ce support est calculé par un algorithme comptant le nombre d'occurrences de l'élément dans la base et dans des séquences distinctes.

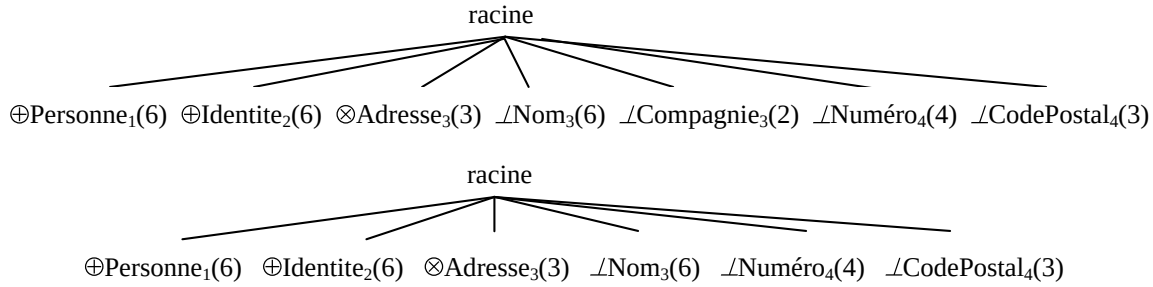


Figure 52 – La structure préfixée au niveau 1

Exemple 36 :

La Figure 52 illustre (en haut) un exemple de l'état de la structure après évaluation du support. Considérons que pour être retenue une séquence doit apparaître au moins dans trois séquences de données. La figure du bas représente la structure contenant uniquement les éléments fréquents.

K>1 : chaque nœud de l'arbre représente un élément pour une ou plusieurs séquences. Chaque chemin de la racine de l'arbre vers une feuille représente en fait un sous arbre. Pour distinguer les éléments situés dans une même parenthèse, nous avons séparé les fils d'un nœud en deux catégories : « même ensemble » et « autre ensemble ».

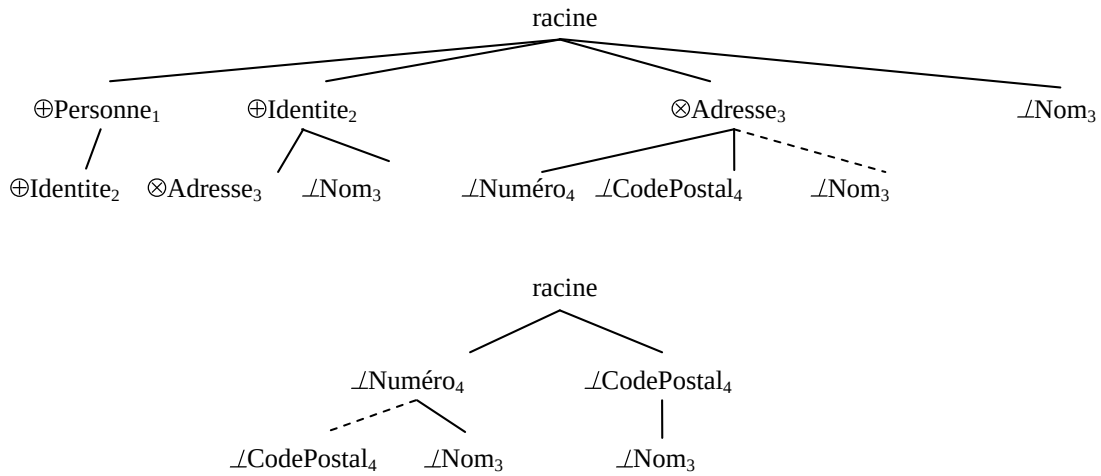


Figure 53 – La structure préfixée au niveau 2

Exemple 37 :

L'arbre de la Figure 53 représente des séquences fréquentes de taille 2. Une branche en trait plein indique que les éléments ne sont pas dans le même ensemble (par exemple <(⊕Personne1) (⊕Identite2)>) et une branche en trait pointillé entre deux éléments indique que ceux-ci sont dans le même ensemble, i.e. <(⊿Numéro4 ⊿CodePostal4)>.

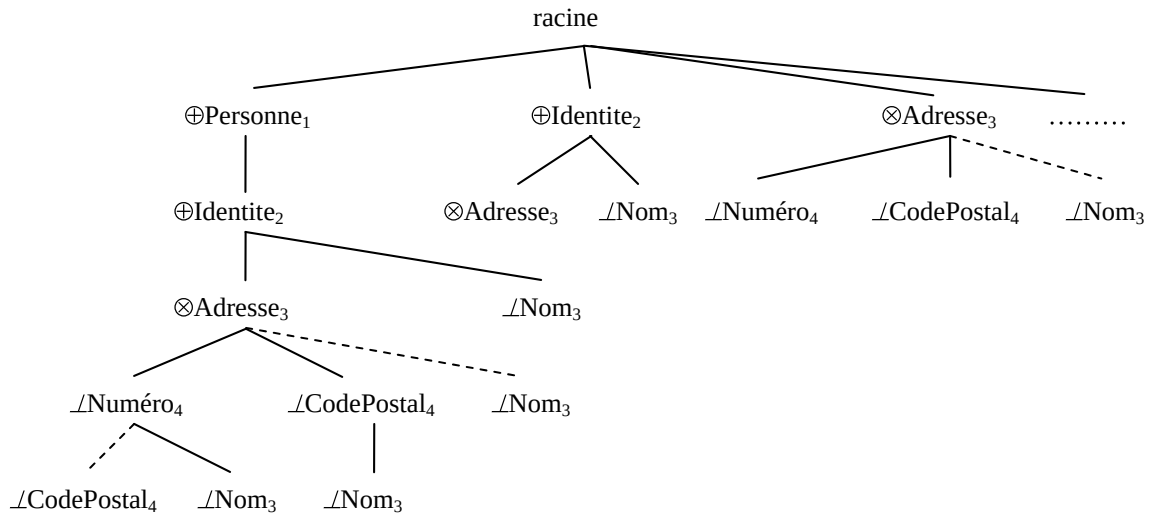


Figure 54 – La structure préfixée au niveau 5

Nous disions précédemment que la structure préfixée était déséquilibrée dès que le niveau était supérieur à 2. La Figure 54 illustre la structure au niveau 5. Dans la mesure où nous recherchons des sous arbres partageant la même racine, nous constatons, en effet, que les seules extensions possibles se font sur la racine, i.e. l'élément $\oplus Personne_1$. Dans ce cas, les candidats représentés dans l'arbre sont les suivants : C_1 = {<($\oplus Personne_1$) ($\oplus Identite_2$) ($\otimes Adresse_3$) ($\perp Numéro_4$ $\perp CodePostal_4$)>}, C_2 = {<($\oplus Personne_1$) ($\oplus Identite_2$) ($\otimes Adresse_3$) ($\perp Numéro_4$) ($\perp Nom_3$)>} et C_3 = {<($\oplus Personne_1$) ($\oplus Identite_2$) ($\otimes Adresse_3$) ($\perp CodePostal_4$) ($\perp Nom_3$)>}.

3.1.2 Génération des candidats

Dans cette section, nous présentons comment les candidats sont générés à partir de la structure préfixée. Nous allons donc examiner les candidats en fonction de *k*, leur longueur.

K=1 : Les candidats de longueur 1 correspondent à l'ensemble des éléments de la base de données. Ainsi, un premier parcours sur la base de données est réalisé de manière à créer la structure préfixée de niveau 1.

K = 2 : L'objectif de cette étape est de définir les relations possibles existantes entre les fréquents de longueur 1 pour générer les candidats de longueur 2. Ces relations seront appelées par la suite extensions possibles d'un nœud. Un nœud peut être étendu soit par ses fils, soit par ses frères, soit par les frères de ses ancêtres. Nous pouvons voir les extensions possibles comme la liste des suivants dans un parcours en profondeur des arbres de la forêt sur laquelle l'extraction de connaissances est effectuée. L'algorithme *extensionPossible*, à partir de la structure préfixée de niveau 1 et la base de données de départ *DB*, génère pour chaque nœud les différentes extensions possibles en étendant la structure au niveau 2. Chaque nœud représentant un 1-fréquent de la structure est noté n_i. Les candidats de longueur 2 issus de n_i sont les structures représentées par la séquence (n_i) (r_i) (où r_i est une extension possible de n_i). Elles sont représentées dans la structure par la création d'un arc entre n_i et r_i. Nous ne conserverons dans la structure que les candidats de longueur 2 dont le support est supérieur au support minimal. Pour vérifier le support de ces candidats nous utilisons la fonction *verifyCandidate* définie dans la section 3.1.3, cette procédure supprime de *T*, les candidats dont le support n'est pas suffisant.

L'objectif de l'algorithme *extensionPossible* est de mettre à jour la structure *T* en ayant pris en compte toutes les relations existantes entre les différents éléments de niveau 1. Dans un premier temps, (lignes 1-5), trois listes associées à chaque élément fréquent de niveau 1 sont initialisées à ∅. Elles contiendront respectivement les *Fils*, *Frères* et *FrèresAncêtres* de ces nœuds. Ensuite, nous considérons chacun des arbres stockés dans *DB* noté t_j pour mettre à jour, si nécessaire, les différentes listes pour les éléments de niveau 1 fréquents noté F_i^1 de la structure *T* au niveau 1 noté T^1 (lignes 6 – 35). Dans un premier temps pour chaque couple (t_i, F_i^1) considéré, nous initialisons la liste des *FrèresAncêtresLocaux* de F_i^1 à ∅. Les F_i^1 sont considérés dans l'ordre d'un parcours en profondeur de t_j. La première étape de l'algorithme consiste à mettre à jour, pour le F_i^1 considéré, la liste de ses *Fils* en explorant l'arbre t_i (lignes 9 – 14). Si de tels fils sont trouvés, nous appelons la fonction *ajoute* sur la liste des fils associée à ce F_i^1. Cette fonction ajoute l'élément f_i à cette liste s'il n'y appartenait pas déjà en

renvoyant vrai, sinon elle n'effectue aucun changement et renvoie faux. Si la valeur retournée par *ajoute* vaut vrai alors la structure *T* est mise à jour par l'ajout d'un arc entre F_i^1 et le fils f_i considéré (lignes 11 – 12). La seconde étape est similaire et réalise les mêmes opérations pour les frères de F_i^1 (lignes 15 – 20). La troisième étape se divise en deux phases. Dans la première phase (lignes 21-27), nous mettons à jour la liste des *FrèresAncêtres* du nœud F_i^1, en ajoutant les frères fréquents du ou des pères de F_i^1. En effet, un nœud peut avoir plusieurs pères, s'il apparaît plusieurs fois dans l'arbre. Il peut même avoir comme frère lui-même. Chaque nœud ainsi ajouté l'est aussi dans la liste contenant les frères des Ancêtres Locaux. Dans la seconde phase (lignes 28-33), nous ajoutons à la liste des *FrèresAncêtres* de F_i^1 si nécessaire les éléments contenus dans la liste des *FrèresAncêtres Locaux* de son ou de ses pères. Nous traitons ainsi tous les nœuds appartenant à F_i^1 (les 1-fréquents) pour chaque arbre t_i de *DB*. Ainsi, à la ligne 35, toutes les structures candidates de taille 2 sont stockées dans *T*, il ne reste plus qu'à supprimer de *T*, celles dont la valeur de support est inférieure au support minimal par un appel à *verifyCandidate(T,DB,minSupp)*..

Algorithm extensionPossible

Input : l'arbre des candidats ***T***, 1-fréquents, la base de donnée *DB*, une valeur de support minimal *minSupp*.
Output : ***T*** étendu à la profondeur 2.

```
1 :  ForEach F_i^1 ∈ T^1 do
2 :      F_i^1.listeFils ← ∅;
3 :      F_i^1.listeFrères ← ∅;
4 :      F_i^1.listeFrèresAncêtres ← ∅;
5 :  enddo
6 :  ForEach Tree t_j ∈ DB do
7 :      ForEach F_i^1 ∈ t_i do
8 :          F_i^1.listeFrèresAncêtresLocaux ← ∅;
9 :          ForEach fils f_i ∈ T^1 of F_i^1 in t_i do
10 :             If (F_i^1.listeFils).ajoute(f_i) then
11 :                 créerNoeud(f_i) ;
12 :                 ajouterArc(F_i^1,f_i) ;
13 :             endif
14 :         enddo
15 :         ForEach frère f_i ∈ T^1 of F_i^1 in t_i do
16 :             If (F_i^1.listeFrères).ajoute(f_i) then
17 :                 créerNoeud(f_i) ;
18 :                 ajouterArc(F_i^1,f_i) ;
19 :             endif
20 :         enddo
21 :         ForEach frère f_i ∈ T^1 of père(s)(F_i^1) in t_i do
22 :             If (F_i^1.listeFrèresAncêtres).ajoute(f_i) then
23 :                 créerNoeud(f_i) ;
24 :                 ajouterArc(F_i^1,f_i)
25 :             endif
26 :             (F_i^1.listeFrèresAncêtresLocaux).ajoute(f_i) ;
27 :         enddo
28 :         ForEach f_i ∈ T^1 of (père(s)(F_i^1)).listeFrèresAncêtresLocaux do
29 :             If (F_i^1.listeFrèresAncêtres).ajoute(f_i) then
30 :                 créerNoeud(f_i) ;
31 :                 ajouterArc(F_i^1,f_i) ;
32 :             endif
33 :         enddo
34 :     enddo
35 : enddo
35 : verifyCandidate(T, DB, minSup) ;
```

Algorithme 4 – L'algorithme extensionPossible de PSPtree

Propriété 8 :

A l'issue de l'algorithme *extensionPossible*, la structure *T* possède l'ensemble de toutes les extensions possibles pour les nœuds fréquents de niveau 1.

Démonstration :
La propriété précédente est basée sur la conservation de l'invariant suivant : « A la fin de la boucle (lignes 7 – 32), nous avons l'ensemble des extensions possibles du nœud F_i^1 (ligne 7) de l'arbre t_j pour l'ensemble des arbres t_j de *DB* (pour *j* allant de 1 à *i*) ». Il existe uniquement trois types d'extensions possibles pour un nœud. En effet, un nœud peut être étendu par ses fils, par ses frères et par les frères de ses ancêtres (noté ancêtres par la suite). Il n'existe aucune autre extension possible, et ces trois extensions sont prises en compte dans notre algorithme.
A la fin de la boucle (lignes 9 – 14), nous avons ajouté à F_i^1.*listeFils*, l'ensemble de ses fils comme étant des extensions possibles de type fils. A la fin de la boucle (lignes 15 – 20), nous avons ajouté à F_i^1.*listeFrères*, l'ensemble de ses frères comme étant des extensions possibles de type frères.
A la fin de la boucle (lignes 21 – 27), nous avons ajouté à F_i^1.*listeAncêtres*, l'ensemble des frères de son père comme étant des extensions possibles de type ancêtres. Nous ajoutons lors de cette boucle les frères du père de F_i^1 comme ancêtres locaux de F_i^1 (ligne 26).
A la fin de la boucle (lignes 28 – 33), nous avons ajouté à F_i^1.*listeAncêtres*, l'ensemble des ancêtres locaux de son père comme étant des extensions possibles de type ancêtres. Ceux-ci sont de plus ajoutés en tant que frères d'ancêtres locaux de F_i^1, ce qui assure la récursivité et permet d'obtenir pour chaque nœud l'ensemble des frères de tous ses ancêtres.
Ainsi nous avons à la fin de la boucle (lignes 7 – 32), pour chaque nœud, l'ensemble des extensions possibles de ce nœud pour l'ensemble des arbres t_j (pour *j* allant de 1 à *j*). L'algorithme traite tous les arbres de 1 à *n* (*n* étant le nombre d'arbres de la forêt considérée). Nous avons donc après le traitement de l'arbre *n*, l'ensemble des extensions possibles pour tous les arbres de 1 à *j* avec *j* égal à *n* ; donc nous avons bien l'ensemble des extensions possibles pour l'ensemble de notre forêt.

Complexité :
Nous notons *N*, le nombre total de nœuds dans l'ensemble des transactions de la base *DB*. Nous avons *N = somme de i = 1 à n* (nombre de nœuds de l'arbre t_i). Le premier nœud peut avoir jusqu'à *N-1* successeurs, le second *N-2* etc. Donc, cet algorithme est dans le pire des cas en *O (somme de i = 1 à n (N-i)) (= O(N(N+1)/2))*, ce qui peut être majoré par $O(N^2)$.

Exemple 38 :

Considérons l'arbre préfixé de la Figure 52. A partir du nœud ($\oplus Personne_1$), nous pouvons générer <($\oplus Personne_1$) ($\oplus Identite_2$)> car le nœud $\oplus Identite_2$ a une profondeur supérieure de 1 au nœud $\oplus Personne_1$. Par contre, si nous considérons le nœud $\perp Numéro_4$, il peut être étendu par le noeud $\perp CodePostal_4$. Ne sachant pas encore quels sont leurs parents dans la base, ces deux éléments peuvent être dans le même élément ou dans deux éléments séparés. Les candidats deviennent alors : <($\perp Numéro_4$)($\perp CodePostal_4$)> et <($\perp Numéro_4$ $\perp CodePostal_4$)>. En effet, les deux éléments sont des nœuds terminaux dans les arbres associés (Id($\perp Numéro_4$) = $\perp$ et Id ($\perp CodePostal_4$)= $\perp$).

K >2 : Dans le cas où la profondeur est supérieure à 2, seul un nœud feuille situé sur un chemin issu de la racine de *T* et contenant un élément de profondeur 1 peut être étendu. Cet élément particulier de *T* de profondeur 1 représente la racine commune à tous les arbres de la forêt (représentée par *DB*). Cette opération est réalisée par l'intermédiaire de l'algorithme *candidateGeneration*. Pour chaque nœud n_i vérifiant ces conditions, l'algorithme recherche le fils de la racine de *T* correspondant à n_i, noté *x*. Les descendants de *x* dans *T* représentent les extensions possibles de ce nœud calculées lors de l'appel à *extensionPossible*. L'algorithme étend le nœud feuille n_i en construisant pour cette feuille une copie des fils de *x*. Cette copie au vu de la propriété 8, représente les extensions possibles de n_i. Les nouvelles séquences constituées par ces nœuds étendus représentent les structures candidates de niveau *k*.

Exemple 39 :

Considérons l'arbre préfixé de la Figure 53. $\oplus Personne_1$ est le nœud racine qui peut être étendu. La feuille de la branche issue de ce nœud est : $\oplus Identite_2$. A partir de la racine, le fils de ce dernier est : $\otimes Adresse_3$. Nous savons donc que <($\oplus Identite_2$)($\otimes Adresse_3$)> intervient suffisamment fréquemment dans la base de données et

nous pouvons donc générer deux candidats en recopiant les fils de ⊕Identite₂. Le premier candidat généré est <(Personne₁) (⊕Identite₂) (⊥Nom₃)>. Le second candidat est donc <(⊕Personne₁) (⊕Identite₂) (⊗Adresse₃)>. Considérons que ces deux candidats soient fréquents sur la base de données, nous pouvons donc les étendre. Dans le cas du premier fréquent ainsi généré, l'élément ⊥Nom₃ n'a aucune extension possible, nous en avons donc terminé avec celui ci. Dans le cas du deuxième fréquent, à partir de la racine, les fils de ⊗Adresse₃ sont ⊥Numéro₄, ⊥CodePostal₄ et ⊥Nom₃. Après ajout, les candidats à tester sur la base de données deviennent <(⊕Personne₁) (⊕Identite₂) (⊗Adresse₃) (⊥Numéro₄) >, <(⊕Personne₁) (⊕Identite₂) (⊗Adresse₃) (⊥CodePostal₄) > et <(⊕Personne₁) (⊕Identite₂) (⊗Adresse₃) (⊥Nom₃) > . Considérons que ces trois candidats soient fréquents. Etant donné qu'à partir de la racine, le nœud ⊥Nom₃ n'a pas de fils il ne sera pas étendu plus en avant. Par contre de nouveaux candidats peuvent être générés en étendant le nœud ⊥Numéro₄, donnant ainsi les nouveaux candidats : <(⊕Personne₁) (⊕Identite₂) (⊗Adresse₃) (⊥Numéro₄ ⊥CodePostal₄)> et <(⊕Personne₁) (⊕Identite₂) (⊗Adresse₃) (⊥Numéro₄) (⊥Nom₃) > .

Algorithm candidateGeneration

Input : l'arbre des candidats ***T***, de profondeur k ($k \geq 2$) représentant les h-fréquents ($h \in [1..k]$).
Output : ***T*** étendu à la profondeur $k+1$, contenant les candidats de longueur $k+1$ à tester.

```
1 :  ForEach F_i^k ∈ T^k do
2 :      ForEach f_i = F_i^1.fils ∈ T^1 do
3 :          créérNoeud(G);
4 :          ajouterArc(F_i^k, G);
5 :      enddo
6 :      ForEach f_i = F_i^1.frères ∈ T^1 do
7 :          créérNoeud(G);
8 :          ajouterArc(F_i^k, G);
9 :      enddo
10 :     ForEach f_i = F_i^1.ancêtres ∈ T^1 do
11 :         créérNoeud(G);
12 :         ajouterArc(F_i^k, G);
13 :     enddo
14 : enddo
```

Algorithme 5 – L'algorithme candidateGeneration de PSPtree

Dans cet algorithme, nous considérons la structure T au niveau k. L'objectif de l'algorithme est d'obtenir la structure T avec les candidats au niveau $k + 1$. Le principe de l'extension est très simple, il consiste en un parcours des différents éléments de niveau k de la structure (ligne 1). Pour chaque élément de T de niveau k considéré, nous recherchons dans T les extensions possibles calculées par l'algorithme précédent et stockées dans T au niveau 2. Nous considérons tour à tour les extensions possibles de ce nœud par ses fils (lignes 2-5), ses frères (lignes 6 – 9) et ses ancêtres. Pour chacune de ces extensions, un nouveau nœud (lignes 3 – 7 – 11) est créé et il est rattaché à la structure T par la création d'un arc entre le nœud à étendre et celui-ci (lignes 4 – 8 – 12).

Propriété 9 :

Après exécution de *candidateGeneration*, tous les candidats de niveau $k+1$ ont été générés dans T.

Démonstration :
Il est trivial de dire qu'une utilisation de l'algorithme *candidateGeneration* pour une profondeur k ($k \geq 2$) génère bien tous les candidats possibles de niveau $k+1$. En effet, la génération de ces candidats s'effectue à partir d'extensions possibles calculées et stockées dans T par l'algorithme *extensionPossible*. Nous avons démontré précédemment que ce dernier génère bien toutes les extensions possibles pour les nœuds de *DB*. Par conséquent, l'application des règles d'extensions définies par celui-ci et du principe énoncé précédement conduisent à la génération de tout l'espace des candidats possibles.

Complexité :
La complexité d'un tel algorithme est linéaire et dépend exclusivement du nombre de nœuds N à étendre.

3.1.3 Vérification des candidats

Dans cette section, nous présentons comment les candidats stockés dans l'arbre sont vérifiés par rapport aux séquences contenues dans la base de données.
Le principe général de l'algorithme *verifyCandidate* consiste à parcourir la base de données séquence par séquence. Pour effectuer ces vérifications, chaque séquence candidate est comparée à chacune des séquences de la base à l'aide de la fonction récursive *verifySequence*. Si cette séquence candidate est incluse dans la séquence courante alors son support est augmenté de un. Une fois que le support de ces séquences est à jour, *verifyCandidate* supprime de *T* les séquences candidates non fréquentes.

Algorithm verifyCandidate

Input : L'arbre des candidats *T*. La base de données *DB*. Le support minimal *minSupp*.
Output : L'arbre *T* contenant seulement les candidats fréquents.

```
1 :   ForEach Seq_i ∈ DB do
2 :       ForEach seqCand_j ∈ T do
3 :           posPossible.clear();
4 :           posPossible.Ajoute(1) ;
5 :           teteTemp = seqCand_j.Tete ;
6 :           seqCand_j.SupprimerTete ;
7 :           supportValide = true;
8 :           If seqCand.size() > 0 then
9 :               verifySequence(seqCand_j, Seq_i,posPossible, teteTemp);
10 :          endif
11 :          If supportValide then seqCand_j.Support++; endif
12 :      enddo
13 : enddo
14 :  ForEach seqCand_j ∈ T do
15 :      If seqCand_j.support < minSupp then
16 :          T.supprimer(seqCand_j.dernier());
17 :      endif
18 :  enddo
```

Algorithme 6 – L'algorithme verifyCandidate

L'algorithme *verifyCandidate* parcours successivement chacune des séquences de *DB* (ligne 1). Pour chaque séquence de *DB* considérée Seq_i (ligne 2), l'objectif de l'algorithme est de déterminer quelles sont les séquences candidates incluses dans cette séquence. Pour cela, nous comparons tour à tour (ligne 2 – 12) chaque séquence candidate $seqCand_j$ avec la séquence Seq_i. Afin de comparer ces deux séquences entre elles, nous initialisons la liste *posPossible* et nous lui ajoutons la valeur un (lignes 3 – 4). Cela signifie que Seq_i contient le premier élément de $seqCand_j$ à la position un. En effet, comme nous l'avons vu précédemment, ces arbres possèdent une racine commune. Par la suite, nous supprimons de $seqCand_j$ le premier élément après l'avoir stocké dans une variable temporaire *teteTemp* (lignes 5 – 6). Nous mettons la variable booléenne *supportValide* à vrai. Cela signifie qu'actuellement il existe une position possible pour le dernier élément testé de $seqCand_j$ dans Seq_i. Nous faisons ensuite appel à la procédure récursive *verifySequence* (ligne 9) sous réserve que $seqCand_j$ contienne au moins un élément. Après exécution de cette procédure, *supportValide* permet de déterminer si $seqCand_j$ est incluse dans Seq_i. Si c'est le cas nous augmentons le support de $seqCand_j$ (ligne 11). La dernière étape de l'algorithme *verifyCandidate* (lignes 14 – 18) consiste à supprimer de *T*, les séquences candidates non fréquentes. Une séquence candidate de taille *n* est construite par extension d'une séquence fréquente de taille *n-1*. Chaque élément d'une séquence est représenté par un nœud dans l'arbre. Pour supprimer une séquence candidate, il suffit donc d'enlever le nœud de *T* correspondant au dernier élément de cette séquence (ligne 16).
L'algorithme *verifySequence* constitue le cœur de la procédure v*erifyCandidate*. Lors de l'appel à cette procédure, la variable booléenne *supportValide* vaut vrai. Cela signifie que la séquence constituée successivement par les éléments supprimés, notée *seqSup*, de *seqCand* (ligne 27 de *verifySequence* et ligne 6 de *verifyCandidate*) est incluse dans *Seq*. La liste *posPossible* contient toutes les positions possibles du dernier élément de *seqSup* (la séquence constituée par les éléments supprimés de *seqCand)* dans *Seq*. Le nombre de ces positions représente les différentes manières pour lesquelles la structure *seqSup* est incluse dans *Seq*.
L'objectif de chaque appel récursif est de déterminer pour chacune de ces positions possibles (ligne 2 – 25), les différentes positions pour lesquelles la séquence constituée de *seqSup* et du premier élément de *seqCand* est

incluse dans *Seq*. Pour cela nous devons déterminer la relation de parenté au sens de *extensionPossible* existante entre le premier élément de *seqCand* et le dernier élément de *seqSup* stockés dans t*eteTemp*. Cette information s'obtient en comparant la différence de profondeur entre ces deux éléments. Si celle-ci est de 0 alors ils sont frères (lignes 3-9). Si elle est de 1 alors celui issu de *seqCand* est fils de celui contenu dans *teteTemp* (lignes 10–16). Enfin si la différence est supérieure à 1 alors la relation qui les relie est celle d'ancêtre (lignes 17–24). Il suffit alors de stocker dans *posRappel* la (les) position(s) des éléments de *Seq* répondant à cette contrainte de parenté. Prenons le cas où la relation de parenté est frère. Dans un premier temps nous récupérons tous les frères du nœud situé à la position *p* de *Seq* (ligne 4). Ensuite nous examinons si un de ces frères est identique au premier élément de *seqCand* (lignes 5 - 9). Si cela est le cas alors la séquence constituée de *seqSup* et de *seqCand.Tête* est incluse dans *Seq* à la position de ce frère. Nous ajoutons donc cette position à *posRappel* (ligne 7). La méthode est identique pour les autres liens de parenté. Une fois la liste *posRappel* à jour au vu des différentes positions possibles, nous pouvons supprimer le premier élément de *seqCand* (ligne 27) après l'avoir stocké préalablement dans *teteTemp* (ligne 26). Si la liste *posRappel* n'est pas vide (ligne 28), nous devons vérifier que *seqCand* contient au moins un élément (ligne 29), si cela est le cas alors nous rappelons *verifySequence* (ligne 30). Si *posRappel* est vide alors la variable booléenne *supportValide* est mise à faux (ligne 32) signifiant que l'inclusion entre *seqCand* et *Seq* n'est pas possible.

Algorithm verifySequence

Input : Une séquence candidate *seqCand*, une séquence de *DB* notée *Seq*, la liste des positions possibles *PosPossible*, le dernier élément supprimé de *seqCand* noté *teteTemp*.
Output : *supportValide* vaut vrai si *seqCand* est inclus dans *Seq*, faux sinon.

```
1 :  posRappel.clear();
2 :  ForEach p ∈ posPossible do
3 :      If ((seqCand.Tete).Profondeur == teteTemp.Profondeur) then
4 :          listeFrere = Frere(Seq, p);
5 :          ForEach F_i ∈ listeFrere do
6 :              If F_i = seqCand.Tete then
7 :                  posRappel.Ajoute(Position(F_i, Seq));
8 :              endif
9 :          enddo
10 :     ElseIf ((seqCand.Tete).Profondeur == teteTemp.Profondeur + 1) then
11 :         listeFils = Fils(Seq, p);
12 :         ForEach F_i ∈ listeFils do
13 :             If F_i = seqCand.Tete then
14 :                 posRappel.Ajoute(Position(F_i, Seq));
15 :             endif
16:          enddo
17 :     Else
18 :         listeAncetre = Ancetre(Seq, p);
19 :         ForEach F_i ∈ listeAncetre do
20 :             If F_i = seqCand.Tete then
21 :                 posRappel.Ajoute(Position(F_i, Seq));
22 :             endif
23:          enddo
24 :     endif
25 : enddo
26 : teteTemp = seqCand.Tete;
27 : seqCand.SupprimerTete ;
28 : If posRappel.Size() > 0 then
29 :    If seqCand.Size() > 0 then
30 :       verifySequence(seqCand, Seq,posPossible, teteTemp);
31 :    endif
32 : Else supportValide = false;
33 : endif
```

Algorithme 7 – L'algorithme verifySequence

Propriété 10 :

Après l'exécution de *verifyCandidat*, le support de chaque séquence candidate de *T* correspond au nombre de séquences contenues dans *DB* pour lesquelles *seqCand* est incluse.

Démonstration :
Dans la boucle (lignes 1 – 13) chaque séquence de *DB*, notée Seq_i est comparée à chaque séquence candidate, notée $seqCand_j$, (lignes 2 – 12). Si une séquence $seqCand_j$ est incluse dans une séquence Seq_i alors son support est augmenté de 1 (ligne 11) sinon son support reste inchangé. La procédure récursive est basée sur l'invariant suivant : « si *supportValide* vaut vrai alors la séquence constituée successivement par les éléments supprimés, notée *seqSup*, de *seqCand* est incluse dans *Seq* sinon elle ne l'est pas». Au premier appel de *verifySequence* dans *verifyCandidat* (ligne 9), il est trivial de dire que cet invariant est respecté. Dans *verifySequence*, nous nous intéressons au comportement de *posRappel*. Cette liste est initialement mise à 0 (ligne 1). Après exécution des lignes 1 – 27 de *verifySequence*, comme explicité dans la description de l'algorithme, *posRappel* contient la position de tous les éléments de *Seq* tels que *seqSup* est incluse dans *Seq*. Ces positions représentent, en fait, les différentes positions possibles du dernier élément de *seqSup* dans *Seq*. Ce dernier élément était le premier de la séquence candidate *seqCand* dont nous cherchons à déterminer l'inclusion. Plusieurs cas se présentent alors : soit la taille de *posRappel* est nulle. Dans ce cas la séquence définie comme étant *SeqSup* n'est pas incluse dans *Seq*. *supportValide* est mis à faux (ligne 32). Or *seqSup* constitue les *n +1* premiers éléments de *seqCand*, où *n* est le nombre d'appels récursifs à *verifySequence*. Donc *seqCand* n'est pas inclus dans *Seq*, l'invariant est bien respecté. Soit la taille de *posRappel* est supérieure à 0, dans ce cas nous nous intéressons à la taille de *SeqCand*. Si celle-ci est nulle, cela signifie, par construction, que nous avons *seqSup = seqCandInitiale* (avant les suppressions). La valeur de *supportValide* permet de statuer sur l'inclusion entre *seqCand* et *Seq*. Sinon l'invariant est bien respecté et l'algorithme récure sur *verifySequence*. Cette récursivité s'arrête bien car à chaque passage de *verifySequence* un élément est supprimé à *seqCand* (ligne 27), donc le nombre d'éléments de celle-ci tend vers 0.

Complexité :
Soit *N*, le nombre de structures de *DB*. Soit *Z*, le nombre d'éléments de la structure maximale de la forêt. Soit *C* le nombre de candidats de *T* à vérifier. La boucle (lignes 1 à 13) a pour complexité *M * N ** la complexité de l'appel à *verifySequence*. *M* représente le nombre de nœuds de la séquence $seqCand_j$ considérée. Quant à la complexité, de la boucle lignes 14 à 18, elle est linéaire en le nombre de candidats *C*. La complexité de *verifyCandidate* est donc en *O (M*N* la complexité de verifySequence)*. Le coût de la seconde boucle est négligeable. Pour affiner cette complexité, nous calculons la complexité de *verifySequence*. Cet algorithme se divise en deux étapes. Dans la boucle de la ligne 1 à 25, pour une position possible nous pouvons majorer le nombre de fils, frères et ancêtres ajoutés à *posRappel* par *M*. Soit *K* le cardinal de *posPossible*. La complexité de cette boucle peut donc être majorée par *K * N*. La deuxième étape de l'algorithme consiste à rappeler récursivement *verifySequence* sur la séquence candidate diminuée d'un élément. Soit *L* la taille initiale de cette séquence. Dans le pire des cas nous ferons *L* appels récursifs. Donc la complexité de cet algorithme est de *K * M * L*. Nous pouvons borner *L* et *M* par Z. Nous obtenons donc une complexité pour *verifySequence* majorée par $M * Z^2$. Au final, la procédure *verifyCandidate* est donc majorée par $O (N * M^2 * Z^2)$.

Exemple 40 :

Considérons la séquence $S_1 = <(\oplus Personne_1)\ (\oplus identité_2)\ (\otimes adresse_3)\ (\perp X_4)\ (\perp Y_4)\ (\otimes Z_3)\ (\perp numéro_4)\ (\perp codepostal_4)>$ associée à l'arbre de la Figure 55. Considérons la Figure 56 qui représente l'arbre des candidats. A partir de l'élément $(\oplus Personne_1)$, nous pouvons retrouver sa correspondance dans l'arbre. Nous descendons donc d'un niveau dans l'arbre et nous nous déplaçons parallèlement dans la séquence sur le fils de $(\oplus Personne_1)$, i.e. $(\oplus Identite_2)$. De la même manière nous pouvons aller jusqu'à $(\otimes adresse_3)$. N'ayant pas dans cette séquence de fils de cet élément inclus dans l'arbre des candidats, nous passons au sommet suivant $(\otimes Z_3)$, i.e. le frère de l'élément. Etant donné qu'il n'existe pas dans l'arbre en tant que fils d'$\oplus Identite_2$, de sommet $(\otimes Z_3)$, la recherche s'arrête et ne prend pas en considération les sommets $(\perp numéro_4)$ et $(\perp codepostal_4)$.

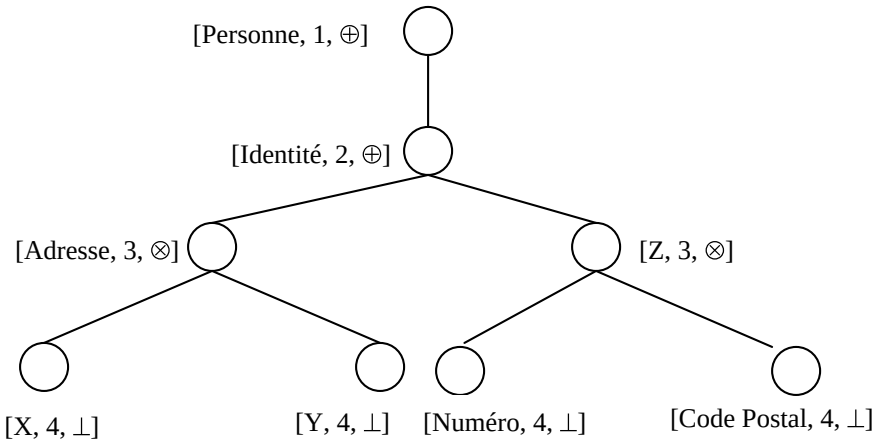


Figure 55 – Un arbre à vérifier

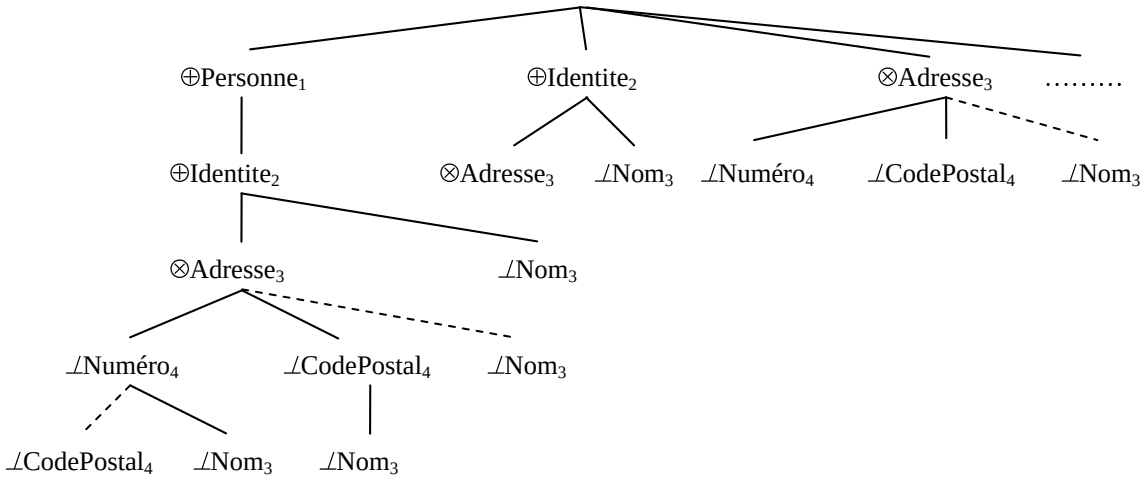


Figure 56 – Un arbre des candidats

3.1.4 PSP_{tree} : un algorithme pour l'extraction de sous arbres fréquents

A partir des algorithmes précédents, nous pouvons donc définir l'algorithme PSP_{tree} qui est basé sur la méthode *générer-élaguer* et qui procède donc par passes successives sur la base de données en alternant génération des candidats (*generateCandidate*), vérification de leurs supports et suppression des candidats non fréquents (*verifyCandidate*).

Soit *P* le nombre d'extensions possibles. Dans le pire des cas, nous devons étendre le nœud racine *P* fois. Ainsi la fonction *generateCandidate* est appliquée au pire des cas $N^2 * Z$ fois, de même pour la fonction *verifyCandidate* qui a un coût majoré de $N * M^2 * Z^2$. La complexité de l'algorithme est donc en $O(P^2 * Z * N * M^2 * Z^2)$.

Algorithm PSP$_{tree}$

Input : Un support minimal (*minSupp*) et une base de données *DB*
Output : L'ensemble L^{DB} des éléments ayant une fréquence d'apparitions supérieure à *minSupp*.

1 : $k = 1$; les éléments fréquents
2 : $C_1 = \{<i>/ i \in$ *ensemble des éléments fréquents de DB}* ;
3 : $T = C_1$;
4 : *extensionPossible(T, C_1,DB,minSupp);* calcul de C_2
5 : **If** *T is updated by extensionPossible*
6: **then** $C_2 = T$;
7 : **else** $C_2 = \varnothing$;
8 : **endif**
9 : **If** $C_2 \neq \varnothing$ **then** *generateCandidate(T)* **endif**; calcul de C_3
10 : **If** *T is updated by generateCandidate*
11: **then** $C_3 = T$;
12 : **else** $C_3 = \varnothing$;
13 : **endif**
14 : **while** $C_k \neq \varnothing$ **do**
15: *verifyCandidate(T,DB,minSupp)* ;
16 : $L^k = \{c \in C_k / c \in T\}$; seuls les candidats fréquents sont encore dans T;
17: $L^{DB} \leftarrow L^{DB} \cup L^k$;
18 : $k = k+1$;
19 : generateCandidate*(T);* calcul de C_k
20 : **If** *T is updated by generateCandidate*
21 : **then** $C_k = T$;
22 : **else** $C_k = \varnothing$;
23 : **endif**
24 : **enddo**
25 : **Return** L^{DB}

Algorithme 8 – L'algorithme PSP$_{tree}$

4 Recherche de sous arbres fréquents généralisés

Dans cette section, nous étendons la problématique précédente en relaxant les contraintes liées à la profondeur des sous arbres inclus et à la notion d'ordre. En effet, au travers des différentes expériences menées, nous avons pu constater qu'il serait intéressant de proposer également de rechercher des structures imbriquées quel que soit le niveau de profondeur de celles-ci. Désormais deux nœuds d'un arbre ayant la même étiquette seront considérés comme identiques quelle que soit leur profondeur. De plus, la notion d'ordre introduite dans le Chapitre II section 1.1 (liste de fils et ensemble de fils) n'est pas prise en compte. Les arbres considérés dans la forêt ne possèdent plus forcément une racine commune. Enfin les racines des structures recherchées peuvent avoir pour racine n'importe quelle étiquette fréquente de *DB*.

Malgré les modifications apportées à la problématique, le principe général de l'algorithme PSP$_{tree}$-GENERALISE reste proche de celui de PSP$_{tree}$ (Algorithme 8).
Dans un premier temps nous présentons un exemple afin de mieux visualiser la notion de sous arbres fréquents généralisés puis nous présentons les différentes modifications apportées aux algorithmes précédents afin de résoudre notre nouvelle problématique.

4.1 Vers une généralisation des sous arbres fréquents

Dans un premier temps, considérons les exemples de structures imbriquées du Chapitre II. En relaxant la contrainte de profondeur sur les sous arbres, la structure de la Figure 57 est maintenant incluse dans celle de la Figure 46. En effet, les nœuds ne possèdent pas la même profondeur mais respectent toujours la topologie de la structure. Par contre, la structure de la Figure 58 n'est pas imbriquée dans celle de la Figure 46 car la topologie n'est pas respectée, i.e. le lien de Parenté entre Adresse et Identité n'est pas respecté.

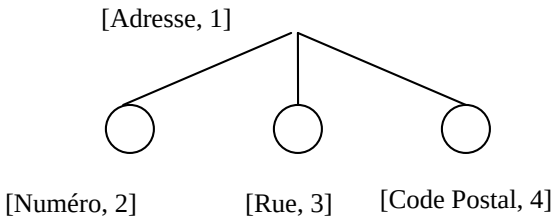


Figure 57 – Un exemple de sous arbre imbriqué pour le cas généralisé

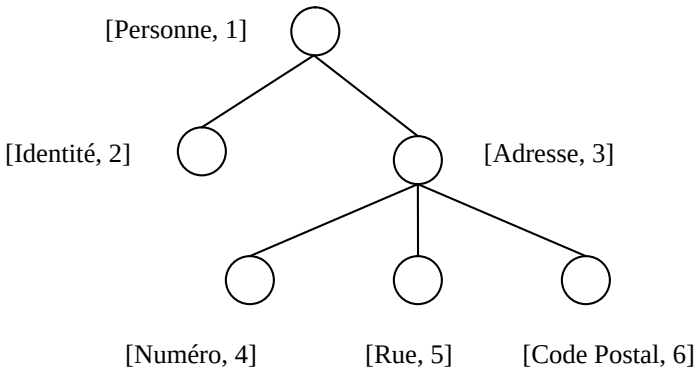


Figure 58 – Un exemple de sous arbre toujours non imbriqué

Considérons à présent la base de données *DB* de la Figure 59.

Arbre_id	Arbre
T_1	personne : {identite : {adresse : {numero : ⊥}}}
T_2	identite : <adresse : {numero : ⊥}, compagnie : ⊥>
T_3	personne : {identite : {adresse : {numero : ⊥}, compagnie : ⊥}, identite : ⊥}

Figure 59 - La base de données

Arbre_id	Arbre
T_1	($personne_1$) ($identite_2$) ($adresse_3$) ($numero_4$)
T_2	($identite_1$) ($adresse_2$) ($numero_3$) ($compagnie_4$)
T_3	($personne_1$) ($identite_2$) ($adresse_3$) ($numero_4$) ($compagnie_3$) ($identite_2$)

Figure 60 – La base de données après application de TreeToSequenceGeneralise

Comme pour la problématique précédente, cette base de données est transformée en une base de séquences par application de la procédure *TreeToSequenceGeneralise*. La différence entre cette procédure et la procédure *TreeToSequence* (Algorithme 1) se situe lorsque des concaténations ont lieu (lignes 3 et 5). En effet l'indicateur d'ordre, noté *I(r)*, n'est pas concaténé du fait de la relaxe de cette contrainte.

La deuxième modification par rapport à la problématique précédente se situe dans le fait que désormais deux nœuds de profondeur différentes ayant la même étiquette sont considérés comme similaires. Ainsi les éléments $identite_2$ et $identite_1$ des arbres T_1 et T_2 sont considérés comme identiques. Le fait de ne plus prendre en compte la notion de profondeur va nécessairement modifier la façon dont les structures recherchées vont être représentées et introduire la notion de profondeur relative d'un élément dans une séquence représentant un arbre.

Définition 16 :

Soit une séquence d'éléments $s_1, s_2, \ldots, s_i, s_j$ représentant un arbre. La profondeur relative de l'élément de la séquence qui représente la racine de l'arbre vaut 0. Un élément s_i de la séquence $s_1, s_2, \ldots, s_i, s_j$ a pour profondeur relative : (+1) si s_j est un fils de s_i, (+0) si s_j est un frère de s_i , (–k) si s_j est un ancêtre de s_i, où k représente la différence de profondeur entre les nœuds s_i et s_j.

Exemple 41 :

La séquence < $(personne_1)$ $(identite_2)$ $(adresse_3)$ $(numero_4)$ $(compagnie_3)$ $(identite_2)$ > se note désormais < $(personne_0)$ $(identite_{+1})$ $(adresse_{+1})$ $(numero_{+1})$ $(compagnie_{-1})$ $(identite_{-1})$. En effet comme $personne_1$ est la racine de l'arbre représenté par la séquence, sa profondeur relative vaut 0. Les deuxième, troisième et quatrième éléments de la séquence sont des fils de l'élément qui les précède dans la séquence donc leur profondeur relative est (+1). Les deux derniers éléments sont des ancêtres de l'élément qui les précède donc leur profondeur relative est de (-1) (k = 1 car la différence de profondeur entre les éléments vaut un).
Cette notion de profondeur relative introduit une nouvelle façon de noter les séquences en faisant abstraction de la profondeur réelle des éléments qui la composent. Les séquences < $(identite_2)$ $(adresse_3)$ $(numero_4)$ > et < $(identite_1)$ $(adresse_2)$ $(numero_3)$ > qui dans le cas de la généralisation, correspondent à des structures similaires peuvent être représentées par la séquence < $(identite_0)$ $(adresse_{+1})$ $(numero_{+1})$ >. Cette représentation sera utilisée ultérieurement pour les candidats.

La troisième modification de la problématique réside dans la relaxe de la contrainte d'unicité de la racine des arbres considérés. Nous considérons dans le cas généralisé, plusieurs forêts d'arbres. Afin de représenter ces forêt d'arbres dans une structure préfixée, nous ajoutons artificiellement une racine commune à celle-ci, notée *racine*. Dans la base de la Figure 60, nous considérerons deux forêts d'arbres, les arbres dont la racine est étiquetée par *personne* et ceux dont la racine est étiquetée par *identite.*

Enfin la dernière modification de la problématique impose le fait que toute étiquette fréquente dans un des arbres de la forêt peut être racine d'une structure typique recherchée. Pour prendre en compte cette nouvelle contrainte, nous ajouterons un élément particulier à la structure préfixée noté *super-racine*. Ce nœud est un fils de la racine commune à toutes les forêts considérées dans *DB.*

Structures fréquentes	**Arbre_id**
$(personne_0)$ $(identite_{+1})$ $(adresse_{+1})$ $(numero_{+1})$	T_1, T_3
$(identite_0)$ $(adresse_{+1})$ $(numero_{+1})$ $(compagnie_{-1})$	T_2, T_3
$(adresse_0)$ $(numero_{+1})$ $(compagnie_{+1})$	T_2, T_3
…. ….. …	… … …

Figure 61 – Quelques structures fréquentes généralisées

L'objectif de la recherche de structures fréquentes généralisées consiste donc à trouver toutes les structures communes aux structures contenues dans *DB*. La Figure 61 représente quelques structures fréquentes généralisées extraites de *DB* pour une valeur de support minimal égale à 2. La colonne de droite représente les arbres de *DB* pour lesquels la structure est incluse. Maintenant que la notion de structure généralisée a été définie nous allons présenter les différentes étapes de l'algorithme PSP_{tree}-GENERALISE en fonction de la profondeur k de la structure préfixée T construite sous l'angle de ses différences avec PSP_{tree}.

K=1 : Cette phase est similaire à celle de PSP_{tree}. Chaque branche issue de la racine relie celle-ci à une feuille qui représente un élément. Chacune de ces feuilles contient l'élément suivi de sa profondeur relative et de son support. Nous ajoutons aussi un nœud appelé super-racine qui est un fils de la racine commune à la forêt d'arbres de *DB.*

Exemple 42 :

La Figure 62 illustre l'état de la structure après évaluation du support pour chaque élément de niveau 1. Nous considérons dans cet exemple que pour être retenue une séquence doit apparaître dans au moins deux séquences de données.

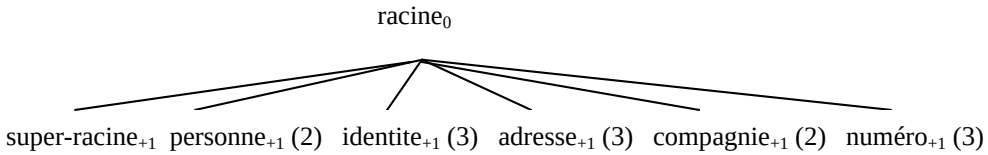


Figure 62 – La structure préfixée au niveau 1

K>1 : Tout comme dans le cas du PSP_{tree}. Chaque chemin, noté n_1, n_2, ...n_k, de la racine de l'arbre vers une feuille représente une séquence. La séquence représentée est $<(n_2)\ldots(n_k)>$ si et seulement si n_2 est différent de *super-racine*$_{+1}$, $<(n_3)\ (n_4)\ldots(n_k)>$ sinon.

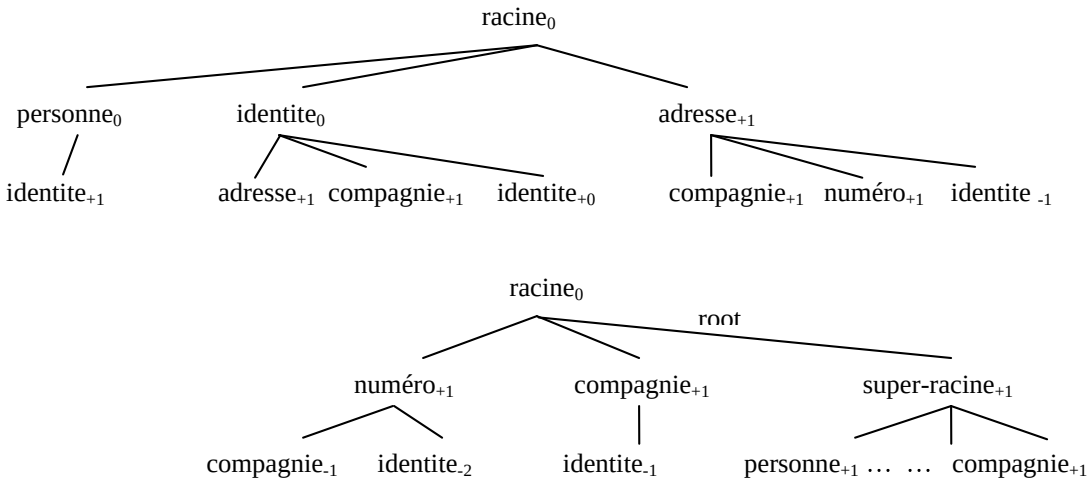


Figure 63 – La structure préfixée au niveau 2

Exemple 43 :

L'arbre de la Figure 63 représente des séquences fréquentes de taille 2. Si on parcourt par exemple le chemin racine$_0$*, identite*$_{+1}$*, compagnie*$_{+1}$*, ce chemin représente la séquence candidate suivante : <(identité*$_0$*) (compagnie*$_{+1}$*)>.Tous les éléments de niveau 1 fréquents de T à l'exception de lui même sont fils du noeud super-racine*$_{+1}$*.*

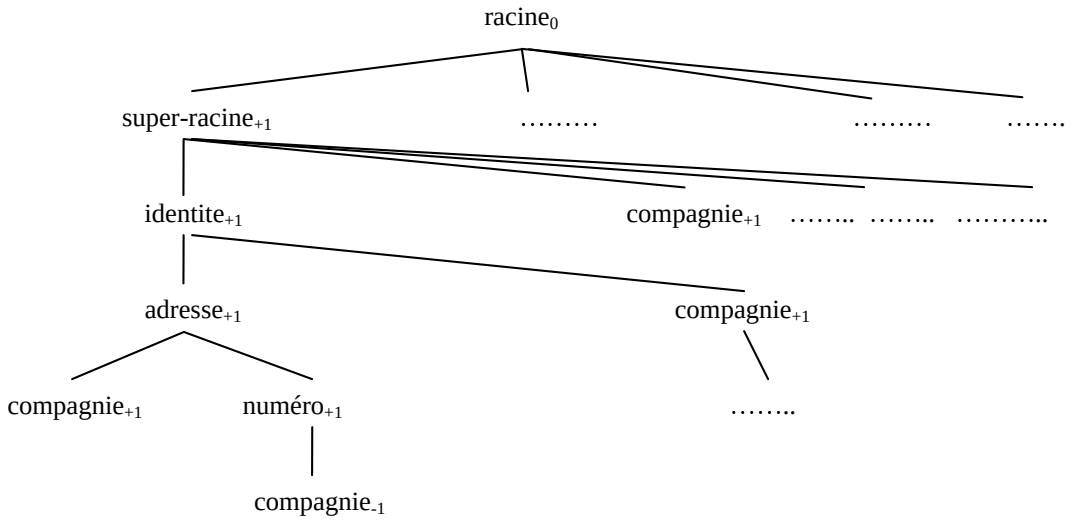


Figure 64 – La structure préfixée au niveau 5

La Figure 64 illustre une partie de la structure au niveau 5. A partir de la profondeur k = 2 seuls les fils du nœud particulier *super-racine*$_{+1}$ sont étendus. Ce qui a pour effet de déséquilibrer la structure préfixée comme pour PSP_{tree}. Le principe de l'extension est identique à celui de PSP_{tree}.

4.2 Génération des candidats

Dans cette section, nous présentons les différences existantes dans la génération de candidats généralisés par rapport à la génération classique utilisée dans $PSP_{tree.}$

K=1 : La méthode est identique sauf que nous ajoutons un élément nommé *super-racine* dont seuls les fils seront étendus lorsque *k* aura une valeur supérieure à 2.

K = 2 : Le calcul des règles d'extension utilise le même mécanisme sauf que la profondeur relative est utilisée à la place de la profondeur réelle. Cela a pour conséquence de regrouper des règles du type X_i -> Y_j et X_l -> Y_m en une seule règle X -> Y_n, dans le cas où *(j-i)= (m-l)*. A l'issue de *extensionPossibleGeneralise*, pour vérifier le support de ces candidats nous utilisons la fonction *verifyCandidateGeneralise* définie en 4.3. Il reste une étape supplémentaire à effectuer. Elle consiste à créer autant de nœuds qu'il existe d'éléments fréquents de niveau 1 et de lier ces nœuds comme étant des fils de l'élément *super-racine*$_{+1}$.

Nous ne détaillerons pas l'algorithme *extensionPossibleGeneralise*, la seule différence à prendre en compte est la notion de profondeur relative comme explicité ci-dessus. L'explication de l'algorithme est donc similaire, il en est de même pour sa démonstration. Du fait de la relaxe de certaines contraintes, le nombre d'extensions possibles généralisées est supérieur ou égal au nombre d'extensions possibles. Néanmoins la complexité dans le pire des cas est toujours majorée par $O(N^2)$.

K >2 : Dans le cas où la profondeur est supérieure à 2, seuls les fils de la super-racine des arbres peuvent être étendus. Cette opération est réalisée par l'intermédiaire de l'algorithme *candidateGenerationGeneralise* d'une manière semblable à PSP_{tree}. Toutes les structures candidates construites par *candidateGeneration* ont obligatoirement une structure d'arbre. Cependant lors de l'utilisation de la notion de profondeur relative pour la construction de candidats il est nécessaire d'effectuer une vérification. Pour cela nous appliquons l'algorithme *verifProf* sur chacun des candidats. Nous ne détaillerons pas *candidateGenerationGeneralise*, sa complexité linéaire, son explication et sa démonstration sont similaires à *candidateGeneration*. Par contre nous détaillons l'algorithme *verifProf*.

Algorithm verifProf

Input : L'arbre des candidats *T*, un nœud *N* représentant un candidat
Output : Si la séquence associée au chemin allant de la racine de *T* à *N* n'est pas un arbre alors il est élagué de *T*.

```
1 : S = Sequence(T,N) ;
2 : S.supprimer(S.premier);
3 : profCour = 0 ;
4 : ForEach s ∈ S do
5 :     profCour = profCour + s.profRelative;
6 : enddo
7 : if profCour ≤ 0 then
8 :     Supprimer(T,N) ;
9 : endif
```

Algorithme 9 – L'algorithme verifProf

Dans cet algorithme, nous considérons la structure *T* et un de ses nœuds. Le chemin depuis la racine *T* jusqu'au nœud feuille *N* représente un candidat dont on souhaite vérifier la cohérence. L'objectif de l'algorithme est de déterminer si ce candidat représente bien une structure d'arbre. Dans un premier temps, nous construisons la séquence des nœuds constituée par ce chemin dans *T* (ligne 1). Pour un chemin, noté n_1, n_2, ...n_k, la séquence obtenue est <(n_3) (n_4)…(n_k)>. Nous supprimons le premier élément de cette séquence (ligne 2). Après avoir initialisé la profondeur courante à 0 (ligne 3), nous parcourons chacun des éléments de cette séquence en ajoutant à *profCour* la profondeur relative de l'élément courant de la séquence (lignes 4 – 6). Si la profondeur

courante, après traitement de tous les éléments, est inférieure ou égale à 0 (ligne 7) alors ce candidat n'est pas un arbre donc il est supprimé de *T* (ligne 8).

Démonstration de verif¨Prof:
Soit *N* un nœud de *T* représentant un candidat construit par l'algorithme *candidateGenerationGeneralise*. Soit n_1, n_2, ...n_k, le chemin depuis le nœud racine de *T* jusqu'à *N*. La séquence candidate représentée par ce chemin est < (n_3) (n_4)… (n_k)>. Cette séquence candidate est un arbre, si la somme des profondeurs relatives des éléments qui la composent est strictement positive. La profondeur relative du premier élément de la séquence est nulle. En effet soit s_1, s_2, ..., s_n les éléments de cette séquence pour $n \geq 2$. Dans le cas où $n = 1$, la séquence est forcément un arbre. Supposons que $S' = s_1, s_2, ..., s_{n-1}$ soit une séquence représentant un arbre dont la somme des profondeurs relatives est $p \geq 0$ (hypothèse de récurrence). Si nous ajoutons l'élément s_n, au vu de *extensionPossibleGénéralisée*, il existe trois types de relations entre s_{n-1} et s_n. Soit s_n est fils de s_{n-1}, auquel cas la profondeur relative de s_n est (+1) donc la profondeur relative de $S' + s_n$ est $p+1$. Or p est positif et s_{n-1} est un arbre par hypothèse. Donc l'ajout d'un lien de parenté de type fils entre s_{n-1} et s_n aboutit bien à la construction d'un arbre dont la somme des profondeurs relatives est $p+1$. Soit s_n est un frère de s_{n-1}, auquel cas la profondeur relative de s_n est (+0). Les nœuds s_{n-1} et s_n ont donc la même profondeur. Or p est positif et S' est un arbre par hypothèse. Si p vaut 0 alors dire que $S' + s_n$ est un arbre est absurde. Si $p > 1$, l'ajout d'un lien de parenté de type frère entre s_{n-1} et s_n aboutit bien à la construction d'un arbre dont la somme des profondeurs relatives est p. Dans le dernier cas s_n est un ancêtre de s_{n-1}, la profondeur relative de s_n est (–i) avec (i > 0) donc la profondeur relative de $S' + s_n$ est $p - i$. Or p est positif et s_{n-1} est un arbre par hypothèse. Deux cas se présentent soit $i \geq p$, ce qui signifie que nous nous situons à la profondeur p et que nous essayons de construire un arc entre un nœud de cette profondeur et un nœud qui aurait une profondeur inférieure ou égale à 0 ce qui est absurde. Soit $i < p$, auquel cas s_{n-1} a une profondeur suffisante pour pouvoir créer un arc signifiant l'existence d'un ancêtre ayant une profondeur $p - i$. Dans ce cas si on ajoute un lien de parenté de type ancêtre entre s_{n-1}et s_n, $S' + s_n$ représente bien un arbre dont la profondeur relative de son dernier nœud est $p - i$. Cette démonstration justifie le choix de *profCour* (ligne 7) comme critère pour établir si ce candidat est un arbre ou non.

Complexité :
La complexité d'un tel algorithme est linéaire en la longueur de la séquence à vérifier. Cette longueur pour la profondeur *k* vaut *k-2*. Le fait que cet algorithme soit linéaire ne modifie pas la complexité dans le pire des cas de l'algorithme *candidateGenerationGeneralise.* Néanmoins, en moyenne, l'algorithme va être plus lent du fait de l'existence d'un nombre d'extensions possibles plus important dans le cas généralisé que dans le cas normal.

4.3 Vérification des candidats

La stratégie utilisée lors de cette étape est identique à PSP_{tree}. Nous allons examiner les quelques différences entre *verifyCandidateGeneralise* et *verifyCandidate,* ainsi que celles existant entre *verifySequenceGeneralise* et *verifySequence*.

verifyCandidateGeneralise et verifyCandidate :

Le seul changement qui intervient dans l'algorithme *verifyCandidateGeneralise* se situe à la ligne 4 lors de l'initialisation de *posPossible*. En effet, la relaxe des contraintes de profondeur et le fait que tout élément de la séquence fréquente puisse être racine d'une structure candidate imposent l'appel à la fonction *trouverPos*. Celle-ci renvoie toutes les positions possibles du premier élément de la séquence candidate considérée $seqCand_j$ dans la séquence Seq_i de *DB*. La variable temporaire *teteTemp* n'est plus utilisée car la valeur de la profondeur relative permet de déterminer la relation entre deux éléments de la séquence. Nous la supprimons donc et nous remplaçons l'appel à *verifySequence* par un appel à *verifySequenceGeneralisee* dans la ligne 9. Le reste du raisonnement est identique.

verifySequenceGeneralise et verifySequence :

Les modifications à effectuer sont sommaires. La relation de parenté entre le dernier élément supprimé de la séquence et la tête de celle-ci est donnée directement par la valeur de la profondeur relative de l'élément en tête (lignes 3,10). La comparaison entre la profondeur de *teteTemp* et le premier élément de *seqCand* est remplacée par une comparaison de cette profondeur relative avec *+0* pour le cas des frères et *1* pour le cas des fils. L'appel à *verifySequence* est remplacé par un appel à *verifySequenceGeneralise* dans la ligne 30.

Complexité :
Nous reprenons ici les notations utilisées dans la section 3.1.3. Il n'existe aucune différence au niveau de la complexité dans le pire des cas que ce soit pour *verifyCandidateGeneralise* (*O(M*Z²)*) ou pour *verifySequenceGeneralise* (*O(N * M² * Z²)*) par rapport à *verifyCandidate* et *verifySequence*. Seule la durée moyenne d'exécution de l'algorithme est changée. Ceci est lié à la relaxe des contraintes qui implique que la valeur du nombre de positions possibles (notée *K*), dans le cas généralisé est supérieur ou égal à celle du cas normal. De ce fait on vérifie plus de séquences en moyenne. La complexité dans le pire des cas demeure elle identique.

4.4 PSP_{tree}- GENERALISE : un algorithme pour l'extraction de sous arbres fréquents généralisés

A partir des algorithmes précédents, nous pouvons donc définir l'algorithme PSP_{tree} -GENERALISE. Cet algorithme est bien entendu très proche de celui de PSP_{tree}. Le nom des fonctions appelées de génération et de vérification des candidats diffère. L'autre modification réside dans la nécessité d'appeler la procédure *verifProf* sur chaque candidat généré pour valider le fait qu'il représente bien une structure d'arbre.

Algorithm PSP_{tree} - GENERALISE

Input : Un support minimal (*minSupp*) et une base de données *DB*
Output : L'ensemble L^{DB} des éléments ayant une fréquence d'apparitions supérieure à *minSupp*.

1 : $k = 1$; les éléments fréquents
2 : $C_1 = \{<i> / i \in$ *ensemble des éléments fréquents de DB*$\}$;
3 : $T = C_1$;
4 : *extensionPossibleGeneralise(T, C_1,DB,minSupp);* calcul de C_2
5 : **If** *T is updated by extensionPossibleGeneralise*
6: **then** $C_2 = T$;
7 : **else** $C_2 = \varnothing$;
8 : **endif**
9 : **If** $C_2 \neq \varnothing$ **then** *generateCandidateGeneralise(T)* **endif**; calcul de C_3
10 : **ForEach** $c_i \in C_3$ **do**
11 : *verifProf(c_i);*
12 : **enddo**
13 : **If** *T is updated by generateCandidateGeneralise*
14: **then** $C_3 = T$;
15 : **else** $C_3 = \varnothing$;
16 : **endif**
17 : **while** $C_k \neq \varnothing$ **do**
18: *verifyCandidateGeneralise(T,DB,minSupp)* ;
19 : $L^k = \{c \in C_k / c \in T\}$; seuls les candidats fréquents sont encore dans T;
20 : $L^{DB} \leftarrow L^{DB} \cup L^k$;
21 : $k = k+1$;
22 : *generateCandidateGeneralise(T);* calcul de C_k
23 : **ForEach** $c_i \in C_k$ **do**
24 : *verifProf(c_i);*
25 : **enddo**
26 : **If** *T is updated by generateCandidateGeneralise*
27 : **then** $C_k = T$;
28 : **else** $C_k = \varnothing$;
29 : **endif**
30 : **enddo**
31 : **Return** L^{DB}

Algorithme 10 – L'algorithme PSPtree-GENERALISE

Soit *P* le nombre d'extensions possibles. Dans le pire des cas, nous devons étendre les fils de *super-racine P* fois. *P* étant majorable pas *N*, la fonction *generateCandidate* est appliquée au pire des cas $N^2 * Z$ fois, de même pour la fonction *verifyCandidate* qui a un coût majoré de $N * M^2 * Z^2$. La complexité de l'algorithme est donc en $O(P^2 * Z * N * M^2 * Z^2)$. Elle est dans le pire des cas similaire à celle de *PSPtree*. Néanmoins, comme précédemment cet algorithme est plus couteux en temps en moyenne. L'espace des candidats générés est plus grand et le nombre de séquences à verifier en moyenne aussi.

5 Expérimentations

De manière à évaluer et valider notre approche nous avons mené différentes expérimentations que nous présentons dans cette section. Avant d'évaluer les performances des algorithmes proposés, nous nous intéressons à la validation de l'approche.

5.1 Validation de l'approche

Pour valider notre approche, nous avons utilisé plusieurs jeux de données issus d'Internet ou d'organismes avec lesquels nous effectuons des collaborations. Dans le premier cas, IMDB et Navires, la recherche de sous arbres fréquents consistait à rechercher dans des bases de données semi structurées quelles étaient les structures les plus fréquentes. Dans le second cas, les données manipulées représentaient le parcours d'utilisateurs sur des sites Web. En effet, comme nous le verrons dans le Chapitre V, il est tout à fait possible de retranscrire le parcours d'un usager sur un site Web sous la forme d'un arbre de parcours.

5.1.1 La base de données IMDB

Figure 65 – http://us.imdb.com

Figure 66 – Un exemple d'information sur un film

Support	Sous arbres fréquents
10%	* {actor:{name, birthname, dateofbirth, dateofdeath, filmographyas :{title, notabletv}}} * {actor:{name, birthname, dateofbirth, minibibliography, filmographyas :{title, notabletv}}} * {actor:{name, birthname, dateofbirth, sometimescreditas:{name}, filmographyas :{title, notabletv}}} * {actor:{name, birthname, dateofbirth ,trivia, filmographyas :{title, notabletv}}} * {actor:{name, dateofbirth, filmographyas :{title, director, notabletv, producer}}}
20%	* {actor:{name, birthname, dateofbirth, filmographyas :{title, notabletv}}} * {actor:{name, dateofbirth, dateofdeath, filmographyas :{title, notabletv}}} * {actor:{name, dateofbirth, minibibliography, filmographyas :{title, notabletv}}} * {actor:{name, dateofbirth, sometimescreditas:{name}, filmographyas :{title}}} * {actor:{name, dateofbirth, trivia, filmographyas :{title, notabletv}}} *{actor:{name, sometimescreditas:{name},filmographyas :{tit-le, notabletv}}}
30%	* {actor:{name, birthname, dateofbirth, filmographyas :{title}}} * {actor:{name, dateofbirth, filmographyas:{title,notabletv}}} * {actor:{name, trivia, filmographyas:{title}}}
40%	* {actor:{name, dateofbirth, filmographyas:{title, notabletv}}}
50%	* {actor:{name ,dateofbirth, filmographyas:{title, notabletv}}}

Figure 67 – exemple de résultats obtenus

De la même manière que [WaLi99], nous avons tout d'abord appliqué nos algorithmes sur la base de données des films sur Internet (C.f. Figure 65) afin de rechercher des structures typiques de documents concernant le cinéma. Cette base de données regroupe les informations sur les films de 1892 à nos jours. Toutes les informations sont organisées sous forme de pages HTML reliées entre elles par des liens hypertextes. La Figure 66 illustre un exemple d'informations correspondant au film « Star Wars : Episode II – Attack of the Clones (2002) ».

Suite à l'examen de quelques exemples de films, nous nous sommes intéressés à la partie de la base qui concerne les 250 meilleurs films afin d'en extraire les informations concernant les acteurs.
Après avoir récupéré les données concernant les acteurs, nous avons extrait des pages HTML, les informations significatives des pages concernant la structure des documents de type acteur. Cette étape a été réalisée en partie automatiquement à l'aide d'un parser mais également manuellement dans la mesure où le contenu des pages ne permettait pas d'extraire, sans ambiguïté, les structures sous-jacentes. Au total, nous avons récupéré une base de 500 acteurs dont la profondeur maximale était de 5.

Sur la base de données ainsi obtenue, nous avons appliqué nos algorithmes. La Figure 67 illustre les résultats obtenus pour différentes valeurs de support. Par exemple, pour un support de 50%, nous avons trouvé le sous arbre commun suivant : *{actor :{name, dateofbirth, filmographyas :{title, notabletv}}}* indiquant que pour au moins 250 acteurs de la base, un acteur possède un nom, une date de naissance et une filmographie. Dans cette filmographie, il apparaît à la fois dans un film mais il a également effectué des apparitions à la télévision.

5.1.2 La base de données Navires

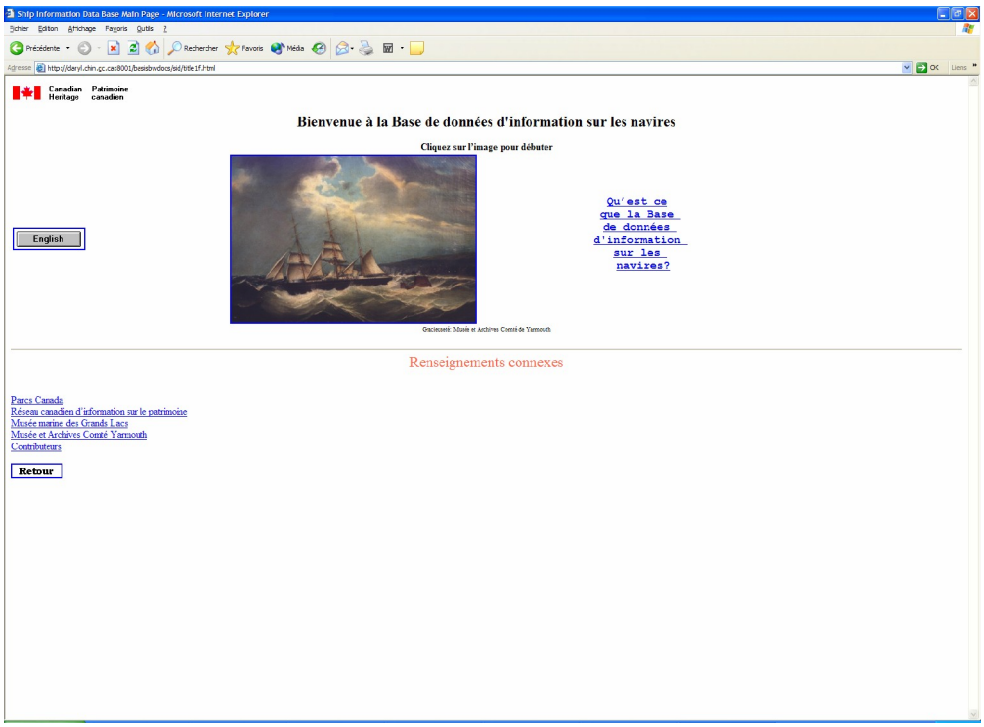


Figure 68 – http://daryl.chin.gc.ca:8081/basisbwdocs/sid/title1f.html

Suite à cette première expérimentation dans le domaine des films, nous avons mis en œuvre une application sur la recherche de sous arbres fréquents dans un ensemble de documents semi structurés correspondant à une base de données canadienne (C.f. Figure 68). Cette base a été créée pour répondre aux besoins des gestionnaires des ressources culturelles chargés de l'information sur les épaves archéologiques. Elle contient des renseignements sur des navires qui ont été immatriculés au Canada ou qui ont navigué dans les eaux canadiennes. Elle se divise en cinq sous bases : Navires, Capitaines, Propriétaires, Constructeurs navals, Voyages. Pour chacune des ces bases, des structures différentes existent. Par exemple, la base Navire contient de nombreuses informations

comme : l'identification du navire, l'identification antérieure, des informations sur le tonnage, la voilure.... De manière générale, les informations contenues dans la base ont une profondeur variant de 2 à 10 et il existe de nombreuses informations incomplètes, i.e. tous les champs ne sont pas renseignés.

Ainsi, pour un support de 85 %, nous avons trouvé 5 sous arbres fréquents.

5.1.3 Parcours d'utilisateurs sur un site Web

Cette dernière expérimentation dans l'extraction de connaissances a pour objet de mettre en relation la notion de structure et celle de parcours d'un utilisateur sur un site. En effet, on peut raisonnablement considérer que le parcours d'un utilisateur sur un site Web est assimilable à un arbre dont la racine correspond à l'entrée dans le serveur. Le jeu de données utilisé est issu des fichiers log d'un site de e-commerce spécialisé en téléphonie. Il a une taille de 400 Mo, il contient 12000 adresses IP différentes, concerne 900 pages visitées et en moyenne les arbres associés aux parcours ont une profondeur de 5.

Avec un support de 5%, nous avons trouvé le parcours fréquent suivant : <(/default.asp) (/Manage.asp) (/Paybox.asp) (/SecurePay.asp) (/SecurePayAtLeast.asp) (/RedirectSite.asp) (/PayboxDelivery.asp) (/Account.asp)> indiquant qu'au moins 600 personnes avec des IP différentes sont allées après la page d'accueil (default.asp) accéder à leur compte (Manage.asp) pour recharger leur crédit de temps (Paybox.asp, SecurePay.asp, SecurePayAtLeast.asp, RedirectSite.asp, PayboxDelivery.asp) et enfin ont vérifié sur leur compte leur nouveau crédit temps (Account.asp).

5.2 Evaluation des algorithmes

Les différents algorithmes PSP_{tree} et PSP_{tree}-GENERALISE ont été implémentés en C++ en utilisant une bibliothèque similaire à STL appelée GTL (Graph Template Library) qui gère toutes les fonctions classiques de manipulation de graphes.

Afin de tester ces algorithmes, nous avons mis en œuvre un générateur pseudo aléatoire de forêt d'arbres qui admet les paramètres suivants :

Paramètre	**Description**
Nb_tid	Le nombre d'arbres à générer dans la forêt.
Prof_max	La profondeur maximale des arbres générés.
Nb_items	Le nombre d'étiquettes différentes possibles pour un nœud de l'arbre.
Stop[i]	Indique la probabilité, pour un nœud de profondeur i d'avoir un fils de profondeur i+1.
Nb_noeud_Max[i]	Indique pour un nœud de profondeur i le nombre de fils de profondeur i+1 qu'il peut avoir.
Prob_Item[i,j]	Indique la probabilité d'une étiquette i (i allant de 1 à Nb_item) pour la profondeur j.

Figure 69 – Paramètres du générateur

De manière à évaluer nos propositions, différents types de jeux de données ont été générés. Nous avons fait varier le nombre d'étiquettes différentes maximales (Nb_item de 5 à 30) ainsi que la profondeur maximale de l'arbre (Prof_max de 3 à 9). Le tableau de la Figure 70 illustre les fichiers générés et les algorithmes appliqués pour les évaluer. Par exemple, le fichier 1N correspond à des données où le nombre d'étiquettes par nœud est de 5 au maximum, la profondeur maximale dans l'arbre est de 3 et sur lequel nous avons appliqué l'algorithme PSPtree (N pour normal). Le G correspond au cas où l'algorithme Généralisé a été utilisé.

	Prof max = 3	Prof max = 6	Prof max = 9
Etiquette (Nb_items=5)	1N, 1G		
Etiquette (Nb_items=15)	2N, 2G	5N, 5G	8N, 8G
Etiquette (Nb_items=30)	3N, 3G		

Figure 70 – Les fichiers générés

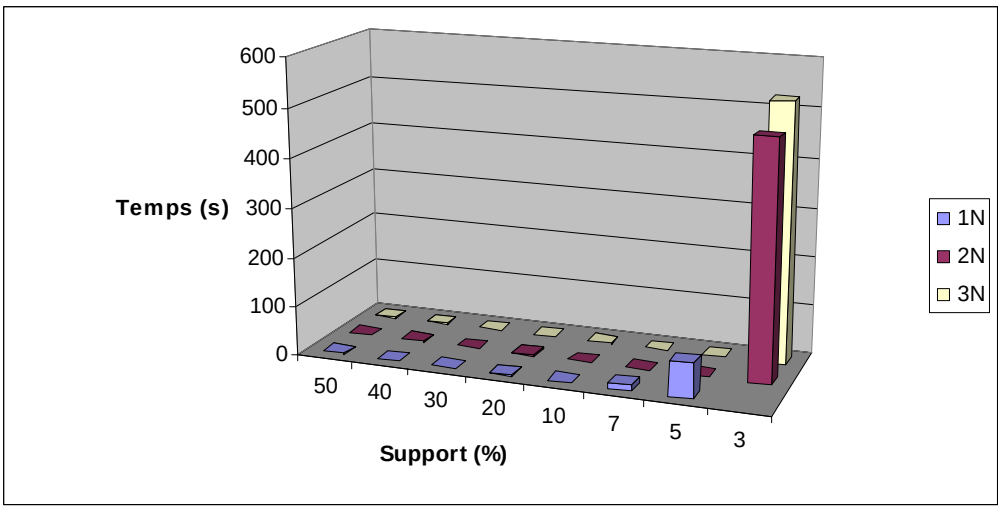
Nom	Nb_items	Profmax	Profondeur moyenne	Nombre de Nœuds max par arbre	Nombre de nœuds en moyenne par arbre
1N	5	3	2,73	57	16,73
1G	5	3	2,91	61	18,72
2N	15	3	2,89	57	19,7
2G	15	3	2,8	59	18,84
3N	30	3	2,82	73	20,14
3G	30	3	2,83	60	69,08
5N	15	6	5,63	1044	367,79
5G	15	6	5,34	1544	351,21
8N	15	9	8,24	12510	419,58
8G	15	9	8,54	12882	517,17

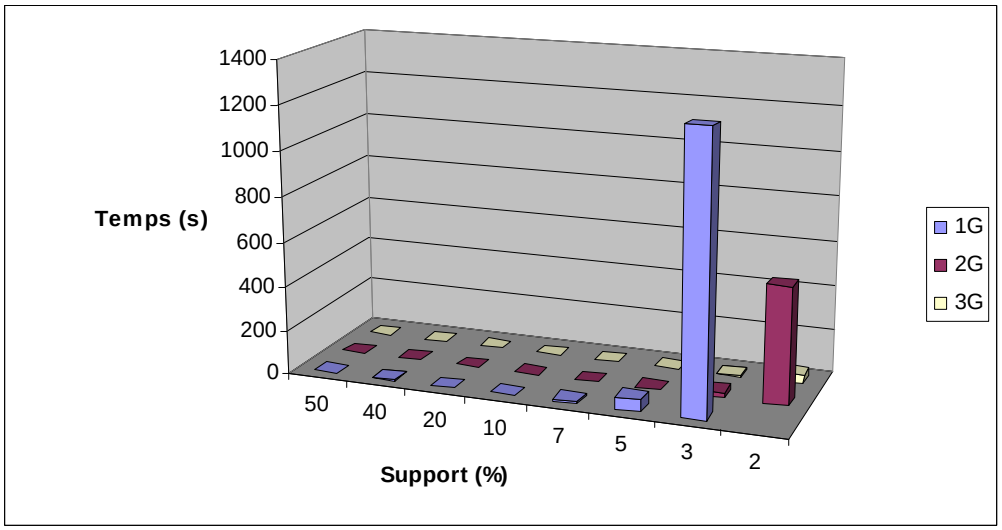
Figure 71 – Caractéristiques des fichiers générés

Dans un premier temps, nous avons mené des expériences de manière à examiner les temps de réponse des algorithmes. L'idée générale était de montrer que notre approche possède le même type de comportement que les algorithmes classiques de règles d'association ou d'extraction de motifs. Nous souhaitions bien entendu étudier également l'impact des différents paramètres, tels que la profondeur et le nombre d'items moyen par nœud, sur les performances de nos algorithmes. Enfin, de manière à voir comment se comportaient nos algorithmes en fonction du nombre de sous arbres, nous avons fait varier ce nombre pour une même valeur de profondeur. Nous décrivons par la suite les résultats obtenus pour ces expériences.

La Figure 72 illustre les résultats obtenus sur les différents jeux de données en considérant les temps de réponse. Nous ne donnons pas les résultats pour les bases 2G, 5G et 8G car ils sont identiques au cas normal. L'axe des *x* correspond à une variation de support, l'axe des *y* représente les temps de réponse en seconde et l'axe des *z* correspond aux différents jeux de données. Comme attendu, nous constatons que, de manière générale, le coût des algorithmes dépend fortement de la valeur du support : plus le support diminue, plus le nombre de fréquents augmente et nécessite donc plus de calcul. De manière à étudier l'impact du nombre de nœuds, i.e. le comportement des algorithmes lors de graphes développés en largeur, nous avons fait varier la taille du nombre moyen d'items par nœud. Nous pouvons constater entre la figure du haut et la figure du bas que les parcours en largeur pénalisent fortement les algorithmes. Ce constat était attendu dans la mesure où, dans ce cas, un plus grand nombre de candidats est testé et donc les temps de calculs sont étendus. Il faut toutefois constater que dans le cas du jeu de données 8N les temps de calculs sont globalement les mêmes quelque soit le support spécifié. En fait, cela est dû au fait que ce jeu particulier ne comptait pas beaucoup de sous arbres fréquents et qu'étant donné le nombre moyen d'items et le fait que les sous arbres étaient profonds (profondeur = 9), un grand nombre de calculs étaient indispensables.

Enfin, pour vérifier le comportement de nos algorithmes lors de l'augmentation du nombre de sous arbres, nous avons fait varier le nombre de TID (Cf. Figure 69) de 100 à 1000 pour des bases de profondeur 6. La Figure 73 illustre le comportement pour trois valeurs de support : 0.025%, 0.5% et 1%. Nous remarquons qu'à chaque fois nous avons un comportement linéaire des algorithmes avec des temps de réponse proportionnels à l'augmentation du nombre de TID quelque soit le support spécifié.





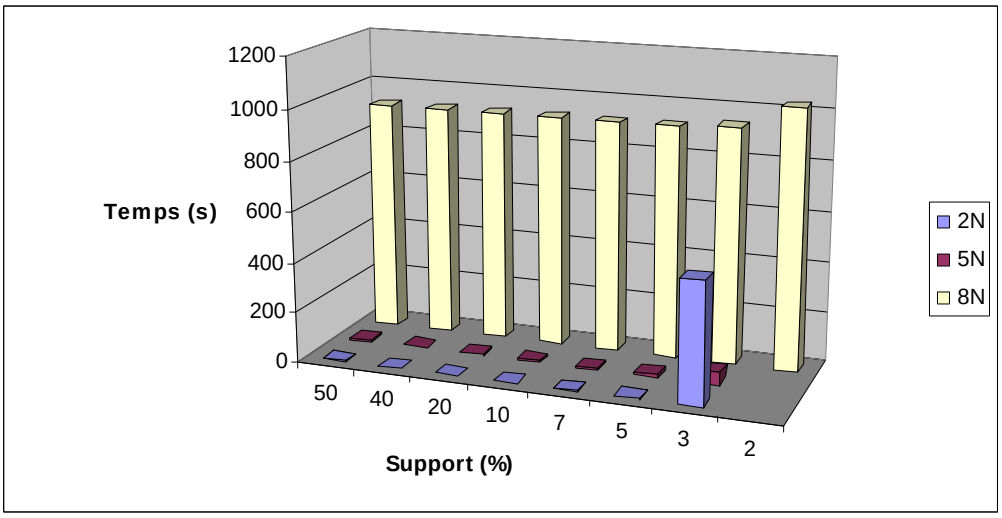
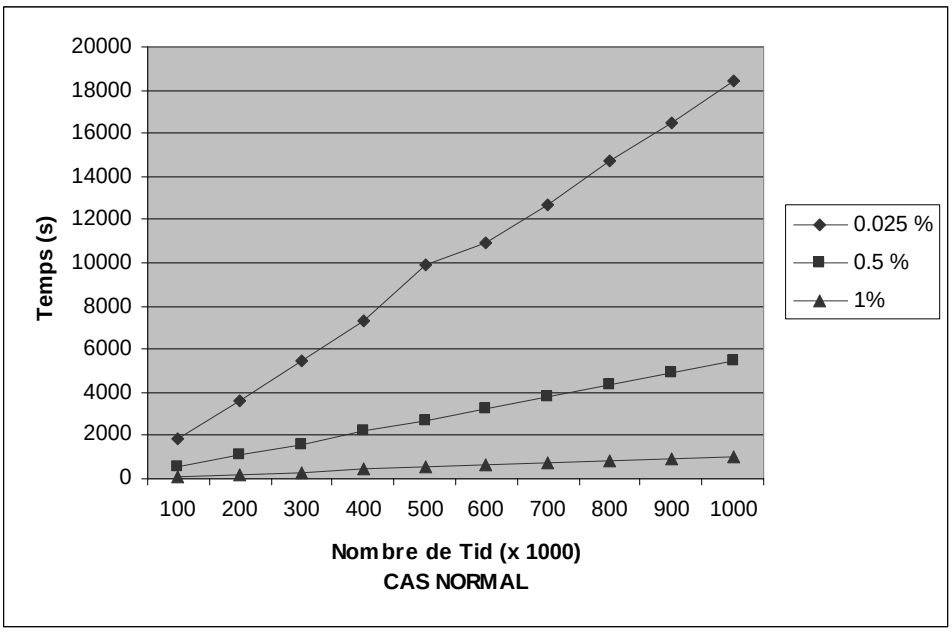


Figure 72 - Temps de réponse



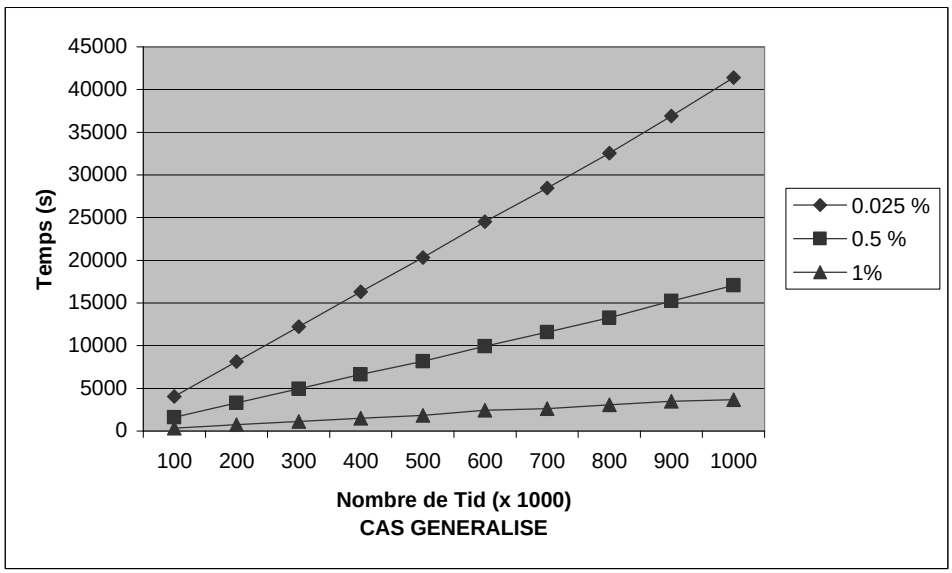


Figure 73 – Linéarité comportementale

6 Discussion

Dans cette section nous revenons sur la validité de l'approche et des algorithmes proposés. Nous avons vu au cours de la section 2 qu'une approche naïve pour résoudre ce problème semblait être, dans un premier temps, d'utiliser des algorithmes de motifs séquentiels pour rechercher des sous arbres fréquents. Cependant nous avons pu constater qu'une telle approche ne pouvait être utilisée pour répondre à notre problématique. Aussi nous avons proposé un nouvel algorithme PSP_{tree} qui permet de rechercher les sous arbres fréquents qui respectent notre problématique. Les expériences menées avec cet algorithme ont montré au travers de jeux de données réelles que les résultats obtenus répondaient à nos espérances et que l'approche proposée était adaptable à un grand nombre de problèmes allant de l'analyse du contenu de site Web à l'analyse du comportement des utilisateurs d'un site. En outre, nous avons poursuivi les expériences pour examiner le comportement de PSP_{tree}. Nous avons ainsi pu constater les performances de cet algorithme qui sont comparables à celles des algorithmes classiques de recherche de règles d'association ou d'extraction de motifs séquentiels. En outre, nous avons pu constater que PSP_{tree} pouvait supporter une « montée en charge » du nombre de sous arbres fréquents à extraire.
Dans un deuxième temps, nous avons souhaité restreindre certaines contraintes associées à notre problématique. En relachant ces dernières et plus particulièrement en permettant de rechercher des sous arbres fréquents dont les

branches peuvent être à différents niveaux nous avons pu constater que notre nouvelle approche, PSP_{tree},-GENERALISE, possédait un comportement assez semblable à celle de PSP_{tree}.
Une question toutefois reste posée : à partir du moment où les contraintes de niveaux dans les arbres ont été supprimées, est-ce qu'une approche basée sur l'utilisation d'algorithme de recherche de séquence pourrait être adéquate pour résoudre notre problématique ? En effet, en permettant de ne plus avoir forcément des imbrications profondes, il semblerait que le problème puisse être résolu par une approche comme GSP. En fait, même si la problématique associée à la notion d'imbrication que nous avons vue dans la section 2 est résolue, celle des frères et du nombre de candidats générés restent les mêmes. En ce qui concerne ce dernier, étant donné que le but poursuivi est de minimiser ce nombre, une approche de recherche de motifs séquentiels n'est donc pas directement utilisable.

Comparons à présent notre approche à celles décrites dans le chapitre précédent pour rechercher des sous arbres fréquents. L'approche la plus proche de notre problématique est celle de *TreeMinerH* ou *TreeMinerV*.

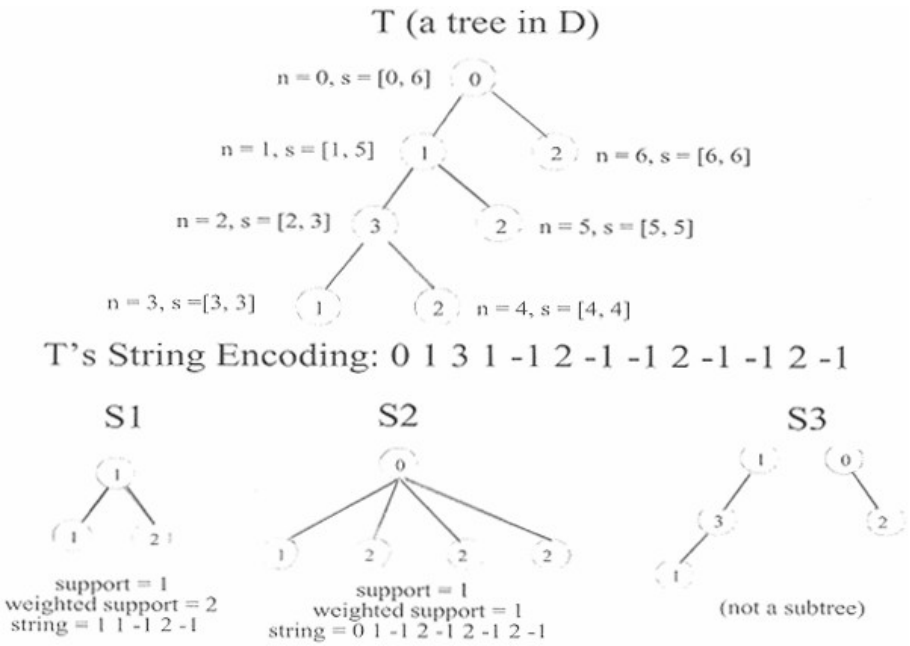


Figure 74 – Inclusion dans TreeMiner

Considérons l'exemple de la structure *S3* de la Figure 74. Dans le cas des deux approches, TreeMiner et PSP_{tree}-GENERALISE, cette structure n'est pas incluse car il ne s'agit pas d'un sous arbre. Par contre, dans les cas *S1* et *S2*, *TreeMiner* considère que ces structures sont incluses dans *T*. Dans notre cas, nous considérons qu'elles ne sont pas incluses. Pour *S1*, il existe le nœud étiqueté « 2 » entre les deux « 1 » de la branche de gauche de l'arbre *T*. De la même manière *S2* n'est pas imbriquée dans la mesure où les deux arbres ne possèdent pas la même topologie. Ainsi même si les problématiques semblent proches, les algorithmes proposés dans *TreeMiner* ne sont pas adaptés à notre contexte.

Les données contenues dans les bases évoluent de manière constante. Par exemple, nous avons vu que notre approche pouvait permettre d'analyser le comportement des utilisateurs d'un site Web. Il est bien évident que le nombre de visiteurs évolue sans cesse. Dans ce cas, comment tenir compte des connaissances acquises par PSP_{tree} ou PSP_{tree}-GENERALISE pour rapidement prendre en compte les modifications des données sources ? Nous répondons à ce problème dans le chapitre suivant.

Chapitre IV - Maintenance des connaissances extraites

Le chapitre est organisé de la manière suivante. Dans un premier temps, nous définissons les différents types de mises à jour ainsi que la notion de bordure négative. Nous montrons également dans cette partie comment intégrer cette bordure négative aux algorithmes du chapitre précédent. Nous nous focalisons dans ce chapitre sur l'évolution des connaissances dans le cadre de l'algorithme PSP_{tree}. En ce qui concerne la mise à jour dans le cadre de l'extraction de connaissances généralisées le principe général reste le même, même si certaines adaptations doivent être mises en œuvre. Ces adaptations ne seront pas détaillées mais les résultats des expérimentations seront présentés. La section 2 précise les conséquences que peut avoir une mise à jour sur une structure préfixée existante et montre comment les prendre en compte. Dans la section 3, nous analysons les algorithmes mis au point pour chacune de ces différentes mises à jour : variation du support minimal, ajout, suppression et modification de transactions. La section 4 présente les différentes expérimentations avec les algorithmes de la section précédente. Enfin dans la section 5, une discussion conclut le chapitre.

1 Mise à jour et bordure négative

1.1 Nécessité de mettre à jour

La mise à jour des connaissances obtenues par les algorithmes du Chapitre III devient une nécessité dans un environnement où les bases de données considérées évoluent. En effet, la validité des résultats est limitée au fait que les bases sur lesquelles nous travaillons sont statiques. Or, ces bases dans de nombreux domaines sont amenées à évoluer : il suffit de considérer le cas de l'analyse de visiteurs sur un site Web pour s'en convaincre. De nouvelles structures peuvent être ajoutées, supprimées ou modifiées. Il est aussi intéressant pour une même base de données d'effectuer plusieurs extractions de connaissance pour des supports différents. Dans le cadre de ces modifications, les résultats obtenus pourraient devenir obsolètes. Il est donc important dans un tel contexte de disposer d'algorithmes permettant de mettre à jour ces résultats. En ce qui concerne les variations de support, il semble judicieux d'utiliser une partie de l'information déjà calculée pour effectuer d'autres calculs. L'objectif visé est de fournir aux utilisateurs des algorithmes permettant d'effectuer la mise à jour des résultats précédents (modification et variation de support) sans pour autant effectuer un calcul depuis zéro.

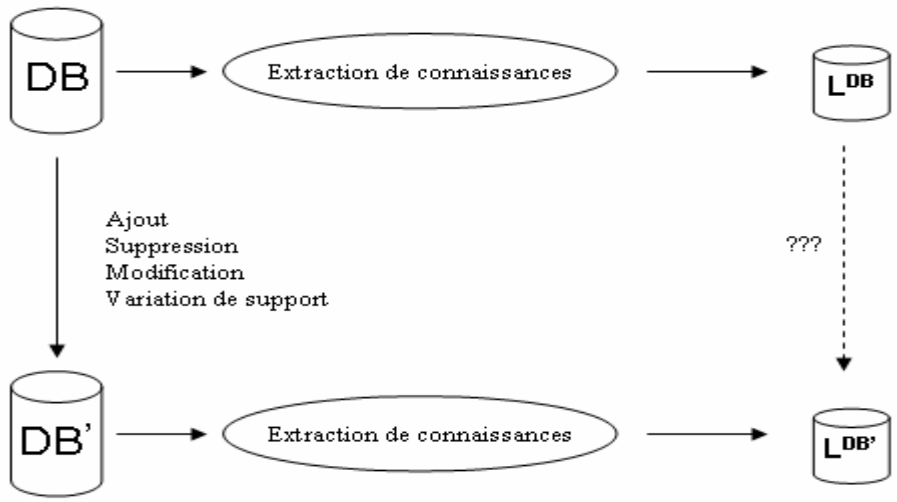


Figure 75 – Problématique de la mise à jour

Nous pouvons voir sur la Figure 75, une description de la mise à jour telle que nous la considérons. L'idée de ce chapitre réside dans la définition de la flèche de droite représentée en pointillé sur le schéma. L'objectif est que cette flèche remplace la solution utilisable actuellement représentée par la flèche en trait plein du bas : elle représente l'application de l'algorithme d'extraction de connaissances sur la base de données modifiées, i.e. à partir de zéro. Afin de mettre en œuvre des algorithmes permettant de résoudre cette problématique, nous utilisons la notion de bordure négative.

1.2 La bordure négative

La notion de bordure négative a été introduite par [MaTo96] et était initialement utilisée dans le domaine des règles d'associations. Etant donné que l'utilisation de cette bordure a été également utilisée dans le domaine des motifs séquentiels (algorithme ISM, C.f. Chapitre II), nous l'adaptons au cadre de la recherche de sous arbres fréquents.

Définition 17 :

La bordure négative est constituée de l'ensemble des séquences $\{s_1, s_2, \ldots, s_n\}$ tel que soit s_i, $1 \leq i \leq n$, un élément de la bordure négative de taille k, si s_j de taille $k-1$ est une sous séquence de s_i représentant un arbre alors s_j est une sous structure fréquente.

Exemple 44 :

Si la séquence $<(\oplus Personne_1)\ (\oplus Identite_2)\ (\perp Nom_3)\ (\oplus Identite_2)\ (\perp Adresse_3)>$ *est membre de la bordure négative alors la séquence* $<(\oplus Personne_1)\ (\oplus Identite_2)\ (\perp Nom_3)\ (\oplus Identite_2)>$ *est fréquente.*

Cette notion comme nous l'expliciterons dans la section suivante constitue la base des algorithmes de mise à jour que nous avons développés. Pour mettre en œuvre ces algorithmes, nous allons dans un premier temps étendre les algorithmes d'extraction de connaissances du Chapitre III pour obtenir le schéma de la Figure 76.

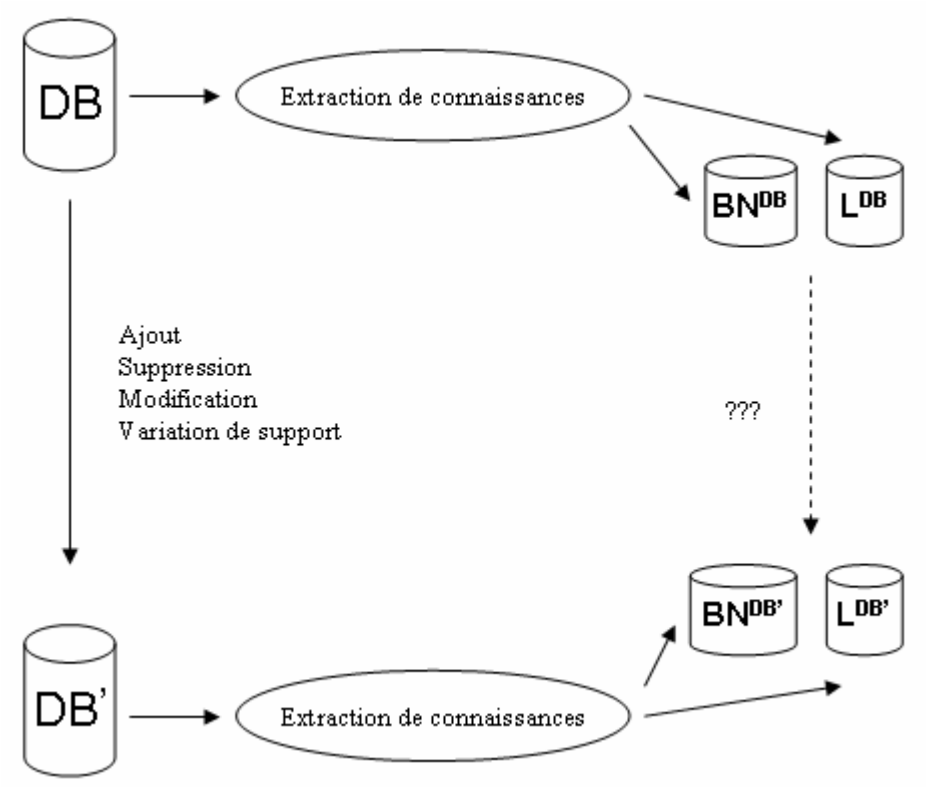


Figure 76 – Mise à jour et bordure négative

Dans la structure préfixée utilisée par PSP_{tree} et PSP_{tree}-*GENERALISE* nous stockons cette bordure négative. Pour cela, il suffit de modifier les algorithmes *extensionPossible* et *verifyCandidate* de la manière suivante. Nous stockons tous les éléments de taille 1 et non pas seulement les fréquents de taille 1. Nous appelons également l'algorithme *extensionPossible* avec tous les éléments de taille 1, ce qui permettra d'obtenir toutes les règles d'extension possibles fréquentes ou non. Lors de l'extension des niveaux $k > 2$, au lieu d'éliminer les candidats dont la valeur de support est inférieure au support minimal, nous les conservons dans la structure préfixée. Il suffit de modifier les dernières lignes de *verifyCandidate* pour marquer un élément dont le support est inférieur au support minimal comme membre de la bordure négative en positionnant un booléen à faux. Une fois la recherche de structures fréquentes terminée, nous obtenons un arbre préfixé contenant les informations suivantes : tous les éléments de niveau 1, toutes les règles possibles d'extension (fréquentes ou non), les structures fréquentes et les éléments de la bordure négative correspondants. Nous conservons également la valeur du support de tous ces éléments et un booléen définissant son statut : fréquent ou membre de la bordure négative. Cet arbre préfixé, noté *T*, est la structure qui nous servira de base de travail pour les algorithmes de mise à jour.

2 Conséquences des mises à jour sur la structure préfixée

Lors des différentes mises à jour, la structure préfixée va évoluer. Certains de ses éléments vont changer d'état : devenir fréquents ou membres de la bordure négative, d'autres vont apparaître ou disparaître. Nous allons définir sous la forme d'événement atomique ces modifications et nous proposons des algorithmes pour prendre en compte les conséquences de ces événements sur *T*. Les nœuds contenus dans la structure *T* représentent par leur construction pour $k \geq 2$ soit une règle d'extension (fréquente ou non), soit une séquence qui représente un arbre (fréquente ou membre de la bordure négative). Lors de ces mises à jour les différents événements atomiques à considérer sont : le changement d'état d'une règle d'extension, l'apparition d'une nouvelle règle, la disparition d'une règle et le changement d'état d'une séquence.

2.1 Changement d'état d'une séquence

Toute séquence contenue dans *T* possède un support. Selon la valeur de ce support cette séquence, i.e. l'arbre qu'elle représente, est soit fréquente, soit membre de la bordure négative. Le changement d'état d'une séquence signifie que soit elle était fréquente et elle devient membre de la bordure négative, soit elle était membre de la bordure négative et elle devient fréquente. Nous considérons ces deux cas dans les sections suivantes.

2.1.1 Passage du statut de fréquent à celui de membre de la bordure négative

Si une séquence passe du statut de fréquent à celui de membre de la bordure négative, cela signifie qu'elle ne peut pas être étendue par l'algorithme d'extraction de connaissances. Par conséquent tous les nœuds descendants initialement dans *T* n'ont plus de raison d'exister. Ces conséquences sont traduites dans l'algorithme *seqFreqToNegBorder*.

Algorithm seqFreqToNegBorder

Input : la structure préfixée *T*, le nœud n de T qui représente la séquence changeant d'état.
Output : *T* après prise en compte du changement d'état de n.

1 : *n.etat = false; // false = membre de la bordure négative*
2 : *supDesc(n,T);*

Algorithme 11 – L'algorithme seqFreqToNegBorder

L'algorithme est relativement simple. Après avoir changé l'état du nœud *n* considéré (ligne 1), les descendants de ce nœud sont supprimés par un appel à la procédure *supDesc*. Cette dernière recherche tous les descendants de *n* dans *T* et les supprime. La structure *T* est ainsi mise à jour au vu du passage du nœud *n* depuis l'ensemble des séquences fréquentes à celui de membre de la bordure négative.

2.1.2 Passage du statut de membre de la bordure négative à celui de fréquent

Dans le cas où l'inverse se produit, cette séquence peut-être étendue à nouveau pour générer de nouveaux candidats comme si cette séquence avait été construite lors de la phase de génération classique de l'algorithme

d'extraction de connaissances. En fait ce changement revient à construire la partie de *T* issue de ce nœud comme le montre l'algorithme s*eqNegBorderToFreq* d'une manière similaire à l'algorithme PSP_{tree}.

Algorithm seqNegBorderToFreq

Input : la structure préfixée *T*, le nœud n de *T* qui représente la séquence changeant d'état, le support minimal *(minSupp)* et la base de données considérée *DB*.
Output : *T* après prise en compte du changement d'état de n.

```
1 :  C = {n};
2 :  while C ≠ ∅ do
3 :      verifyCandidate(T,DB,minSupp) ;
4 :      L = {c ∈ C/ c ∈ T} ;// seuls les candidats fréquents sont encore dans T;
5 :      L^DB ← L^DB ∪ L ;
6 :      generateCandidate(T);
7 :      If T is updated by generateCandidate
8 :          then C = T ;
9 :          else C = ∅ ;
10 :     endif
11 : enddo
12 : Return L^DB
```

Algorithme 12 – L'algorithme seqNegVorderToFreq

L'algorithme *seqNegBorderToFreq* est proche dans son fonctionnement de celui de PSP_{tree}. La seule différence se situe au niveau de l'initialisation à la ligne 1. En effet la seule séquence à considérer dans ce cas est celle représentée par *n*. La suite du déroulement de l'algorithme est identique. Après son exécution, la structure *T* est à jour au vu du changement d'état du nœud *n*.

2.2 Changement d'état d'une règle d'extension

Tout comme dans la section 2.1, une règle d'extension contenue dans *T* peut changer d'état, i.e. devenir fréquente ou membre de la bordure négative. Nous allons analyser les conséquences d'un tel changement.

2.2.1 Passage du statut de règle fréquente à celui de membre de la bordure négative

Le passage d'une règle fréquente au statut de membre de la bordure négative signifie que lors de la phase d'extraction de connaissances précédentes il est possible que des candidats aient été générés à partir de cette règle car elle était fréquente. Du fait que cette règle est maintenant membre de la bordure négative ses éléments n'ont plus lieu d'être présents dans *T*. Il faut donc supprimer les nœuds de *T* générés par cette règle.

Algorithm ruleFreqToNegBorder

Input : la structure préfixée *T*, la règle d'extension *r : x => y* changeant d'état, le nœud *n* de profondeur 1 de *T*.
Output : *T* après prise en compte du changement d'état de *r*.

```
1 : ForEach  nœud_i descendant de n in T
2 :     If (nœud_i = r.y and pere(nœud_i) = r.x) then
3 :         supDesc(noeudi,T);
4 :         T.delete(nœud_i);
5 :     endif
7 : enddo
```

Algorithme 13 – L'algorithme ruleFreqToNegBorder

L'algorithme *ruleFreqToNegBorder* parcours successivement tous les noeuds descendants du noeud *n* de la structure préfixée *T* (ligne 1 – 7). Le nœud *n* est le seul nœud de *T* ayant une étiquette de profondeur valant 1. Si le nœud courant, noté $noeud_i$, a été construit par utilisation de la règle d'extension *r* alors il n'a plus de raison d'exister (ligne 4) mais au préalable nous supprimons tous les descendants de ce nœud dans *T* (ligne 3), i.e. nous

supprimons toutes les séquences construites par extension de la séquence représentée par $noeud_i$. Après son exécution, la structure *T* est à jour au vu du changement d'état de la règle d'extension *r*.

2.2.2 Passage du statut de règle membre de la bordure négative à celui de règle fréquente

Le passage d'une règle membre de la bordure négative au statut de règle fréquente signifie que des structures candidates peuvent être générées à partir de cette règle. Pour mettre à jour la structure *T* il faut donc générer toutes ces structures candidates qui sont des extensions de structures fréquentes existantes et les étendre si cela est possible.

Algorithm ruleNegBorderToFreq

Input : la structure préfixée *T*, la règle d'extension *r : x => y* changeant d'état, le support minimal (*minSupp*) et la base de données considérées *DB*.
Output : *T* après prise en compte du changement d'état de *r*.

```
1 :  C = ∅;
2 :  ForEach  nœud_i  in T where noeudi.Etat = true
3 :       If (nœud_i = r.x) then
4 :          nouv_noeud = ajouterArc(noeudi,creerNoeud(r.y));
5 :          C += nouv_noeud;
6 :       endif
7 :  enddo
8 :  while C ≠∅ do
9 :     verifyCandidate(T,DB,minSupp) ;
10 :     L = {c ∈ C/ c ∈ T} ; seuls les candidats fréquents sont encore dans T;
11 :           L^DB ← L^DB ∪ L ;
12 :     generateCandidate(T);
13 :     If T is updated by generateCandidate
14 :        then C = T ;
15 :        else C = ∅;
16 :     endif
17 : enddo
18 : Return L^DB
```

Algorithme 14 – L'algorithme ruleNegBorderToFreq

Après avoir initialisé l'ensemble des structures candidates à l'ensemble vide (ligne 1), l'algorithme *ruleNegBorderFreq* parcourt successivement tous les noeuds représentant des structures fréquentes de *T* (ligne 2 – 7). Pour chacune d'elles si le dernier élément qui la compose est égal à la première partie de la règle d'extension *r*, alors nous pouvons réaliser une extension de cette séquence par utilisation de cette règle. Pour cela, nous créons un nouveau nœud (ligne 4) que nous ajoutons à la structure *T* (ligne 4). Ce nœud nouvellement créé, représente une structure candidate, nous l'ajoutons donc à *C*, l'ensemble des structures candidates (ligne 5). Une fois cette première étape terminée, il suffit de tester ces structures candidates et de les étendre si possible de la même manière que l'algorithme d'extraction du chapitre précédent (lignes 8 – 18). Après son exécution, la structure *T* est à jour au vu du changement d'état de la règle d'extension *r*.

2.3 Apparition d'une nouvelle règle d'extension

Il peut arriver du fait de l'ajout de nouvelles transactions dans la base de structures considérée, que de nouvelles règles d'extension apparaissent. Soit *r* une nouvelle règle d'extension. Soit il s'agit d'une règle fréquente soit elle est membre de la bordure négative. Dans tous les cas il faut ajouter cette règle dans *T*, puis selon son état propager les conséquences au sein de *T*.

Algorithm ruleAdd

Input : la structure préfixée *T*, la règle d'extension *r : x => y*, le support minimal (*minSupp*), la base de données *DB.*
Output : *T* après prise en compte de l'ajout de *r*.

```
1 : nouv_noeud = creerNoeud(r.y);
2 : ajouterArc(r.x, nouv_noeud);
3 : C = nouv_noeud;
4 : verifyCandidate(T,DB,minSupp);
5 : If nouv_noeud.support ≥ minsupp then
6 :     ruleNegBorderToFreq(T,r,minSupp,DB);
7 : endif
```

Algorithme 15 – L'algorithme ruleAdd

Dans un premier temps, l'algorithme *ruleAdd* ajoute la règle dans la structure *T* (lignes 1 – 2) en créant un nouveau nœud et un arc dans l'arbre. Ensuite, nous utilisons la procédure *verifyCandidate* (ligne 4) pour calculer le support de cette nouvelle règle au sein de *T*. Si cette nouvelle règle est fréquente (ligne 5) alors les conséquences sur *T* sont identiques au passage d'une règle du statut de membre de la bordure négative à celui de règle fréquente, d'où l'appel à la procédure *ruleNegBorderToFreq* (ligne 6). Dans le cas où cette règle est membre de la bordure négative il n'y a aucune autre modification à apporter à *T*. Après son exécution, la structure *T* est à jour au vu de l'ajout de la règle d'extension *r* dans *T*.

2.4 Disparition d'une règle d'extension

Si des suppressions de transactions interviennent dans la base de structures, une règle d'extension *r* peut voir son support devenir nul. Si cette règle était fréquente il faut par un appel à *ruleFreqToNegBorder* effectuer les changements nécessaires dans *T* et la supprimer de *T*.

Algorithm ruleDel

Input : la structure préfixée *T*, la règle d'extension *r : x => y*, le support minimal (*minSupp*), la base de données *DB.*
Output : *T* après prise en compte de l'ajout de *r*.

```
1 : If r.support ≥ minSupp then
2 :     ruleFreqToNegBorder(T,r,n);
3 : endif
4 : T.supprimer(r.y) ;
```

Algorithme 16 – L'algorithme ruleDel

Si la règle *r* à supprimer de *T* était une règle fréquente (ligne 1) alors les conséquences sur *T* sont identiques au passage de cette règle de l'état fréquent à celui de membre de la bordure négative. Il suffit pour prendre en compte sa suppression de faire appel à la procédure *ruleFreqToNegBorder* (ligne 2). Dans le cas où cette règle était membre de la bordure négative sa suppression n'a pas d'autre impact sur *T*. La ligne 4 supprime la règle de *T*. Après son exécution, la structure *T* est à jour au vu de la suppression de la règle d'extension *r* dans *T*.

Maintenant que nous avons analysé les différentes conséquences des événements sur la structure *T* et proposé des algorithmes pour les prendre en compte, nous allons dans la section suivante montrer pour chaque cas de mise à jour quels sont les événements déclenchés.

3 Les différentes mises à jour

Nous allons considérer plusieurs types de mises à jour dans cette section. Pour chaque mise à jour étudiée : variation du support minimal, ajout, suppression et modification de transactions, nous présentons le principe de cette mise à jour, un exemple et l'algorithme mis en œuvre.

3.1 Variation du support minimal

Le premier cas de mise à jour étudié considère la variation de la valeur du support spécifié par l'utilisateur. La problématique qui définit ce premier cas est la suivante : soit *DB* une base de données de structures, soit *T* la structure préfixée générée par l'extraction de connaissances pour une valeur de support *minSupp* donnée et soit *minSuppNew* la nouvelle valeur de support. L'objectif de la mise à jour consiste à mettre à jour la structure *T* afin qu'elle reflète la recherche de structures fréquentes pour une valeur de support minimal *minSuppNew*. Nous distinguons deux cas : dans le premier la valeur de *minSuppNew* est supérieure à celle de *minSupp* et dans le second cette valeur est inférieure.

3.1.1 Augmentation du support minimal

Dans le cas où *minSuppNew* est plus grand que *minSupp*, intuitivement nous pouvons dire que tous les éléments qui pour une valeur de *minSupp* étaient fréquents vont le rester, et que d'autres vont au contraire devenir membres de la bordure négative. Il en est de même pour les règles d'extension. Il suffira de propager les conséquences de ces changements d'états. Considérons l'exemple suivant qui illustre les mécanismes mis en œuvre.

Arbre_id	Arbre
T_1	($\oplus$Personne$_1$) ($\oplus$Identite$_2$) ($\perp$Nom$_3$) ($\oplus$Identite$_2$) ($\perp$Adresse$_3$)
T_2	($\oplus$Personne$_1$) ($\oplus$Identite$_2$) ($\perp$Nom$_3$)

Figure 77 - La base de données de l'exemple

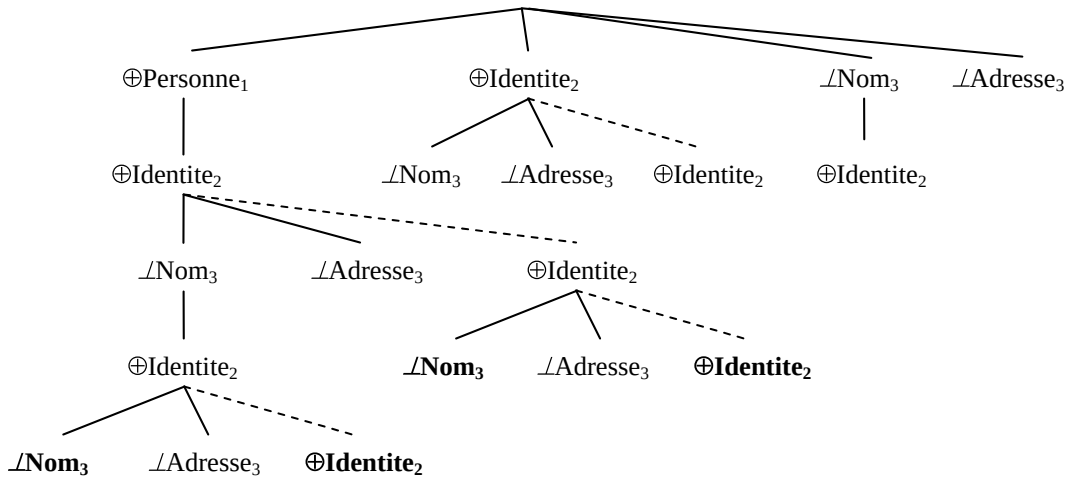


Figure 78 - La structure préfixée pour minSupp = 50%

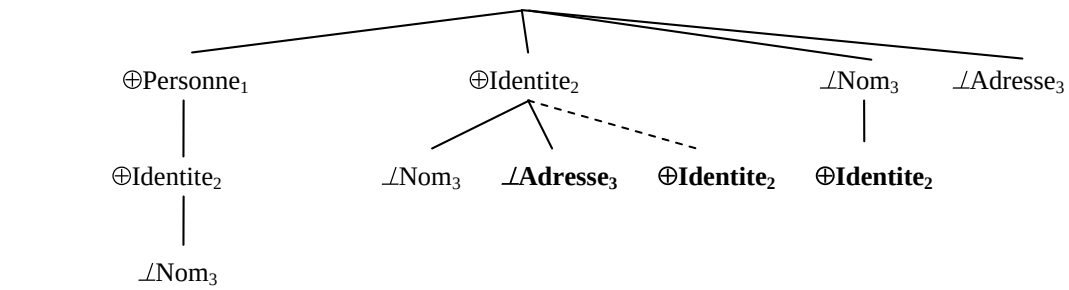


Figure 79 - La structure préfixée pour minSuppNew = 100%

Exemple 45 :

La Figure 78 montre la structure préfixée T contenant les résultats de l'extraction de connaissances pour une valeur de support minimal de 50% sur les éléments de la base de données exemple de la Figure 77. Sur cette figure, nous pouvons voir en gras les éléments de la structure appartenant à la bordure négative. Le nœud (⊕Personne$_1$) est appelé racine de l'arbre des candidats, nous pouvons l'identifier car il est le seul élément ayant une profondeur de 1. Les autres nœuds sous la racine de T représentent les règles d'extension possibles. La Figure 79 montre cette structure après prise en compte du passage de la valeur de support minimal de 50% à 100%. Nous constatons que certaines règles d'extension sont passées du statut de fréquent à celui de membre de la bordure négative : (⊕Identite$_2$) (⊥Adresse$_3$), (⊕Identite$_2$) (⊕Identite$_2$), (⊥Nom$_3$) (⊕Identite$_2$). Les autres règles initialement fréquentes n'ont pas changé d'état. La structure (⊕Personne$_1$) (⊕Identite$_2$) (⊥Adresse$_3$) qui était fréquente pour une valeur de support égale à 50%, est supprimée de l'arbre car la règle d'extension qui avait permis sa génération a été supprimée donc cette structure n'a pas lieu d'être. Il en est de même pour (⊕Personne$_1$) (⊕Identite$_2$) (⊕Identite$_2$). La séquence (⊕Personne$_1$) (⊕Identite$_2$) (⊥Adresse$_3$) n'est plus extensible du fait du changement d'état de la règle (⊥Nom$_3$) (⊕Identite$_2$). De ce fait ses descendants sont élagués dans l'arbre préfixé. La structure fréquente de taille maximum est (⊕Personne$_1$) (⊕Identite$_2$) (⊥Nom$_3$).

Algorithm MajAugSupp

Input : Une structure préfixée *T*, la nouvelle valeur du support *minSuppNew*. Un nœud *R* de l'arbre des candidats. La liste *RèglesPlusFréquentes* des règles d'extension qui ne sont plus fréquentes et qui l'étaient avant.
Output : L'arbre *T* mis à jour.

```
1  :  if support(R) < minSuppNew then
2  :     if Etat(R) = F then
3  :        seqFreqToNegBorder(R);
4  :     endif ;
5  :     MajAugSupp(T,minSuppNew,frère(R));
6  :  else
7  :    ruleFreqToNegBorderLocal(R,RèglesPlusFréquentes);
8  :    MajAugSupp(T,minSuppNew,frère(R));
9  :    MajAugSupp(T,minSuppNew,fils(R));
10 :  endif ;
```

Algorithme 17 – L'algorithme MajAugSupp

Avant de lancer cet algorithme, il est nécessaire de déterminer quelles sont les règles d'extension contenues dans *T* qui ne sont plus fréquentes. Cet algorithme est trivial, il consiste à parcourir les règles d'extension contenues dans *T* (i.e. le niveau 2 de *T*) et de mettre dans *RèglesPlusFréquentes* les règles d'extension qui ne sont plus fréquentes et de changer l'indicateur d'état. Le principe de l'algorithme suit l'intuition énoncée précédemment. Il est récursif, le premier appel s'effectue sur le nœud racine de l'arbre des candidats noté *R*. Le support du nœud de cet arbre est comparé à la valeur de support minimal *minSuppNew* passée en paramètre (ligne 1). Ce support est soit inférieur, soit supérieur ou égal à cette valeur. Dans le premier cas, si ce nœud était fréquent (ligne 2) alors nous le changeons d'état et nous supprimons ses descendants par appel à la procédure *seqFreqToNegBorder* (ligne 3). Il reste à rappeler la procédure sur le frère de *R* (ligne 5). Dans le second cas, il faut supprimer les descendants de ce nœud qui ont été générés par des règles qui ne sont plus fréquentes d'une manière similaire à la procédure *ruleFreqToNegBorder* (ligne 7) par l'appel à la procédure *ruleFreqToNegBorderLocal* sur ce nœud avec comme paramètre la liste des *RèglesPlusFréquentes*. Cette procédure récursive a un comportement similaire à *ruleFreqToNegBorder*. Elle détruira les descendants de *R* qui ont été générés par des règles préalablement fréquentes. Puis, nous rappelons l'algorithme sur le frère de ce nœud (ligne 8) et sur son premier fils (ligne 9). Lorsque tous les appels récursifs sont terminés, l'arbre préfixé *T* est mis à jour.

Propriété 11 :

Après exécution de l'algorithme, la structure *T* mise à jour contient tous les sous arbres et les règles d'extension fréquentes au vu de *minSuppNew* et uniquement ceux-ci ainsi que la bordure négative correspondante mise à jour.

Démonstration :
L'arbre *T* initial avant application de l'algorithme contient tous les arbres fréquents et toutes les règles d'extension fréquentes pour une valeur de support minimal inférieure. Si nous notons F_{avant} l'ensemble des arbres fréquents pour l'ancien support et $F_{après}$ l'ensemble des arbres fréquents pour le nouveau support, la relation suivante est vraie : $F_{après}$ est inclus ou égal à F_{avant}. Si nous ne touchons pas à cette structure nous avons donc bien « tous les sous arbres fréquents ». Nous appliquons le même raisonnement pour R_{avant} et $R_{après}$ représentant les règles d'extension. Il reste à prouver que nous n'avons que « ceux-la ». Si le support d'un sous arbre est inférieur au nouveau support il devient membre de la bordure négative et tous les sous arbres descendants de ce nœud dans la structure sont supprimés par la procédure *seqFreqToNegBorder*. En ce qui concerne les règles d'extension, préalablement à l'exécution de l'algorithme, nous avons effectué un prétraitement permettant la mise à jour de leur état et la création d'une liste contenant *RèglesPlusFréquentes, i.e.* les règles qui ont changé d'état. Si le support d'un sous arbre est supérieur alors la procédure *ruleFreqToNegBorderLocal (R, RèglesPlusFréquentes)* permet de détruite les éléments de *T* qui n'ont plus de raison d'être. A la fin de l'exécution de l'algorithme et de ces appels récursifs, *T* contient tous les fréquents présents au vu de la nouvelle valeur de support. Enfin la structure contient tous les éléments de la nouvelle bordure négative. En effet, pour chaque nœud considéré du parcours de la structure *T* depuis la racine jusqu'aux feuilles, à tout instant le nœud considéré est soit fréquent soit membre de la bordure négative. S'il est membre de la bordure négative, tous ses descendants sont élagués si nécessaire, donc à la fin de l'algorithme, la bordure négative est bien mise à jour. Pour les règles d'extension le prétraitement aura préalablement changé l'état des règles si nécessaire.

Complexité :
Soit *n* le nombre de nœuds de la structure *T*, dans le pire des cas, notre algorithme parcourt la totalité de l'arbre. Toutes les opérations de modifications effectuées lors de ce parcours ont un coût linéaire. Cet algorithme a une complexité en *O(n)* au pire des cas. Cette complexité est meilleure que celle de l'algorithme d'extraction de connaissance, ce qui valide son utilisation en cas d'augmentation de la valeur du support.

3.1.2 Diminution du support

Dans le cas ou minSuppNew est plus petit que *minSupp*, intuitivement nous pouvons dire que certains éléments qui pour une valeur de *minSupp* étaient fréquents vont le rester, et que d'autres qui étaient au préalable membres de la bordure négative vont au contraire devenir fréquents. Il en est de même pour les règles d'extension. Il suffira de propager les conséquences de ces changements d'états.

Arbre_id	Arbre
T_1	$(\oplus Personne_1)$ $(\oplus identite_2)$ $(\perp nom_3)$ $(\oplus adresse_3)$ $(\perp Rue_4)$ $(\perp Numero_4)$
T_2	$(\oplus Personne_1)$ $(\oplus identite_2)$ $(\perp nom_3)$

Figure 80 - La base de données de l'exemple

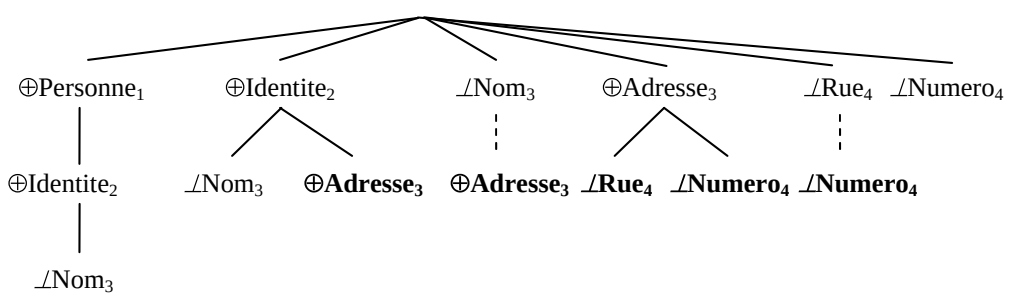


Figure 81 - La structure préfixée pour minSupp = 100%

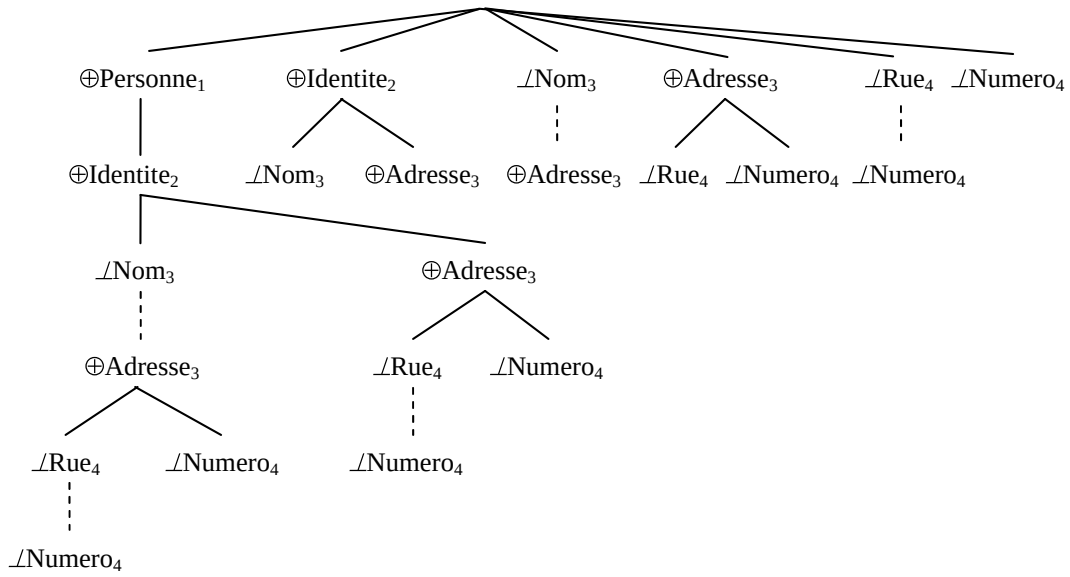


Figure 82 - La structure préfixée pour minSuppNew = 50%

Exemple 46 :

La Figure 81 montre la structure préfixée T contenant les résultats de l'extraction de connaissances pour une valeur de support minimal de 100% sur les éléments de la base de données exemple de la Figure 80. La Figure 82 montre cette structure après prise en compte du passage de la valeur de support minimal à 50%. Toutes les règles d'extension qui étaient précédemment membre de la bordure négative sont devenues fréquentes. Ainsi la structure (⊕Personne$_1$) (⊕Identite$_2$) (⊕Adresse$_3$) a été créée par le changement d'état de la règle (⊕Identite$_2$) (⊕Adresse$_3$). Cette structure est fréquente, de ce fait nous pouvons tester les deux candidats : (⊕Personne$_1$) (⊕Identite$_2$) (⊕Adresse$_3$) (⊥Rue$_4$) et (⊕Personne$_1$) (⊕Identite$_2$) (⊕Adresse$_3$) (⊥Numero$_4$) construits à partir des extensions possibles de l'élément (⊕Adresse$_3$). Il en est de même pour (⊕Personne$_1$) (⊕identite$_2$) (⊥nom$_3$) (⊕Adresse$_3$). Après mise à jour de cette structure les quatre structures fréquentes de taille maximum sont : (⊕Personne$_1$) (⊕Identite$_2$) (⊕Adresse$_3$) (⊥Rue$_4$) (⊥Numero$_4$), (⊕Personne$_1$) (⊕Identite$_2$) (⊕Adresse$_3$) (⊥Numero$_4$), (⊕Personne$_1$) (⊕Identite$_2$) (⊥nom$_3$) (⊕Adresse$_3$) (⊥Rue$_4$) (⊥Numero$_4$) et (⊕Personne$_1$) (⊕Identite$_2$) (⊥nom$_3$) (⊕Adresse$_3$) (⊥Numero$_4$).

Algorithm MajDimSupp

Input : Un arbre des candidats *T*, la nouvelle valeur du support *minSuppNew*. Un nœud *R* de l'arbre des candidats. La liste *NouvellesRèglesFréquentes* des nouvelles règles d'extension qui sont fréquentes.
Output : L'arbre *T* mis à jour.

```
1 :  if (Etat(R) = BN) and support(R) ≥ minSuppNew then
2 :    MajDimSupp(T,minSuppNew,fils(R));
3 :    MajDimSupp(T,minSuppNew,frère(R));
4 :    seqNegBorderToFreq(R);
5 :  else
6 :   MajDimSupp(T,minSuppNew,fils(R));
7 :   MajDimSupp(T,minSuppNew,frère(R));
8 :   ruleNegBorderToFreqLocal(R, NouvellesRèglesFréquentes);
9 :  endif ;
```

Algorithme 18 – L'algorithme MajDimSupp

Avant de lancer cet algorithme, comme pour l'augmentation de support, il est nécessaire de déterminer quelles sont les règles d'extension contenues dans *T* qui deviennent fréquentes. Cet algorithme est trivial, il consiste à

parcourir les règles d'extension contenues dans *T* (i.e. le niveau 2 de *T*) et à mettre dans *NouvellesRèglesFréquentes* les règles d'extension qui sont devenues fréquentes et à changer l'indicateur d'état. Le premier appel de l'algorithme s'effectue sur le nœud racine de l'arbre des candidats noté *R*. Si ce nœud est membre de la bordure négative, son support est comparé à la valeur de support minimal *minSuppNew* passée en paramètre (ligne 1). Ce noeud fréquent peut être étendu, cette opération est effectuée par l'appel de la procédure *seqNegBorderToFreq* (ligne 4). Préalablement, nous rappelons *MajDimSupp* sur les fils et frères de ce nœud (ligne 2- 3) afin d'éviter de traiter les nœuds issus de l'appel à *seqNegBorderToFreq*. Dans le cas où le nœud *R* est toujours fréquent (lignes 5 – 9), il faut étendre ce nœud en tenant compte du fait que certaines règles sont devenues fréquentes. Cette opération est effectuée par un appel à la procédure *ruleNegBorderToFreqLocal* en passant en paramètre les règles d'extension nouvellement fréquentes (ligne 8). Cette procédure est similaire à *ruleNegBorderToFreq*, la seule différence réside dans le fait qu'elle ne s'intéresse pas à tous les nœuds de *T* mais seulement au noeud *R* considéré. Tout comme le cas précédent nous rappelons *MajDimSupport* préalablement à cet appel à *ruleNegBorderToFreqLocal* pour les mêmes raisons (ligne 6 – 7). Lorsque tous les appels récursifs sont terminés, l'arbre préfixé *T* est mis à jour.

Démonstration MajDimSupp: L'arbre préfixé final contient au moins l'arbre préfixé initial. L'état des règles d'extension est mis à jour par le prétraitement préalable à l'appel de l'algorithme. L'algorithme de mise à jour parcourt bien tous les nœuds de la structure initiale (ligne 2-3 et 6-7). Pour les nœuds membres de la bordure négative deux cas se présentent, soit il sont toujours membres de la bordure négative, auquel cas s'ils l'étaient dans l'arbre initial ils le sont dans l'arbre final, soit ils deviennent fréquents. S'ils deviennent fréquents l'appel de la procédure *seqNegBorderToFreq* (ligne 4) est similaire à l'utilisation de l'algorithme d'extraction de connaissance du chapitre précédent sur un nœud particulier. Cet algorithme comme l'indique sa preuve mettra à jour la structure au vu du changement d'état de ce nœud. Il nous reste à traiter les nœuds déjà fréquents. Il n'y a de conséquence sur ces nœuds que si la liste *NouvellesRèglesFréquentes* est non vide, i.e. des règles d'extension qui n'étaient pas fréquentes initialement le sont devenues. Le fait qu'une règle change d'état a pour conséquence la génération d'un nouvel élément dans *T* qui est soit fréquent soit membre de la bordure négative. Les conséquences d'un tel changement sont similaires au fait d'effectuer une extraction de connaissances sur ce nœud. Cette opération est réalisée par un appel à la procédure *ruleNegBorderToFreqLocal* en utilisant les nouvelles règles d'extension. Au final, la structure préfixée est donc à jour au vu de la nouvelle valeur de support.

Complexité :
Dans le pire des cas, cet algorithme revient à appliquer l'algorithme d'extraction de connaissances à la racine de l'arbre. Sa complexité est identique à celle de cet algorithme. En moyenne, comme le montrerons les expérimentations, cet algorithme est plus rapide, car les nœuds changeant d'état sont souvent inférieurs au nombre de nœuds générés par l'algorithme d'extraction de connaissances depuis zéro.

3.2 Ajout de transactions

Si nous considérons *DB* une base de données sur laquelle nous avons effectué une recherche de structures fréquentes pour une valeur de support *minSupp*. Soit *db* la base de données incrément contenant les transactions à ajouter à *DB*. Soit *U = DB + db*, la base de données mise à jour contenant toutes les structures de *DB* et *db*. Le problème consiste donc à trouver toutes les structures fréquentes contenues dans *U*. Intuitivement les conséquences de l'ajout de transactions sont multiples : de nouvelles règles d'extension peuvent changer d'état, des règles d'extension peuvent apparaître, des séquences de *T* peuvent changer d'état. Considérons la base de données de la Figure 77 à laquelle nous ajoutons la transaction suivante : ($\oplus$Personne$_1$) ($\oplus$Identite$_2$) ($\angle$Nom$_3$) ($\angle$Adresse$_3$).

Arbre_id	Arbre
T_1	($\oplus$Personne$_1$) ($\oplus$Identite$_2$) ($\angle$Nom$_3$) ($\oplus$Identite$_2$) ($\angle$Adresse$_3$)
T_2	($\oplus$Personne$_1$) ($\oplus$Identite$_2$) ($\angle$Nom$_3$)
T_3	($\oplus$Personne$_1$) ($\oplus$Identite$_2$) ($\angle$Nom$_3$) ($\angle$Adresse$_3$)

Figure 83 - La base de données U = DB + db après ajout d'une transaction

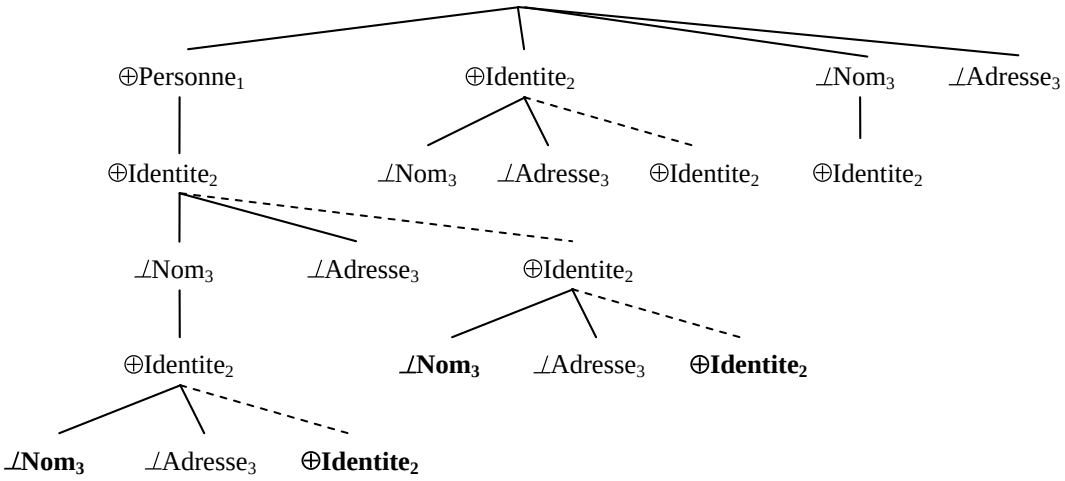


Figure 84 - La structure préfixée de DB pour minSupp = 30%

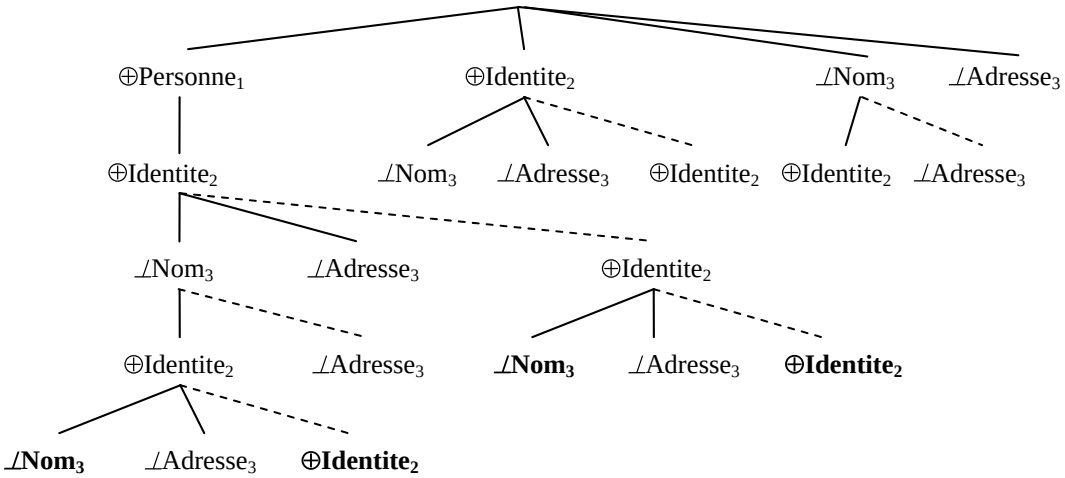


Figure 85 - La structure préfixée de U pour minSupp = 30%

Exemple 47 :

La Figure 84 montre la structure préfixée T contenant les résultats de l'extraction de connaissances pour une valeur de support minimal de 30% sur les éléments de la base de données DB de la Figure 83. La Figure 85 montre la structure préfixée T contenant les résultats de l'extraction de connaissances pour une valeur de support minimal de 30% sur les éléments de la base de données U. U correspond à la base de données DB à laquelle a été ajouté une transaction. Bien que relativement proche de la Figure 84, cette figure présente plusieurs différences. Tout d'abord, une nouvelle règle d'extension a été ajoutée. Elle autorise l'extension de l'élément (⊿Nom3) par l'élément (⊿Adresse3). Ensuite, un nouveau candidat a été testé par l'utilisation de cette nouvelle règle : (⊕Personne1) (⊕Identite2) (⊿Nom3) (⊿Adresse3) et il représente une structure fréquente nouvelle. Le reste de la structure est identique.

Pour effectuer la prise en compte des transactions ajoutées dans *db*, dans un premier temps nous appliquons l'algorithme *MajExtensionPossible*, puis l'algorithme *MajAjoutTransactions*. Le premier algorithme est proche de l'algorithme *MajExtension* présenté dans le chapitre précédent. L'objectif de celui ci est, à partir des transactions ajoutées contenues dans *db* (ligne 6) et des règles d'extension existantes contenues dans *T* (lignes 2 – 4), de détecter les nouvelles règles d'extension issues de *db* et de mettre à jour le support des règles déjà existantes. Chacune des règles contenues dans *db* est ainsi traitée. Elle est soit ajoutée à *T* (lignes 10 – 18 – 26 –

33) et marquée par un booléen comme étant nouvelle si elle n'existait pas, soit sa valeur de support est incrémentée de un (lignes 14 – 22 – 30 - 37). Avant de lancer le deuxième algorithme *MajAjoutTransactions* qui va parcourir les différents nœuds de la structure *T* à partir de la racine de l'arbre des candidats et effectuer les opérations nécessaires sur *T* pour que celle ci soit à jour au vu des extensions possibles, nous parcourons toutes les règles d'extension de *T* pour générer deux listes de manière similaire au prétraitement effectué lors de l'augmentation et de la diminution du support. La première liste *RèglesPlusFréquentes* contient les règles d'extension qui étaient fréquentes initialement et qui ne le sont plus. La seconde liste *NouvellesRèglesFréquentes* contient les règles d'extension qui sont devenues fréquentes et les nouvelles règles fréquentes. Pour chaque nœud considéré, la procédure *MajSupport* à partir de ce nœud et des informations de *db* met à jour le support du nœud *R* considéré (ligne 1), cette procédure utilise les principes de *verifyCandidate* pour ce calcul. Ensuite le support calculé est comparé à la valeur du support minimal *minSupp* (ligne 2). Si ce support est inférieur, deux cas se présentent : soit ce nœud était préalablement fréquent (ligne 3) et nous élaguons ses descendants par appel à la procédure *seqFreqToNegBorder* (ligne 4), soit ce nœud est déjà membre de la bordure négative auquel cas aucune modification n'est nécessaire. Etant donné que ce type de nœud n'a pas de descendant il ne reste plus qu'à rappeler *MajAjoutTransactions* sur le frère de ce nœud. Si le support est supérieur à *minSupp*, deux cas se présentent. Soit ce nœud était préalablement fréquent (ligne 8) et nous utilisons deux procédures *ruleNegBorderToFreq Local* et *ruleFreqToNegBorder*, de la même manière que dans la diminution du support minimal et l'augmentation de ce support, avec les deux listes calculées suite au prétraitement des règles d'extension par l'algorithme précédent (ligne 12 - 13). Au préalable, afin d'éviter le traitement des nœuds générés par ces deux procédures par *MajAjoutTransactions*, deux appels récursifs sont effectués sur les frères et fils de *R*. Soit ce nœud était membre de la bordure négative auquel cas nous appliquons l'algorithme *seqNegBorderToFreq* (ligne 17). Tout comme précédemment, nous rappelons *MajAjoutTransactions* sur les fils et frères de ce nœud (lignes 15 – 16). Cet algorithme comme l'indique sa preuve mettra à jour la structure au vu du changement d'état ou non du nœud considéré. Au final, la structure préfixée est donc à jour au vu de l'ajout des nouvelles transactions de *db*.

Démonstration MajExtensionPossible et MajAjoutTransactions:

Nous devons prouver qu'à la fin de l'exécution de ces deux algorithmes, nous avons bien la structure *T* qui représente les résultats d'une extraction de connaissances sur la base *U = DB + db* pour une valeur de support *minSupp*. Si les éléments de *db* contiennent de nouvelles extensions alors l'application *MajExtensionPossible* les prend en compte. En effet cet algorithme reprend les règles d'extension possibles de *DB*, recherche les nouvelles règles d'extension possibles contenues dans *db* et met à jour le support des règles déjà existantes. Cet algorithme, procède de la même manière que *MajExtension* dont nous avons prouvé la justesse dans le chapitre précédent. Après application de cet algorithme l'ensemble des règles d'extensions possibles et leur support sont à jour au vu des informations contenues dans *U*, i.e. les niveaux 1 et 2 de *T* sont à jour. Il reste à mettre à jour les autres niveaux de *T*. Tous les nœuds existant dans *T* sont atteints par l'algorithme *MajAjoutTransactions*. Pour chaque nœud la valeur de son support est celle de son support dans *DB* plus la valeur de son support *db* (calculé par la procédure *MajSupport*). Pour chaque nœud ainsi considéré de la structure *T* initiale il y a plusieurs possibilités : soit ce nœud était fréquent, soit il était membre de la bordure négative. Dans le cas ou il était membre de la bordure négative, soit il l'est toujours avec son nouveau support, auquel cas ce nœud dans la structure *T* mise à jour est identique, soit il devient fréquent auquel cas on applique la procédure *seqNegBorderToFreq* sur ce nœud avec l'ensemble des règles d'extension. Cette procédure effectue bien les mises à jour au vu des informations de *U* pour ce nœud comme montré dans la diminution de la valeur du support minimal (section 3.1.2). Dans le cas où il était fréquent, avec le nouveau support soit il l'est toujours auquel cas il suffit d'appeler les procédures *ruleNegBorderToFreqLocal* et *ruleFreqToNegBorderLocal* avec les deux listes calculées lors de la mise à jour des règles d'extension : *RèglesPlusFréquentes* et *NouvellesRèglesFré-quentes* pour mettre à jour ce nœud au vu des informations de *U*. Par contre, s'il devient fréquent alors pour mettre la structure à jour il suffit d'appliquer la procédure *seqNegBorderToFreq* sur ce nœud. Après parcours de la structure *T* et application de ces algorithmes, au final celle-ci contient toutes les informations relatives à l'extraction de connaissances sur la base de données *U* pour une valeur de support *suppMin*.

Algorithm MajExtensionPossible

Input : Un arbre des candidats *T*, la base de données *db*.
Output : L'arbre *T* mis à jour au vu des extensions possibles.

```
1 :  foreach F_i^1 ∈ T^1 do
2 :      F_i^1.listeFils ← reconstruitListeFils(T, F_i^1) ;
3 :      F_i^1.listeFrères ← reconstruitListeFrères(T, F_i^1) ;
4 :      F_i^1.listeFrèresAncêtres ← reconstruitListeFrèresAncêtres(T, F_i^1) ;
5 :  enddo
6 :  foreach Tree t_i ∈ db do
7 :      foreach F_i^1 ∈ t_i do
8 :          F_i^1.listeFrèresAncêtresLocaux ← ∅;
9 :          foreach fils f_i ∈ T^1 of F_i^1 in t_i do
10 :             if ((F_i^1.listeFils).ajoute(f_i)) then
11 :                 créerNoeud(f_i) ;
12 :                 ajouterArc(F_i^1,f_i) ;
13 :             else
14 :                 Règle(F_i^1,f_i).support++ ;
15 :             endif
16 :         enddo
17 :         foreach frère f_i ∈ T^1 of F_i^1 in t_i do
18 :             if ((F_i^1.listeFrères).ajoute(f_i)) then
19 :                 créerNoeud(f_i) ;
20 :                 ajouterArc(F_i^1,f_i) ;
21 :             else
22 :                 Règle(F_i^1,f_i).support++ ;
23 :             endif
24 :         enddo
25 :         foreach frère f_i ∈ T^1 of père(s)(F_i^1) in t_i do
26 :             if ((F_i^1.listeFrèresAncêtres).ajoute(f_i)) then
27 :                 créerNoeud(f_i) ;
28 :                 ajouterArc(F_i^1,f_i) ;
29 :             else
30 :                 Règle(F_i^1,f_i).support++ ;
31 :             endif
32 :             (F_i^1.listeFrèresAncêtresLocaux).ajoute(f_i) ;
31 :         enddo
32 :         foreach f_i ∈ T^1 of (père(s)(F_i^1)).listeFrèresAncêtresLocaux do
33 :             if ((F_i^1.listeFrèresAncêtres).ajoute(f_i)) then
34 :                 créerNoeud(f_i) ;
35 :                 ajouterArc(F_i^1,f_i) ;
36 :             else
37 :                 Règle(F_i^1,f_i).support++ ;
38 :             endif
39 :         enddo
40 :     enddo
41 : enddo
```

Algorithme 19 – L'algorithme MajExtensionPossible

Algorithm MajAjoutTransactions

Input : Un arbre des candidats *T*, la base de données *U*, la base de données *db*, la valeur du support minimal *minSupp*. Un noeud *R* de l'arbre des candidats. La liste *NouvellesRèglesFréquentes* des nouvelles règles d'extension qui sont fréquentes et la liste *RèglesPlusFréquentes* des règles d'extension qui ne le sont plus.
Output : L'arbre *T* mis à jour au vu des extensions possibles.

```
1 :  MajSupport(R, db)
2 :  If (Support(R) < minSupp) then
3 :     If (Etat(R) = F) then
4 :        seqFreqToNegBorder(R);
5 :     endif
6 :     MajAjoutTransactions(Frère(R));
7 :  else
8 :     If (Etat(R) = F) then
10 :       MajAjoutTransactions(Fils(R));
11 :       MajAjoutTransactions(Frère(R));
12 :       ruleNegBorderToFreqLocal(R, NouvellesRèglesFréquentes);
13 :       ruleFreqToNegBorderLocal(R, RèglesPlusFréquentes) ;
14 :    else
15 :       MajAjoutTransactions(Fils(R));
16 :       MajAjoutTransactions(Frère(R) ;
17 :       seqNegBorderToFreq(R);
18 :    endif
19 : endif
```

Algorithme 20 – L'algorithme MajAjoutTransactions

Complexité :

Cet algorithme revient à une série de tests préliminaires qui permettent de mettre à jour les supports des éléments contenus dans la structure *T* (fréquents, membres de la bordure négative et règles) et de calculer les nouvelles règles fréquentes. Ce coût est directement proportionnel au nombre de transactions contenues dans *db*. Cet ensemble de tests préliminaires possède un coût linéaire en la taille de l'incrément *db*. Soit *J*, les éléments fréquents de *T* qui demeurent fréquents après la prise en compte de l'incrément. Soit *K* les éléments membres de la bordure négative de *T* qui deviennent fréquents après la prise en compte de l'incrément. Soit *L* les éléments fréquents de *T* qui deviennent membres de la bordure négative. Une approche de la complexité de cet algorithme peut être vue comme suit. Le coût lié aux éléments de *L* est linéaire (suppression des descendants). Le coût lié aux éléments de *K* est similaire a celui engendré par PSP_{tree}. Enfin le coût le plus important est lié aux éléments de *J*. En effet pour chaque élément de *J*, le traitement est plus complexe. D'une part il faut l'étendre en utilisant les nouvelles règles fréquentes, ce qui engendre un coût similaire à celui engendré par PSP_{tree} tout comme dans le cas des éléments de *L*. D'autre part il faut vérifier que chaque fils de cet élément dans *T* n'a pas été généré par une règle qui n'est plus fréquente (nombreux parcours de listes). Si nous regardons globalement cet algorithme nous pouvons caractériser son coût en deux parties. Nous ne tenons pas compte du coût lié au traitement des éléments de *L*. Une première partie qui engendre un coût inférieur à celui de PSP_{tree} (calcul depuis zéro) liée au traitement des éléments de *K* et à une partie du traitement des éléments de *J* (celle qui utilise les nouvelles règles fréquentes). Nous pouvons qualifier cette partie de globalement constructive (i.e. elle ajoute des éléments à la structure). Une deuxième partie qui est liée au traitement des règles devenues non fréquentes pour les éléments de *J*. Son coût dépend du nombre de fils de chaque élément de *J* et du nombre de règles devenues plus fréquentes. Pour conclure on ne peut pas garantir ici quelque soit le jeu de données que le coût total de l'algorithme est inférieur à un calcul depuis zéro.

3.3 Suppression de transactions

La problématique est similaire à la précédente, sauf que les transactions contenues dans *db* sont supprimées de *DB*. Le problème consiste donc à trouver toutes les structures fréquentes contenues dans *U*. Intuitivement les conséquences de la suppression de transactions sont multiples : disparition de règles d'extension, changement d'état de certaines règles d'extension, changement d'état de certaines séquences. Considérons la base de données

de la Figure 86 à laquelle nous supprimons la transaction suivante : ($\oplus$Personne$_1$) ($\oplus$Identite$_2$) ($\oplus$Identite$_2$) ($\perp$Nom$_3$) ($\perp$Adresse$_3$).

Arbre_id	Arbre
T_1	($\oplus$Personne$_1$) ($\oplus$identite$_2$) ($\oplus$identite$_2$) ($\perp$nom$_3$) ($\perp$adresse$_3$)
T_2	($\oplus$Personne$_1$) ($\oplus$identite$_2$) ($\perp$nom$_3$)

Figure 86 - La base de données de l'exemple

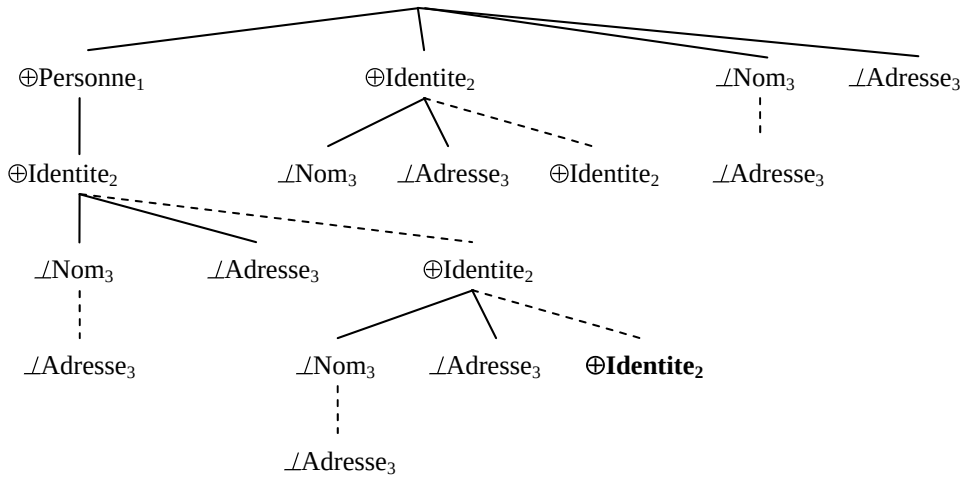


Figure 87 - La structure préfixée de DB pour minSupp = 50%

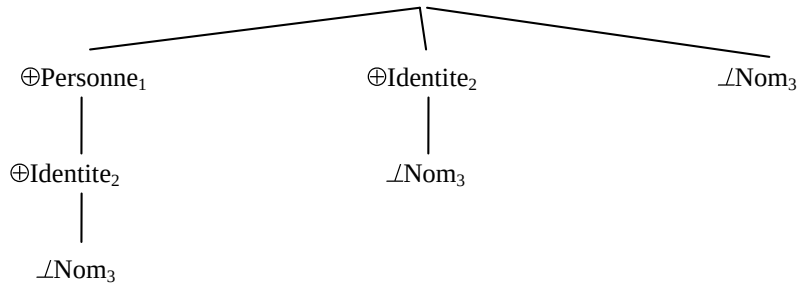


Figure 88 - La structure préfixée de U pour minSupp = 50%

Exemple 48 :

La Figure 87 montre la structure préfixée T contenant les résultats de l'extraction de connaissances pour une valeur de support minimal de 50% sur les éléments de la base de données DB la Figure 86. La Figure 88 montre la structure préfixée T contenant les résultats de l'extraction de connaissances pour une valeur de support minimal de 50% sur les éléments de la base de données U. U correspond à la base de données DB à laquelle on a supprimé une transaction. Cette figure présente plusieurs différences. Tout d'abord, plusieurs règles d'extension ont été supprimées. De ce fait la structure T finale est moins dense et donc la partie de celle-ci représentant les structures fréquentes est plus succincte. La seule structure fréquente existante est ($\oplus$Personne$_1$) ($\oplus$identite$_2$) ($\perp$nom$_3$). La bordure négative est réduite à l'ensemble vide.

La technique utilisée pour la suppression de transactions est similaire à celle de l'ajout de transactions. Un premier algorithme *MajExtensionPossibleSup* met à jour le support des différentes règles d'extension en utilisant les informations contenues dans *db*. Suite à l'application de cet algorithme, les listes *NouvellesRèglesFréquentes* et *RèglesPlusFréquentes* sont calculées. Il reste à exécuter l'algorithme *MajSuppressionTransactions* en tout point identique à *MajAjoutTransactions*. Enfin, il reste à supprimer de *T* les règles qui ne sont plus valides, i.e. dont la valeur de support est égale à zéro, opération effectuée par un parcours du niveau 2 de *T*. Au final, la structure préfixée est donc à jour au vu de la suppression des transactions.

Démonstration MajExtensionPossibleSup et MajSuppressionTransactions :
La démonstration est identique au cas précédent. Au final la structure *T* contient toutes les informations relatives à l'extraction de connaissances sur la base de données *U* pour une valeur de support *suppMin*. Afin de bien répondre à la problématique, il faut parcourir le niveau 2 de *T* et supprimer les règles d'extension dont le support est nul.

Complexité :
Elle est identique à celle de l'algorithme *MajAjoutTransactions*. Le dernier parcours de *T* qui supprime les règles dont le support est nul a un coût linéaire en le nombre de nœud de *T*. Ce coût est négligeable.

3.4 Modification de transactions

La modification de transactions est décomposable en deux sous étapes. Une transaction modifiée est similaire à la suppression de cette transaction avant modification et à l'ajout de la transaction modifiée par la suite. La méthode que nous avons utilisée découle donc directement de ce choix. Elle consiste à appliquer successivement les algorithmes d'ajout et de suppression de transactions expliqués ci dessus. Nous ne présenterons pas d'exemple de modification de transactions de par le fait que les algorithmes mis en œuvre ont les mêmes effets que ceux liés à l'ajout et à la suppression de transactions. La démonstration d'une telle stratégie est triviale et liée aux démonstrations des algorithmes précédents. Quant à la complexité d'une telle stratégie à la vue de la complexité des algorithmes précédents nous pouvons dire qu'elle a pour ordre de grandeur celle des deux algorithmes précédents à un coefficient multiplicateur *k* près.

4 Expérimentations

De manière à évaluer et valider nos algorithmes de maintenance de connaissances extraites, nous avons mené différentes expérimentations que nous présentons dans cette section. Les algorithmes ont été implémentés en C++ en utilisant la bibliothèque GTL (Graph Template Library) comme dans le cas des algorithmes du chapitre précédent. Ces algorithmes ont été testés sur différents jeux de données synthétiques et réels. Nous avons choisi de présenter ici les tests relatifs à la base de données des navires car ceux-ci reflètent bien le comportement de nos algorithmes. Cette base de données contient 4215 voyages, représentés sous la forme d'une structure de profondeur au plus 4 (3,86 de profondeur moyenne). Le nombre de nœuds contenus en moyenne dans chaque structure est de 18,78 pour un maximum de 22.

4.1 Variation du support minimal

Dans un premier temps, nous avons mené des expériences de manière à examiner les temps de réponse des algorithmes permettant d'augmenter ou de diminuer la valeur du support minimal à partir des résultats d'une extraction de connaissances dans le cas normal et dans le cas généralisé. Nous opposons ces temps de calcul à l'algorithme d'extraction de connaissances du Chapitre III. Ainsi nous pouvons mesurer le gain de nos algorithmes de maintenance de connaissances extraites par rapport à une recherche de structures fréquentes depuis zéro.

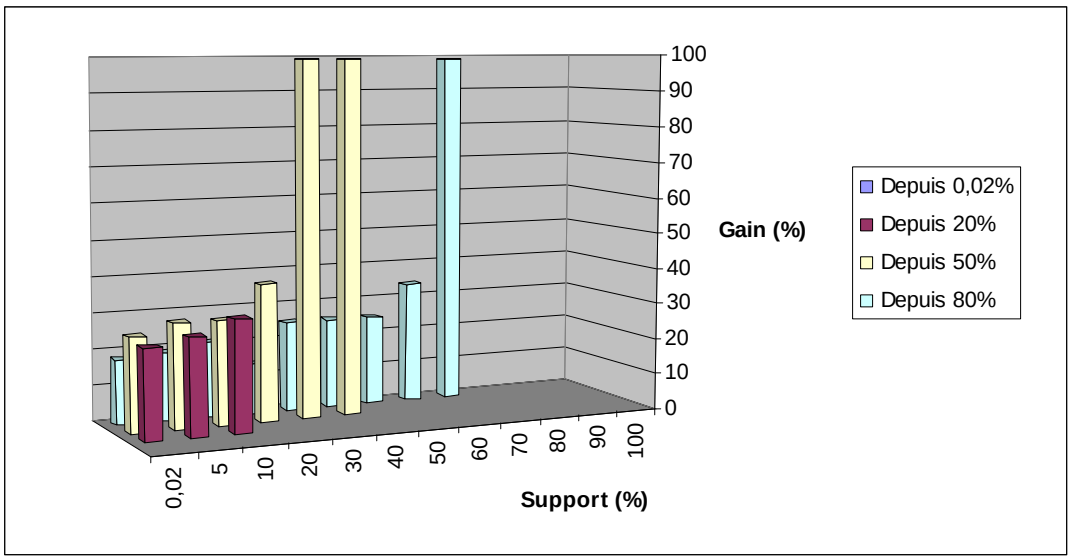


Figure 89 – Variation de support dans le cas normal

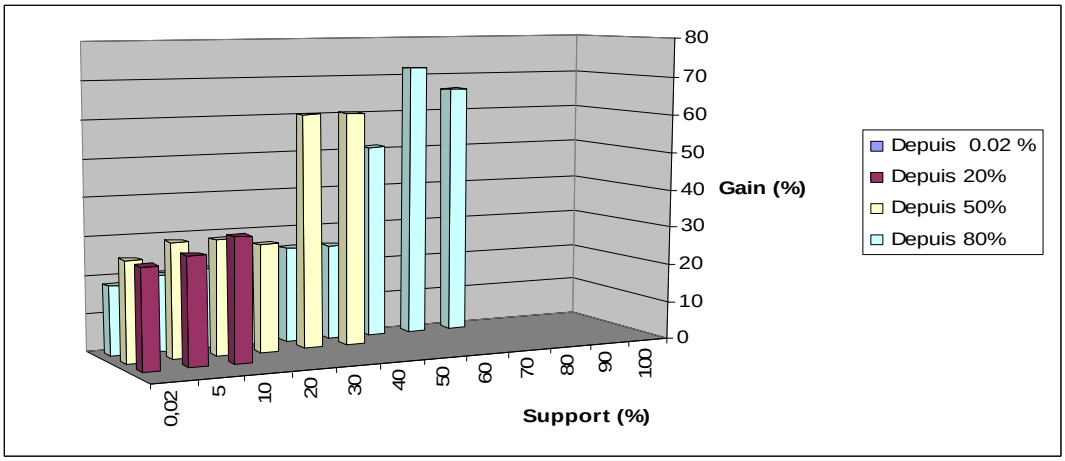


Figure 90 – Variation de support dans le cas généralisé

La Figure 89 illustre les résultats obtenus sur la base de données voyage en considérant le gain en pourcentage par rapport à un calcul depuis zéro dans le cas normal. L'axe des *x* correspond à une variation de support, l'axe des *y* représente le gain par rapport à un calcul depuis zéro en pourcentage. L'axe des *z* correspond aux différentes recherches initiales à partir desquelles la variation de support a été effectuée. L'idée principale est d'illustrer le pourcentage de gain en temps par rapport à une recherche depuis zéro. Pour cela, quatre recherches avec les valeurs de support : 0.02%, 20%, 50% et 80% ont été préalablement réalisées. Pour chacun de ces jeux nous avons mesuré le temps mis par les algorithmes incrémentaux lors de variations de support. Nous avons également mesuré le temps de recherche pour une version non incrémentale. La comparaison de ces résultats nous a permis d'évaluer le pourcentage de gain en temps des algorithmes incrémentaux. La Figure 90 est similaire mais concerne le cas généralisé. Nous n'avons pas représenté les cas ou la valeur du support minimal est augmentée, en effet sur un jeu de cette taille du fait de la linéarité de l'algorithme ($O(n)$), le gain est de 100% aussi bien dans le cas normal que dans le cas généralisé. Comme attendu, nous constatons que, de manière générale, le gain des algorithmes dépend de la distance entre le support initial et le support final : plus la distance est grande entre ces deux valeurs plus le gain est diminué. Ainsi le gain pour un passage de 50% à 40% que ce soit dans le cas généralisé ou non est supérieur au double d'un gain de passage de 50% à 20%. Ce constat était prévisible, dans la mesure où, dans ce cas, les arbres préfixés correspondants à des recherches pour 50% et 40% sont très proches, alors que pour 30% la structure préfixée contient un nombre important de différences nécessitant un plus grand nombre de calculs. Le deuxième constat prévisible réside dans la similarité du comportement de ces algorithmes dans les cas normal et généralisé. En effet les structures et les algorithmes utilisés comme nous l'avons montré dans le chapitre précédent sont quasiment identiques. Cette similarité au niveau des gains de temps est donc tout à fait normale.

4.2 Ajout de transactions

Les expériences considérées dans cette section ont pour objectif d'évaluer le temps de réponse de notre algorithme d'ajout de transactions dans le cadre d'une mise à jour pour le cas normal. Des résultats identiques ayant été obtenus dans le cas généralisé nous ne les présenterons pas ici. Tout comme la section précédente, nous opposons ces temps de calcul à l'algorithme d'extraction de connaissances du chapitre précédent. Ainsi nous pouvons mesurer le gain de nos algorithmes de maintenance de connaissances extraites par rapport à une recherche de structures fréquentes depuis zéro.

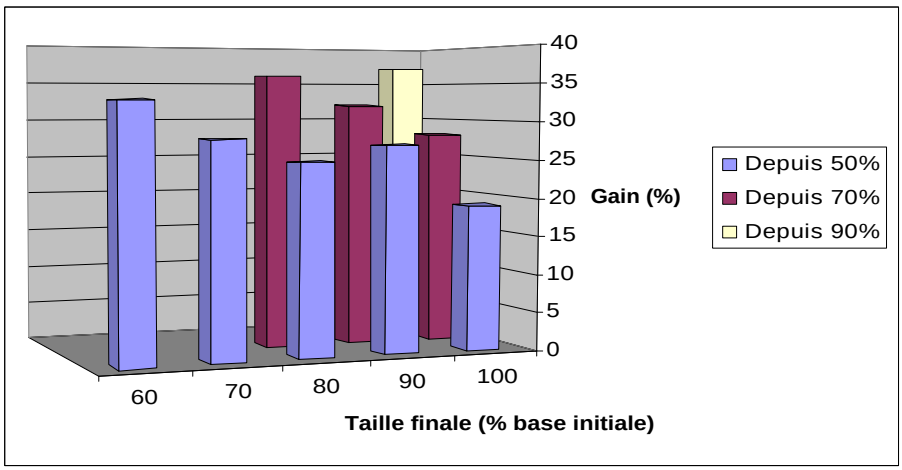


Figure 91 – Ajout de transactions pour un support minimal de 20%

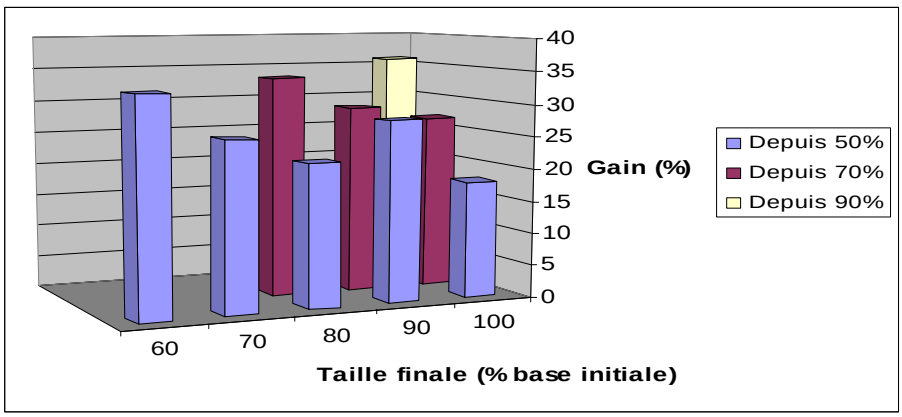


Figure 92 – Ajout de transactions pour un support minimal de 80%

La Figure 91 illustre les résultats obtenus sur la base de données voyages en considérant le gain en pourcentage par rapport à un calcul depuis zéro dans le cas normal pour un support minimal de 20%. L'axe des *x* correspond à une variation du pourcentage de la taille de la base de données voyages considérée. L'axe des *y* représente le gain par rapport à un calcul depuis zéro en pourcentage pour un support minimal de 20%. L'axe des *z* correspond aux différentes recherches initiales à partir desquelles la variation de la taille de la base a été effectuée. L'idée principale est d'illustrer le pourcentage de gain en temps par rapport à une recherche depuis zéro. Pour cela, trois recherches, sur respectivement 50%, 70% et 90% de la taille de la base, ont été préalablement réalisées. Pour chacun de ces jeux nous avons mesuré le temps mis par les algorithmes incrémentaux lors de variations de la taille de la base. Nous avons également mesuré le temps de recherche pour une version non incrémentale. La comparaison de ces résultats nous a permis d'évaluer le pourcentage de gain en temps des algorithmes incrémentaux. La Figure 92 est similaire mais concerne la valeur de support minimal de 80%. Dans la cadre de cette expérience, nous constatons que, de manière générale, le gain des algorithmes dépend, tout comme les variations de support minium, de la distance entre la taille de la base de données initiale et la taille de la base de données finale. Ainsi lorsqu'on on passe d'une base ayant une taille de 50% à une base ayant une taille de 80%, pour un support minimal de 20%, le gain est de 33%. Pour un support minimal de 80%, ce gain est de 31%. Dans le cas d'un accroissement d'une taille de 50% à 100% les gains sont respectivement de 18% et 17%. Nous pouvons conclure dans cette expérimentation que cet algorithme présente un gain positif quelque soit les valeurs choisies pour le test. Cela s'explique par le fait que dans ce jeu de données, les fréquents le sont à des seuils très

faibles. Ceci explique une faible influence du support minimal sur les courbes de gain. De plus le nombre de règles plus fréquentes qui engendre une grande partie de la complexité de l'algorithme de la section 3.2 est quasiment négligeable lorsqu'on ajoute des transactions sur cette base de données, d'où un gain positif. Cela ne sera pas le cas lors de la suppression de transactions.

4.3 Suppression de transactions

Les expériences considérées dans cette section ont pour objectif d'évaluer le temps de réponse de notre algorithme de suppression de transactions dans le cadre d'une mise à jour pour le cas normal. Des résultats identiques ayant été obtenus dans le cas généralisé nous ne les présenterons pas ici. Comme précédemment, nous opposons ces temps de calcul à l'algorithme d'extraction de connaissances du Chapitre III. Ainsi nous pouvons mesurer le gain de nos algorithmes de maintenance de connaissances extraites par rapport à une recherche de structures fréquentes depuis zéro.

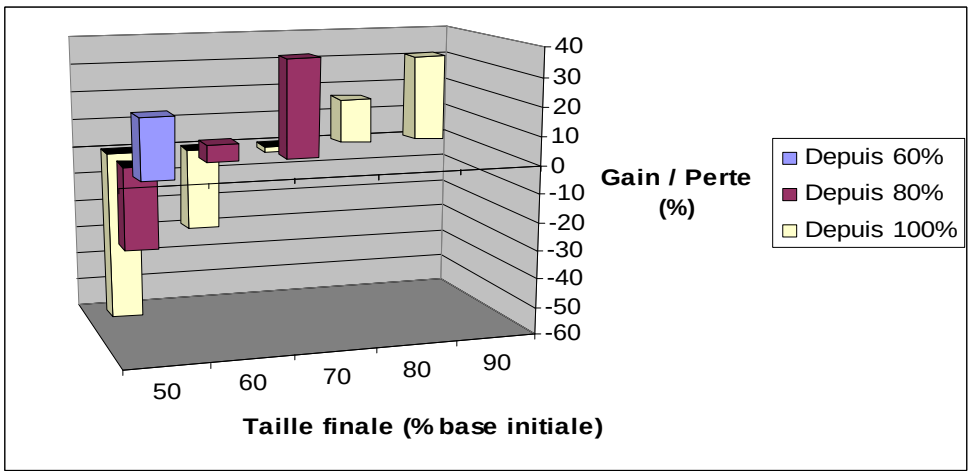


Figure 93 – Suppression de transactions pour un support minimal de 20%

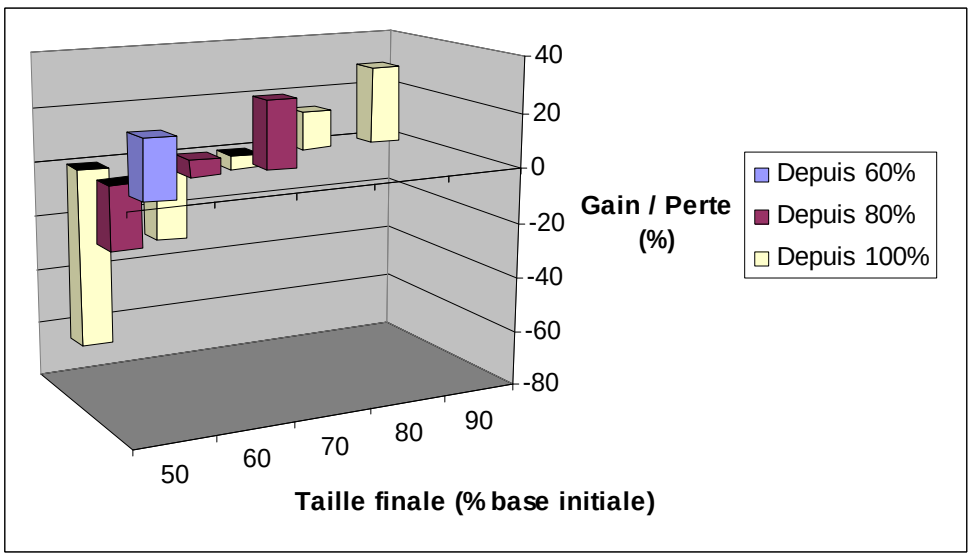


Figure 94 – Suppression de transactions pour un support minimal de 80%

La Figure 93 illustre les résultats obtenus sur la base de données voyages en considérant le gain en pourcentage par rapport à un calcul depuis zéro dans le cas normal pour un support de 20%. L'axe des *x* correspond à une variation du pourcentage de la taille de la base de données voyages considérée. L'axe des *y* représente le gain (ou la perte) par rapport à un calcul depuis zéro en pourcentage pour un support minimal de 20%. L'axe des *z* correspond aux différentes recherches initiales à partir desquelles la variation de la taille de la base a été effectuée. Le principe d'obtention des résultats est similaire à celui de la section précédente. La Figure 94 est similaire mais concerne la valeur de support minimal de 80%. Comme nous pouvions nous y attendre, lorsque

l'on examine la complexité de cet algorithme nous n'avons pas systématiquement de gain. Ainsi si nous observons le rapport gains/pertes depuis une taille initiale de 100%, nous constatons pour des tailles finales de 90% et 80% des gains respectivement de 31% et 16% pour un support minimal de 20% et de 29% et 14% pour un support minimal de 80%. Mais par contre pour des tailles finales de 70%, 60% et 50% des pertes respectivement de 2%, 29% et 60% pour un support minimal de 20% et de 5%, 29% et 65% pour un support minimal de 80% sont constatées. Pour les mêmes raisons que dans le cadre de l'ajout de transactions, l'influence de la valeur du support minimal sur les gains et pertes dans le cadre de cette expérience est conforme aux résultats obtenus. L'apparition de pertes prévisibles au regard de sa complexité dans le cas de l'utilisation de cet algorithme pose le problème de son utilisation. Ce problème sera débattu dans la partie discussion de ce chapitre.

5 Discussion

Dans cette section nous revenons sur l'existence de possibilités de pertes lors de l'utilisation des algorithmes de mise à jour concernant l'ajout et la suppression de transactions. Comme nous avons pu le constater lors des expérimentations, il se présente des situations dans lesquelles l'utilisation de tels algorithmes présente un temps d'exécution supérieur à une extraction de connaissances depuis zéro. Ce constat nous amène à nous interroger sur la nécessité de construire un « oracle » qui nous permettrait de déterminer quand ces algorithmes sont rentables ou quand un calcul depuis zéro est plus intéressant. La construction d'un tel algorithme d'aide à la décision nécessite la mise en place d'une métrique complexe sur laquelle s'appuierait l'algorithme pour prendre cette décision. Une telle métrique n'est pas évidente à mettre en œuvre. Cependant nous pouvons énoncer ici quelques pistes pour la construire. Tout d'abord le point clé de cette métrique sera basé sur les nœuds qui avant l'ajout ou la suppression d'un incrément demeurent fréquents. En effet, ces nœuds sont source de coût de par le fait qu'il faille supprimer tous les descendants de ceux-ci qui ont été générés par des règles plus fréquentes. Un deuxième élément de cette métrique doit prendre en compte ce nombre de règles fréquentes qui deviennent non fréquentes. Un troisième élément de cette métrique peut être la taille de l'incrément ajouté ou supprimé. Enfin une fois cette métrique construite, une question perdure : quelle est la valeur à partir de laquelle faut il ou ne faut il pas utiliser l'algorithme incrémental ? Les tests que nous avons réalisés tendent à montrer que cette valeur doit être déterminée en fonction du jeu de données. Intuitivement elle est intrinsèque à celui-ci, dans sa méthode de calcul, elle doit donc tenir compte des spécificités de ce jeu. Une étude approfondie d'une telle métrique ou de sa généralisation présenterait un grand intérêt dans le domaine de l'utilisation d'une approche incrémentale utilisant le concept de la bordure négative. Elle permettrait de définir la limite de telles approches et surtout de mieux définir les cas où il est plus intéressant de partir depuis zéro de ceux où l'utilisation de cette bordure est rentable pour la mise à jour.

Une deuxième conséquence issue des résultats expérimentaux a été de se replonger dans l'algorithme lui-même pour déterminer de manière plus précise le point faible (celui qui engendre le coût) de celui-ci. Notre conclusion sur ce domaine est la méthode que nous avons employée pour le stockage des règles. En effet lorsque nous cherchons à déterminer si un nœud qui demeure fréquent après ajout ou suppression de l'incrément n'a pas été généré par une règle qui elle n'est plus fréquente, nous parcourons des listes contenant ces règles d'extension. Cette opération est répétée pour chaque nœud fréquent qui le demeure et elle constitue un des éléments coûteux de l'algorithme. Afin d'améliorer le coût de l'algorithme, il faudrait remplacer le parcours de ces listes par une structure de hachage ou par un marquage de la règle génératrice sur chaque nœud pour répondre rapidement à la question : « quelle règle a généré ce nœud ? ». Néanmoins ce type d'amélioration n'a pour conséquence que de diminuer le coût global de l'algorithme. Il est évident que la mise en œuvre d'un « oracle » basé sur les éléments précisés ci-dessus permettrait de garantir un coût de l'algorithme de mise à jour inférieur à une extraction de connaissances depuis zéro.

Chapitre V - Le système AUSMS et ses applications

Les différents algorithmes décrits dans les chapitres précédents ont été intégrés dans un système nommé AUSMS (Automatic Update Schema Mining System) [LaPo03, LaTe03a]. Dans ce chapitre, nous en présentons l'architecture générale ainsi que les choix d'implémentation. Ce système a été utilisé dans différents domaines d'application : analyse du comportement d'utilisateurs sur des serveurs Web et aide à la sélection de vues dans le cas de données semi structurées.

Le chapitre est organisé de la manière suivante. Dans la section 1, nous présentons l'architecture générale du système AUSMS et décrivons chacun de ces composants. La section 2 s'intéresse plus particulièrement au Web Usage Mining, i.e. l'analyse du comportement d'utilisateurs sur un ou plusieurs serveurs Web. Dans cette section, nous précisons la problématique générale et présentons deux nouvelles problématiques associées à une analyse plus fine des comportements et aux tendances des utilisateurs au cours du temps. Un aperçu des travaux autour du Web Usage Mining est également proposé. L'aide à la sélection de vues dans des données semi structurées est présentée dans la section 3. Nous montrons, dans cette section, comment les résultats obtenus avec nos algorithmes peuvent être intégrés dans le modèle de vues VIMIS et plus particulièrement comment notre approche peut être utilisée pour la construction de dataguide. Enfin, nous concluons par une discussion concernant le système AUSMS et les domaines d'application que nous avons expérimentés.

1 Le système AUSMS

Le but du système AUSMS est de proposer un environnement de découverte et d'extraction de connaissances pour des données semi structurées depuis la récupération des informations jusqu'à la mise à jour des connaissances extraites. Ces principes généraux sont illustrés dans la Figure 95. Ils sont similaires à un processus classique d'extraction de connaissances comme nous l'avons présenté dans le chapitre d'introduction.

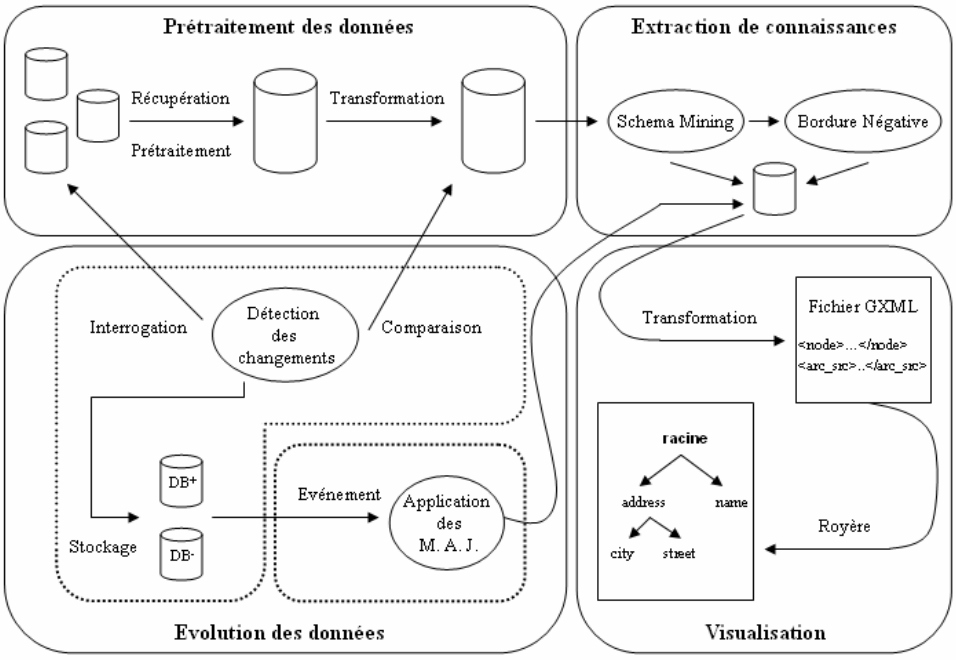


Figure 95 - Architecture générale

La démarche se décompose en trois phases principales. Tout d'abord à partir de fichiers de données semi structurées brutes, un prétraitement est nécessaire pour éliminer les données inutiles et en assurer leur transformation. Dans la seconde phase, un algorithme d'extraction de connaissances est utilisé pour extraire les sous structures fréquentes. De manière à permettre la maintenance des connaissances extraites, les informations obtenues lors de la phase d'extraction sont maintenues dans une base de données. Enfin l'exploitation des résultats obtenus par l'utilisateur est facilitée par un outil de visualisation des sous structures fréquentes.
Nous allons maintenant détailler chacun des différents modules qui composent le système en montrant leurs objectifs ainsi que la manière dont ils sont reliés. Nous présenterons, pour chaque module, les choix d'implémentations effectués au sein de notre prototype.

1.1 Prétraitement des données

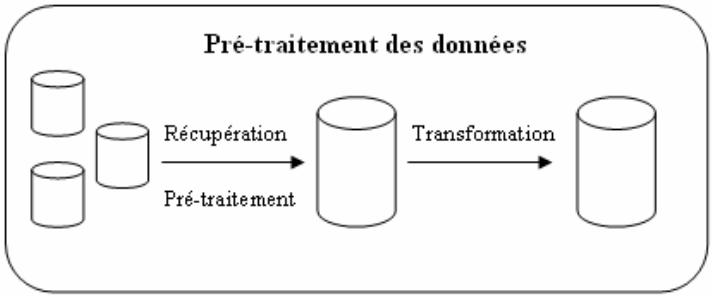


Figure 96 - Le prétraitement des données

Le premier module de AUSMS est consacré au prétraitement des données. Son rôle principal consiste à gérer l'accès aux différentes sources de données. La première partie de ce module est chargée de récupérer les informations des différentes sources. Ensuite, une étape de filtrage est réalisée de manière à éliminer les données qui ne sont pas utiles pour l'analyse. Cette étape est réalisée dans notre système par un parseur qui transforme les données brutes issues des sources en données prétraitées.

De manière à clarifier le fonctionnement de ce parseur, considérons l'exemple suivant :

Exemple 49 :

Considérons des données issues de documents HTML d'un site relatif aux navires dans les eaux canadiennes. Sur ce site, chaque navire présente des informations générales sur le navire, sur son immatriculation, sa construction, son enregistrement et sa description. Etant donné que nous sommes principalement intéressés par les informations uniquement relatives aux navires, les parties du document qui concernent la navigation (liens hypertextes) ou bien les images doivent être supprimées. Dans la Figure 97 nous voyons les informations pour le navire « A. Smithers », un filtrage de ce document consiste donc à éliminer de celui-ci les informations telles que : les différents liens hypertextes, les différents boutons de navigation, le drapeau canadien... Ensuite selon le point de vue de l'utilisateur, nous pouvons conserver ou non une partie des informations. Par exemple, nous pouvons nous intéresser à la totalité des informations sur un navire, ou bien seulement aux informations relatives à la construction de celui-ci ou encore à celles correspondant à sa description.

Un tel outil peut donc être utilisé pour prendre en compte le point de vue de l'utilisateur et certaines informations peuvent être éliminées durant cette étape car considérées comme inutiles. Dans notre système ce parseur est implémenté sous la forme d'un analyseur syntaxique.

Une fois les sources prétraitées, la seconde étape consiste à transformer chaque document en arbre. A chaque arbre nous associons un identifiant qui servira de clé primaire. A l'aide de l'algorithme *TreeToSequence* présenté dans le Chapitre III, chaque arbre est transformé en séquence qui sera stockée dans une base de données *DB*. Cette étape de transformation est représentée dans la Figure 98.

Exemple 50 :

Nous pouvons voir sur la Figure 99, un exemple de représentation d'un navire décrit à gauche. En haut à droite, nous avons représenté ce même document après une étape de filtrage sous la forme d'un arbre. La séquence correspondante est décrite en bas à droite. Par la suite, l'étape de transformation lui associe un identifiant et le stocke dans DB.

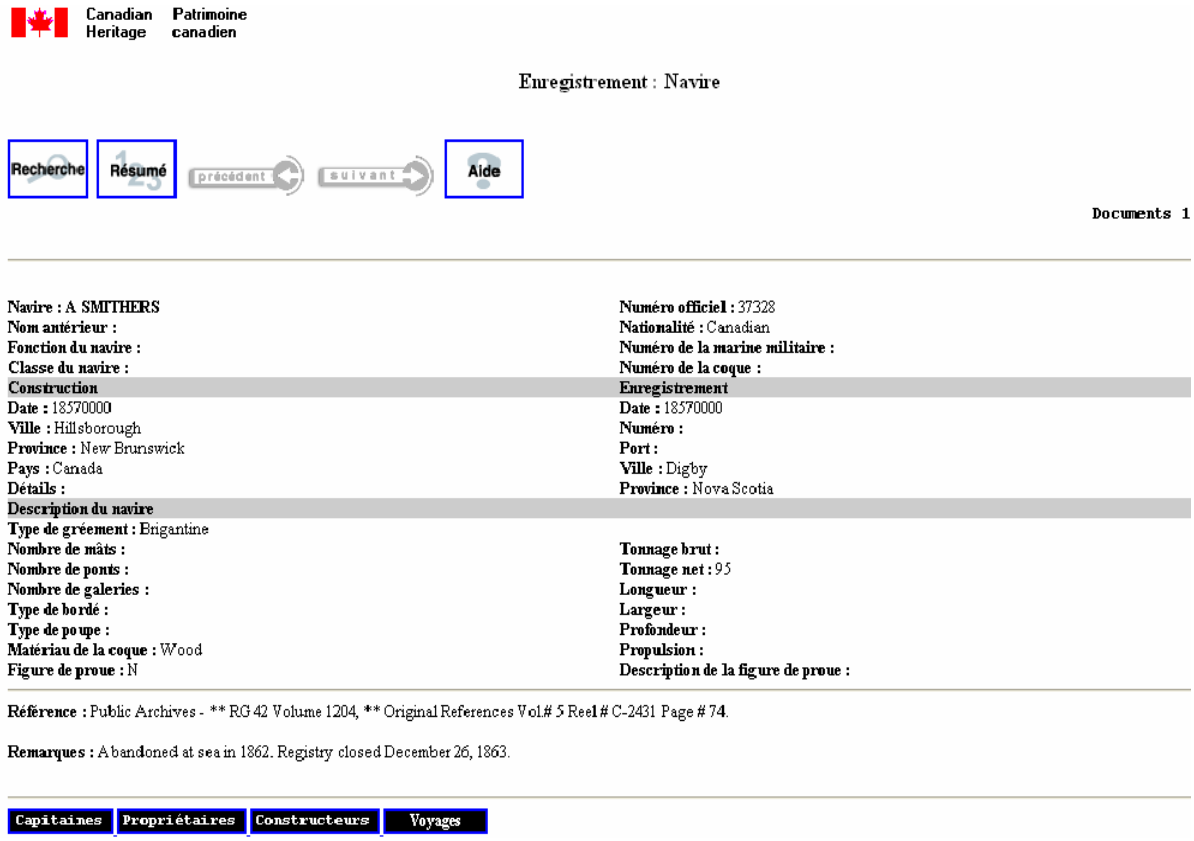


Figure 97 - Un exemple de navire

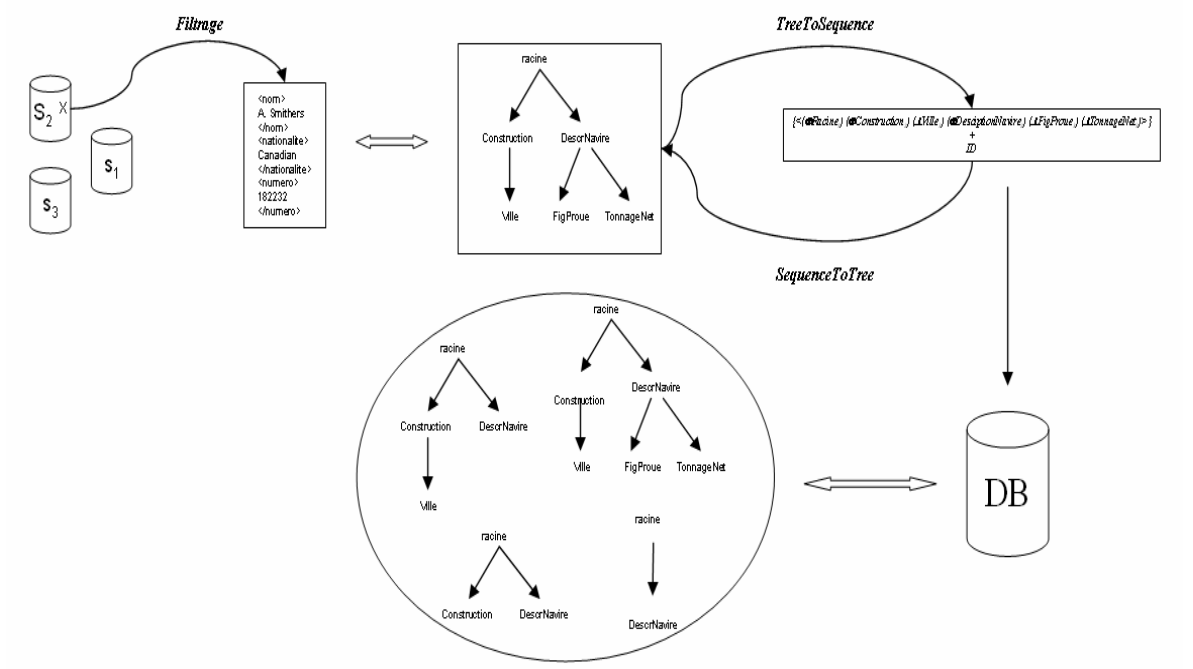


Figure 98 - La transformation

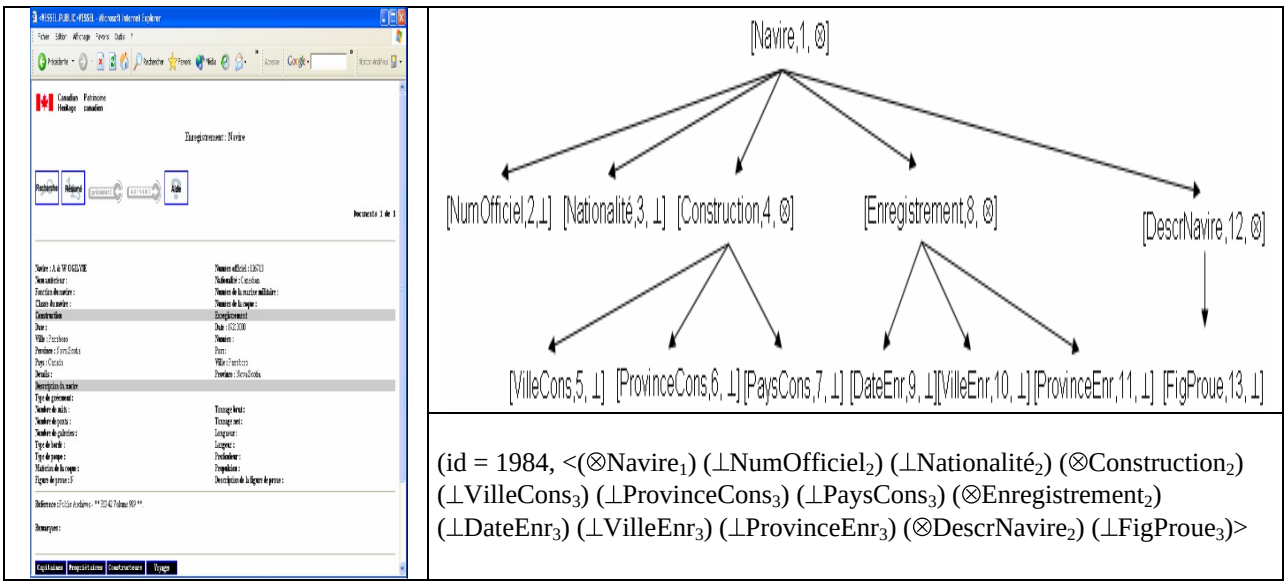


Figure 99 – Un exemple de transformation

Figure 100 - Les informations relatives à une source

Au sein d'AUSMS, à chaque source de données est associé un ensemble d'informations. Par exemple, nous pouvons voir, dans la Figure 100, une source sélectionnée via le menu « Source » et ses informations. Dans cette figure nous nous intéressons à la source de données relative aux capitaines de la base de données canadienne. Les informations sur cette source sont diverses : la localisation de celle-ci au niveau local (*répertoire de la source*), le nom du fichier à utiliser pour la phase d'extraction de connaissances (*list_capi*), la date à laquelle les informations concernant cette source ont été incorporées dans l'environnement AUSMS (*date de création*), la date de la dernière mise à jour de cette source (*date de mise à jour*). Des informations supplémentaires calculées à partir des sources apparaissent également : le nombre d'éléments de la collection (*Nombre de tid*), la profondeur maximale des structures (*Profondeur maximale*) et le nombre d'éléments maximum contenus dans une structure (*Nombre d'éléments maximum*). Lors de la première utilisation d'une source, l'information « aucun calcul effectué sur cette source » est affichée. Ultérieurement, i.e. après traitement des données, la liste des différents calculs (valeurs de support minimal) est affichée de manière à faciliter son accès à l'utilisateur.

1.2 Extraction de connaissances

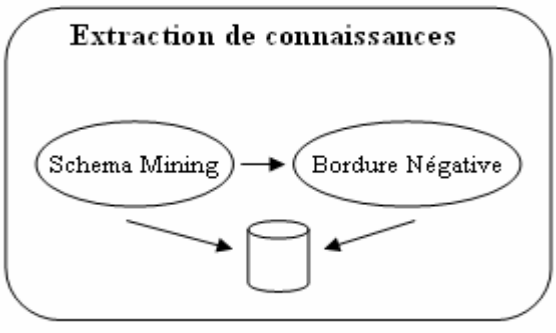


Figure 101 - L'extraction des connaissances

Le deuxième module du système AUSMS a pour objectif l'extraction de connaissances sur les éléments de la base de données issus de la phase précédente ainsi que le calcul de la bordure négative. Les résultats des extractions et du calcul de la bordure négative sont conservés dans une base de données comme le montre la Figure 101.

Dans la suite de cette section nous montrons comment les sous arbres fréquents sont recherchés ainsi que la manière dont est mise à jour la bordure négative.

Figure 102 - La méthode manuelle

Nous proposons à l'utilisateur deux méthodes pour la recherche de sous arbres fréquents : la première est manuelle et la seconde automatique.

Dans la méthode manuelle, comme le montre la boîte de dialogue correspondante de la Figure 102, l'utilisateur spécifie une valeur de support pour laquelle il désire effectuer cette recherche. Si aucun calcul n'a été réalisé précédemment sur cette source, le système exécute alors l'algorithme d'extraction de connaissances. Autrement, comme nous l'avons expliqué précédemment, un appel à l'algorithme de mise à jour incrémental qui gère les variations de support est réalisé. A l'issue de cette recherche, les structures fréquentes obtenues (ainsi que les éléments de la bordure négative) sont conservées dans des fichiers au format GXML de façon à être réutilisables ultérieurement (C.f. section 1.4).

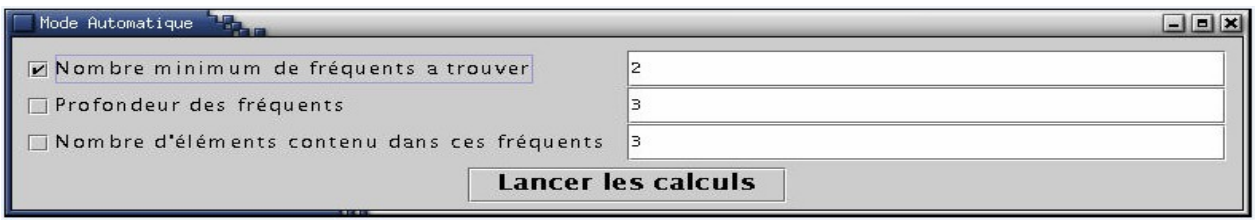


Figure 103 - La méthode automatique

Nous mettons également à la disposition de l'utilisateur un système de recherche automatique qui admet trois paramètres combinables entre eux sous la forme de conjonction logique. La Figure 103 illustre ces différents paramètres : le nombre minimal de fréquents à trouver, la profondeur des fréquents, le nombre d'éléments contenus dans ces fréquents. L'objectif est de trouver la valeur de support maximale remplissant ces conditions. Ainsi il devient possible de demander au système de chercher, par exemple, le support maximal pour lequel il existe au moins *n* fréquents, ou bien de rechercher en sélectionnant les deux premières cases à cocher, le support pour lequel il existe au moins *n* fréquents de profondeur supérieure ou égale à *m*. L'algorithme automatique procède par recherche dichotomique afin de trouver le support maximal répondant à ces conditions. Pour effectuer les calculs, le système utilise l'algorithme d'extraction depuis zéro si aucune information n'est disponible, autrement il tire profit de la version incrémentale qui permet d'obtenir plus rapidement les différents résultats.

Exemple 51 :

La Figure 104 illustre le résultat après une recherche automatique sur la base de données des capitaines. Plusieurs supports ont été calculés : 50, 51, 52, 53, 54, 55 et 60. Cette recherche automatique a abouti, comme le montre la boîte de dialogue, et nous informe qu'un support répond aux contraintes imposées. Les résultats de ces calculs sont conservés et l'utilisateur pourra s'il le souhaite les consulter ultérieurement. Le support répondant aux conditions choisies est de 50% dans notre exemple.

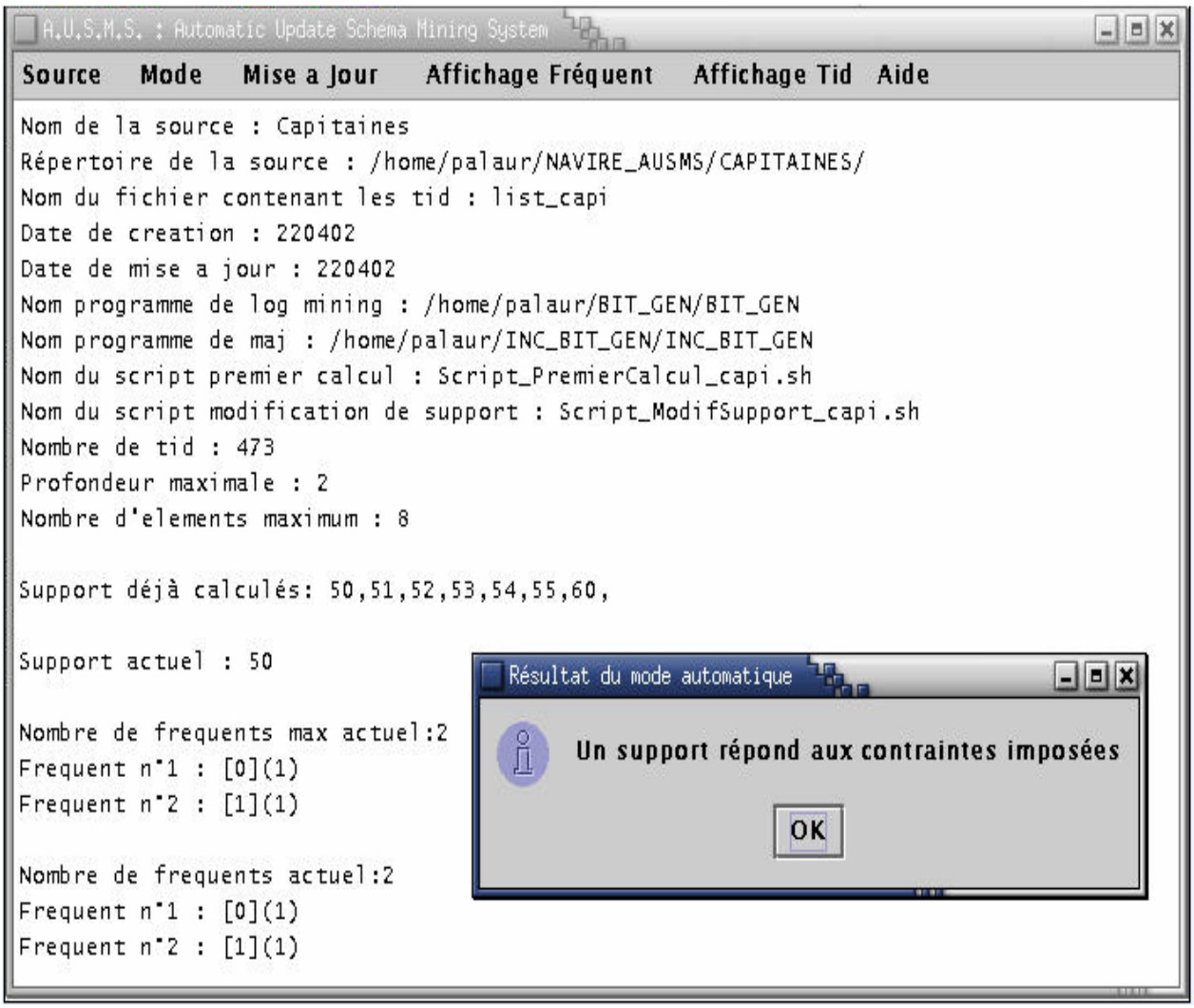


Figure 104 - Un exemple de recherche automatique

Nous présentons, à présent, comment sont prises en compte les évolutions des données dans AUSMS.

1.3 Evolution des données

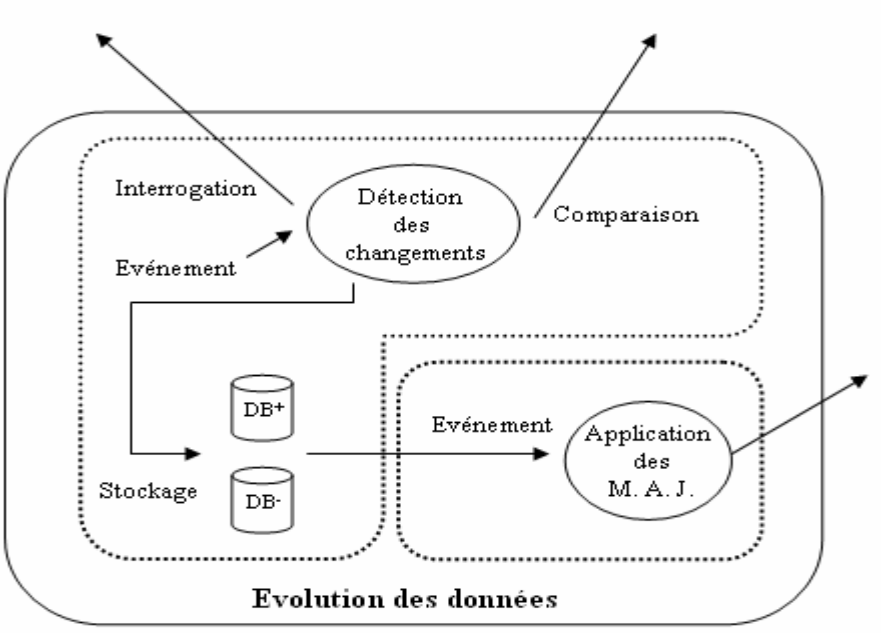


Figure 105 - Evolution des données

Nous avons expliqué dans le chapitre précédent, comment le calcul de la bordure négative permet de tenir compte des mises à jour et de maintenir la connaissance extraite. Nous avons également vu dans la section précédente son utilisation dans le cas du processus automatique d'extraction ou de l'approche manuelle, i.e. si des extractions ont déjà été réalisées et qu'il y a une modification de la valeur du support. Dans cette section, nous présentons la manière dont les modifications sur les sources sont prises en compte. Le module d'évolution des données est en fait divisé en deux sous modules comme l'illustre la Figure 105 : détection et application des évolutions des sources.

Nous présentons, par la suite, chacun des ces modules.

1.3.1 Détection des évolutions

Ce module a pour objet la détection des changements au niveau des sources. Cette détection, dans le système AUSMS est réalisée par l'intermédiaire d'un agent qui est déclenché par l'utilisateur.

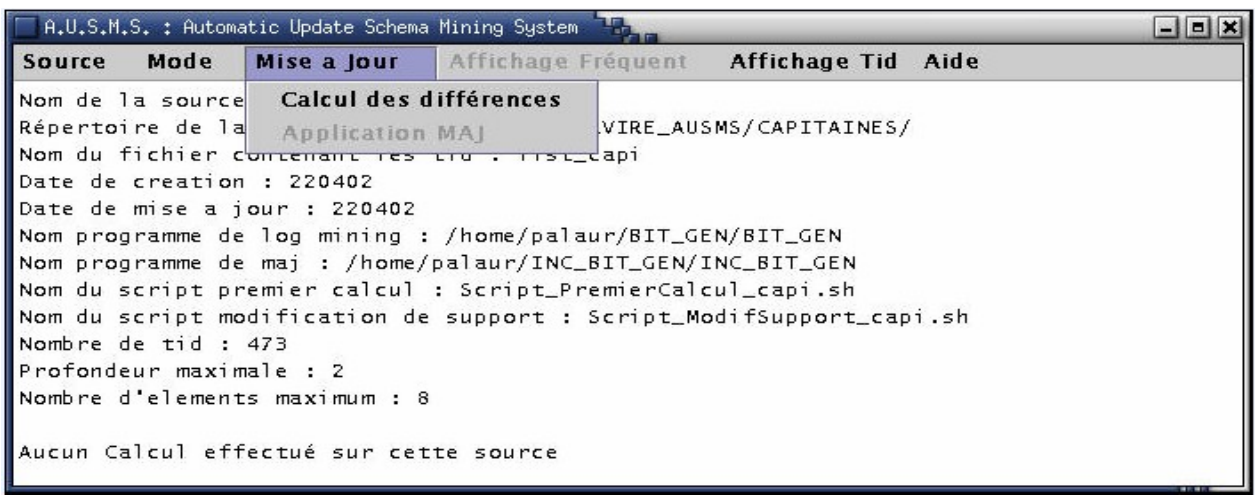


Figure 106 - Déclenchement de l'agent de détection des évolutions

La Figure 106 illustre, au travers du menu « Mise à Jour », le déclenchement de l'agent de détection des changements. En effet, avant d'effectuer une mise à jour réalisée par le second module, une ou plusieurs détections des évolutions peuvent être effectuées. Bien entendu, pour éviter des résultats erronés, toute détection ne peut être déclenchée lorsqu'une détection est déjà en cours. Le principe général est le suivant. Les différentes sources de données sont comparées. Pour chaque document de chaque source, identifié de manière unique, nous traitons l'information, i.e. nous appliquons les mêmes traitements que pour les étapes précédentes (nettoyage, application de *TreeToSequence*). S_{new} représente le document issu de ce prétraitement. Pour un même identifiant, nous recherchons le document existant dans la base, S_{old}. Si la source de données a été modifiée, i.e. $S_{new} \neq S_{old}$, nous ajoutons S_{old} dans *db-* (cette source n'est plus cohérente donc elle est supprimée) et nous ajoutons S_{new} dans *db+* (étant donné que la source précédente a été supprimée, nous la réinsérons dans la base). Autrement, s'il n'existe pas de correspondant dans la base, i.e. s'il n'existe pas d'identifiant dans la base, nous ajoutons S_{new} dans *db+*. De la même manière, si, pour une source, il existe un document qui existait précédemment mais qui n'apparaît plus dans la base mise à jour, nous ajoutons ces éléments dans *db-*. A l'issue de ce traitement, nous disposons d'une base d'éléments ajoutés *db+* et d'une base d'éléments supprimés *db-*.

1.3.2 Application des évolutions

Le dernier module du système a pour objectif de répercuter les modifications des données sources sur les connaissances stockées dans la base. Pour permettre de répercuter ces modifications, chaque recherche d'extraction est stockée sous la forme d'une paire <*valeur de support*, *résultat associé*> où les résultats associés correspondent à des fichiers décrivant d'une part les sous arbres fréquents et d'autre part les bordures négatives associées. Ces informations sont stockées sous la forme d'un arbre. A partir des informations recueillies par le module précédent nous construisons deux bases de données : la première contient les informations ajoutées (*db+*) et la seconde contient les informations supprimées (*db-*). A partir de ces deux bases et des résultats précédents, la répercussion des mises à jour est réalisée en appliquant l'algorithme de maintenance décrit dans le chapitre précédent. A l'issue de cette phase, déclenchée dans notre système par la sélection de "Application MAJ" du menu Mise à jour (C.f. Figure 106), les évolutions des sources de données et leurs répercussions sur les connaissances déjà extraites sont prises en compte.

1.4 Visualisation

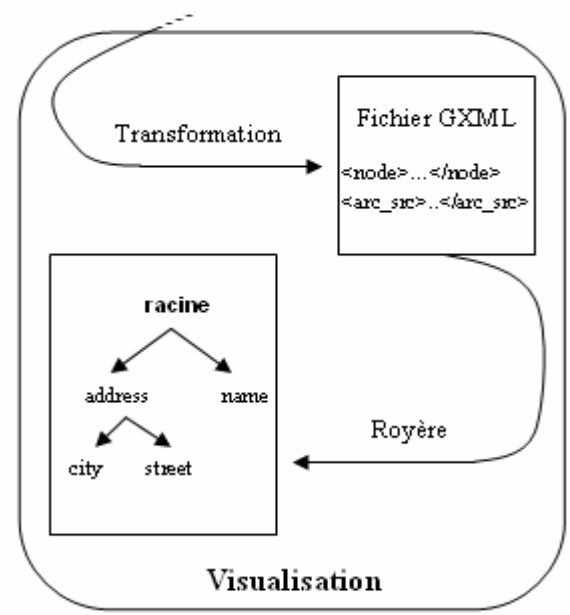


Figure 107 - Visualisation

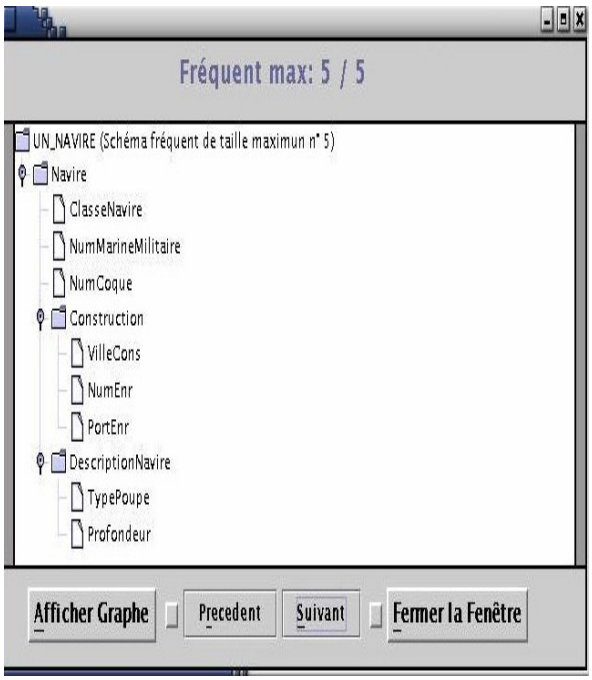
Alors que les modules précédents sont chargés d'extraire et de maintenir des sous arbres fréquents, ce dernier module permet de visualiser les structures extraites et offre un formalisme pour les décrire.
Nous utilisons, pour décrire nos graphes, GraphXML [HeMa00] qui est un langage de description de graphe en XML. En outre, ce langage offre à l'utilisateur la possibilité de rajouter de nombreuses informations aux graphes

manipulés. Dans nos applications, nous avons ainsi ajouté des informations concernant le nom de la base de données manipulées, la date des calculs, le nombre de transactions, la valeur du support...

Au sein d'AUSMS, les résultats sont représentés de différentes façons. La première approche est une représentation sous la forme d'un « JTREE », i.e. une classe Java définie pour représenter les arbres. Par exemple, la Figure 108 (en haut à gauche) illustre une structure fréquente extraite sous la forme d'un JTREE. Pour générer ce JTREE, nous convertissons dans un premier temps les structures fréquentes ou non sous la forme d'une description GraphXML (C.f. Figure 108, en haut à droite) puis un analyseur lexical adapté au traitement des fichiers GraphXML génère l'objet de type JTREE correspondant.

La représentation des arbres sous la forme d'un langage dérivé d'XML, nous permet de proposer, comme nous pouvons le voir sur la Figure 108, une troisième représentation des arbres extraits. Celle-ci est définie à l'aide d'une application ROYERE. Cette dernière est basée sur le projet "Graph Visualisation Framework" [MaHe00] dont l'objectif est de proposer un ensemble de packages Java 2 pour définir des applications de manipulation ou de visualisation de graphes.

Les prochaines sections illustrent deux cas d'utilisation différents du système AUSMS : le premier concerne l'analyse du comportement d'utilisateurs de site Web. Le second est une illustration de l'utilisation d'AUSMS pour aider à sélectionner des vues dans une base de données semi structurées.




```
<?xml version="1.0"?>
<!DOCTYPE GraphXML SYSTEM "file:GraphXML.dtd" [
<!ENTITY % admissibleProperties "
support CDATA #IMPLIED
nombreTid CDATA #IMPLIED
dateCalcul CDATA #IMPLIED
nomBase CDATA #IMPLIED ">
]>
<GraphXML>
<graph isDirected="true" isForest="true"
isAcyclic="true" "referredlayout=
"ReingoldTilford" id="Frequent" version="1.0"
vendor="ausms@lirmm.fr" isPlanar="true">
<support>85</support> <nombreTid>728</nombreTid>
<dateCalcul>031002</dateCalcul>
<nomBase>navire</nomBase>

<node name="1">
<label>Navire</label>
</node>
<node name="2">
<label>ClasseNavire</label>
</node>
<edge source="1" target="2">
</edge>
        … … … … …
<node name="12" class="std">
<label>Profondeur</label>
</node>
<edge source="10" target="12">
</edge>
</graph> </GraphXML>
```

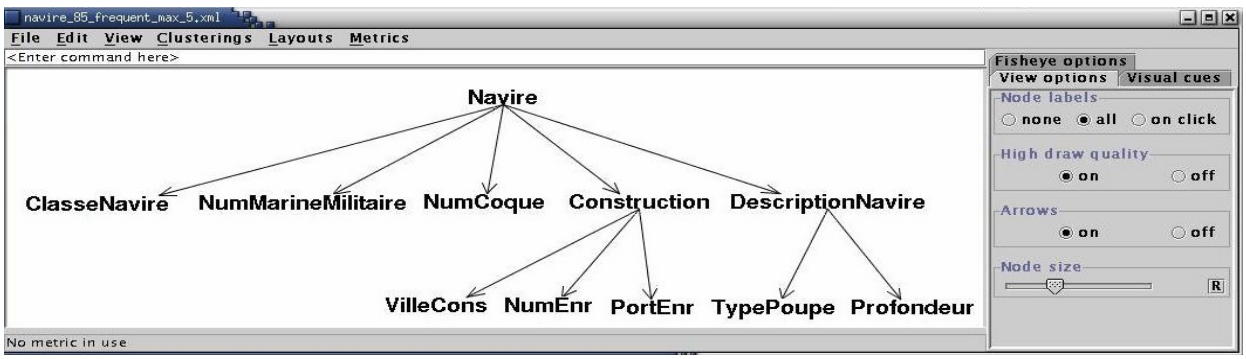


Figure 108 – Différentes visualisations possibles

2 AUSMS un système pour l'étude du Web Usage Mining et des tendances

Comme nous l'avons expliqué dans la section précédente le système AUSMS est modulaire. Dans cette section, nous montrons comment nous l'avons adapté à l'étude de l'analyse du comportement des utilisateurs d'un ou de plusieurs serveurs Web, i.e. le Web Usage Mining.

Dans une première partie, nous rappelons la problématique de l'analyse de comportement et nous l'étendons pour affiner l'analyse et pour étudier les tendances des utilisateurs au cours du temps. Nous présentons dans la section 2.2, les différents travaux existants dans ces domaines. Enfin, dans la section 2.3, nous décrivons le système AUSMS-Web ainsi que les expériences menées.

2.1 Les problématiques du Web Usage Mining

De nos jours, la popularité et l'interactivité du World Wide Web génèrent un volume sans cesse croissant de données. Depuis quelques années, de nombreuses recherches ont été menées pour analyser ces données et portent le nom de Web Mining. Un panorama complet des systèmes et approches du Web Mining est présenté dans [KoBl00]. Le Web Mining se définit comme l'utilisation de techniques de fouilles de données pour découvrir et extraire l'information issue de documents Web. Ce domaine est généralement divisé en deux sous domaines : le Web Content Mining qui s'intéresse principalement aux contenus des données d'un site Web selon différents points de vue (sémantique, structurel, ...) et le Web Usage Mining qui s'intéresse aux parcours des utilisateurs sur un site Web. Les travaux que nous avons menés se situent dans la seconde catégorie.

2.1.1 Analyse de comportements

Du fait de la popularité du World Wide Web, de très grandes quantités de données comme l'adresse des utilisateurs ou les URL demandées sont automatiquement récupérées par les serveurs Web et stockées dans des fichiers access log.

Spécifiée par deux organismes le CERN et le NCSA [HoOd00], chaque entrée contient différents champs, décrits dans la Figure 109.

Colonne	Exemple
Date	2003-03-20
Heure	00 :26 :30
IP du client	172.197.177.126
UserName	...
Service	W3SVC33
Nom du serveur	CLM01
IP du serveur	10.2.1.201
Port du serveur	443
Méthode	POST ou GET
Ressource accédée	/pub/index/hml/index.html
URL Query	…
Status HTTP	200 Voir : http://www.w3schools.com/html/html_httpmesssages.asp
Status Win32	0
Octets envoyés	1128
Octets reçus	568
Temps pris (ms)	2031
Version du protocole	http/1.1
Serveur	Lirmm.lirmm.Fr
Navigateur du client	Mozilla/4.0+(compatible ;+MSIE+6.0 ;+Windows+NT+5.1
Cookie	ASPSESSIONIDCCQAAQDR=JDHJKFLAFANHJNNICLPGHFFG
Referrer	http://www.lirmm.fr/~laur/Cv.html

Figure 109 - Champs d'un fichier de log

La Figure 109 montre que chaque entrée peut globalement être répartie de la manière suivante :

- identification : date, service, machine physique, méthode d'accès à la ressource,
- ressource demandée avec le résultat de la demande,
- informations sur le navigateur du client,
- cookies du client envoyés dans le flux http,
- ressource depuis laquelle l'utilisateur arrive.

Exemple 52 :

Les lignes ci-dessous décrivent un extrait d'un fichier log d'un site de e-commerce dans le domaine des télécommunications.

```
2003-05-22 01:51:36 80.8.55.57 – W3SVC13 CLM01 10.2.1.201 80 POST
/Cob/lesminutes/Offer/TYPE_9/OfferDelivery.asp – 200 0 112 924 7109
http/1.1 www.lesminutes.comMozilla/5.0+(Windows ;+U;+Windows+NT+5.1;+en-
US;+rv:1.3a)+Gecko/20021212 ASPSESSION IDASTASADT =
ICDKEPIBDEOFHAMPPIHOFFMB http://www.lesminutes
.com/Cob/lesminutes/Offer/TYPE_9/OfferDelivery.asp

2003-05-22 01:51:36 80.8.55.57 – W3SVC13 CLM01 10.2.1.201 80
GET/vxx_inc/Templates/s1/t1/frame/bg1/transp_XX.gif– 200 0 289 574 47
http/1.1 www.lesminutes.com Mozilla/5.0+ (Windows;+U;+Windows+NT+5.1;+en-
US;+rv:1.3a)+Gecko/20021212ASP SESSIONIDASTASADT =
ICDKEPIBDEOFHAMPPIHOFFMB;+GEN=GIFT%FLIMIT
=3http://www.lesminutes.com/Cob/lesminutes/Offer/TYPE_9/OfferDelivery.asp

2003-05-22 01:55:15 81.56.67.125 – W3SVC13 CLM01 10.2.1.201 80
POST/Redirect/CompteExistant.asp – 200 0 112 616 1218 http/1.1
www.lesminutes.comMozilla/4.0+(compatible;+MSIE+6.0;+Windows+NT+5.1;+FREE)E
MIN=CLI1=0663285347;+TD=ORDER=NA&AFFPROG=NA&LEAD=NA
```

L'analyse de tels fichiers offre des informations très utiles pour, par exemple, améliorer les performances, restructurer un site ou même cibler le comportement des clients dans le cadre du commerce électronique. La problématique du Web Usage Mining consiste à s'intéresser au problème de la recherche de motifs comportementaux des utilisateurs à partir d'un ou plusieurs serveur Web afin d'extraire des relations entre les données stockées. Soit DB_{log} une base de données générée à partir d'un fichier log dont les éléments sont des couples (*id*,*navigation*) où *id* est un identifiant unique d'un utilisateur et *navigation* correspond au parcours de cet utilisateur sur le site Web considéré. Soit *minSupp* une valeur de support minimal spécifiée par l'utilisateur. La problématique classique du Web Usage Mining consiste à déterminer quels sont les comportements de DB_{log} qui ont une fréquence supérieure ou égale à *minSupp*. Cette problématique est similaire à un processus classique d'extraction de connaissances et de ce fait implique la résolution de problèmes similaires : prétraitement des données, extractions des connaissances et exploitation des résultats.

Nous complétons cette problématique par l'analyse des parcours de manière plus fine en fonction des points d'entrée. En effet, dans le cas de la problématique classique, les comportements correspondent à ceux qui sont majoritairement les plus grands. Cependant, pour l'utilisateur final, une analyse plus fine, à partir des différents points d'entrée du site, offrira des informations et des connaissances supplémentaires pour maintenir son site. Plus formellement, nous partitionnons DB_{log} en DB_{log1}, DB_{log2}, … DB_{logn} qui correspondent aux différents points d'entrée sur le serveur. Soit *minSupp* une valeur de support minimal, la problématique de l'analyse plus fine consiste à déterminer les comportements contigus de DB_{log} ainsi que dans chacune des partitions DB_{log1}, DB_{log2}, ... DB_{logn} qui possèdent une fréquence supérieure à *minSupp*. L'intérêt de rechercher les comportements contigus sur les partitions réside dans le fait que nous recherchons des comportements qui sont peu fréquents sur l'ensemble de la base mais qui deviennent fréquents par rapport aux points d'entrée.

Exemple 53 :

Considérons les parcours suivants sur un site : $S_1 = <(A_1)\ (B_2)\ (X_3)\ (C_4)>$, $S_2 = <(A_1)\ (B_2)\ (A_3)\ (Y_4)\ (C_5)>$ et $S_3 = <(A_1)\ (B_2)\ (Z_3)\ (C_4)>$. La première problématique indiquera avec un support minimal de 2 séquences, que le parcours $< (A_1)\ (B_2)\ (C_4)>$ est fréquent. Par contre, dans le cas de la seconde problématique, seule la séquence $< (A_1)\ (B_2)>$ est fréquente dans la mesure où nous sommes intéressés par les éléments qui se suivent réellement, i.e. les éléments contigus.

2.1.2 Analyse de tendances

Les fichiers access log ne sont pas statiques et de nombreux utilisateurs parcourent les sites Web générant ainsi de nouvelles entrées dans ces fichiers. Même si le Web Usage Mining a rendu possible l'étude des comportements des utilisateurs, l'étude des tendances de ces comportements apporte des informations utiles pour un gestionnaire de site. De telles études peuvent être utilisées pour améliorer la qualité du service, mesurer l'impact des modifications sur un site ou encore dans le cas du commerce électronique mesurer l'impact d'une opération marketing.

Le problème de l'analyse des tendances est complémentaire de celui de la maintenance des connaissances extraites. Dans le cas de l'analyse de tendances, nous devons maintenir, lors de chaque extraction de connaissances, la liste de tous les sous arbres fréquents pour voir comment ceux-ci évoluent au cours du temps. La problématique de l'analyse des tendances consiste donc à analyser au cours du temps les comportements des résultats (croissant, décroissant, cyclique...).

Exemple 54 :

Considérons le sous arbre Personne : {identite : {adresse : ⊥, nom : ⊥ }} fréquent pour une valeur de support et à un instant donné. L'analyse de tendances permet de savoir comment cette structure évolue au cours du temps, i.e. par exemple si ce sous arbre a tendance à apparaître de plus en plus dans la base de données ou si, au contraire, à l'issue des mises à jour des données sources, ce sous arbre a plutôt tendance à disparaître.

2.2 Aperçu des travaux antérieurs

2.2.1 Travaux autour de l'analyse de comportement

Un panorama complet des systèmes et applications existants dans le domaine du Web Usage Mining est proposé dans [Cool00, KoBl00, SrCo00]. Dans cette partie nous présentons les principaux travaux existants.

L'un des précurseurs dans le domaine de l'analyse des comportements des utilisateurs est le projet WebMiner [MoJa96, CoMo97] qui illustre la nécessité de mettre en œuvre une architecture particulière pour le Web Usage Mining. WebMiner divise le processus représenté Figure 110 en deux grandes parties : d'une part les processus dépendants du domaine qui transforment les données Web en informations exploitables (prétraitement des fichiers de log, identification des transactions et intégration), d'autre part des processus plus génériques de fouilles de données et recherche de motifs (règles d'associations et motifs séquentiels).

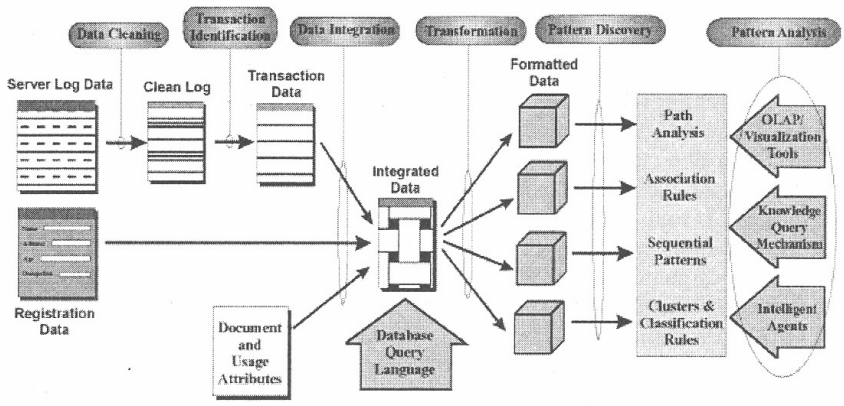


Figure 110 - Architecture générale du Web Usage Mining

Cette architecture générale est globalement la même pour tous les systèmes existants. Aussi, dans la suite de cette section, nous présentons les différences existantes du point de vue des prétraitements et des représentations des données ainsi que sur les différents types d'algorithmes d'extractions utilisés.

Prétraitement et représentation des données
Généralement, un nombre varié de fichiers (images, son, vidéos, cgi) est associé à l'affichage d'une page Web demandée par un client. Ainsi, dans un fichier log peuvent se retrouver des informations jugées inutiles pour le décideur. Une approche classiquement utilisée dans des systèmes comme WebMiner ou WebTool [MaPo99a, MaPo99c] consiste à supprimer ces données lors de l'étape de prétraitement.

Dans WebLogMiner [HaKa01, ZaXi98], les auteurs proposent, par contre, de conserver ces données afin de créer un cube de données (Web Data Cube). Dans ce cube, représenté Figure 111, chaque dimension correspond à un champ possible comme par exemple :

- URL : domaine, dossier, fichier, extension.
- Ressource : html, cgi, image, son, media.
- Temps : heure, minute, seconde, jour, semaine, mois, trimestre, année.
- Référant : domaine à l'origine de la requête.
- Statut : hiérarchie de codes d'erreurs.

Pour faciliter une première analyse, des opérations comme l'agrégation (*roll-up*), le partitionnement (*drill-down*), le découpage en tranche (*slice*) ou en plusieurs dimensions (*dice*) peuvent être appliquées.

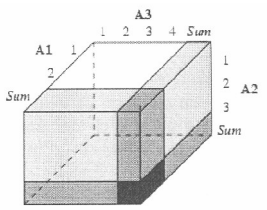


Figure 111 - Un cube de données

Même si classiquement, les données extraites sont stockées dans des bases de données, d'autres manières pour représenter les données existent. La première est basée sur deux applications XML [BrPa98], XGMML et LOGML [PuKr01]. XGMML a pour objet la description d'un graphe quelconque. Il permet d'ajouter à la description purement structurelle de celui-ci des informations d'ordre sémantique si nécessaire. Dans son architecture le document XGMML correspondant à un site donné est généré par programmation du robot W3C [Webb00] qui parcourt le site à la manière d'un aspirateur de site. L'objectif de LOGML est de réaliser une description compressée des fichiers de log. A l'aide de la description XGMML correspondante, les informations relatives aux parcours des utilisateurs sont ainsi regroupées dans une base de données de fichiers LOGML.

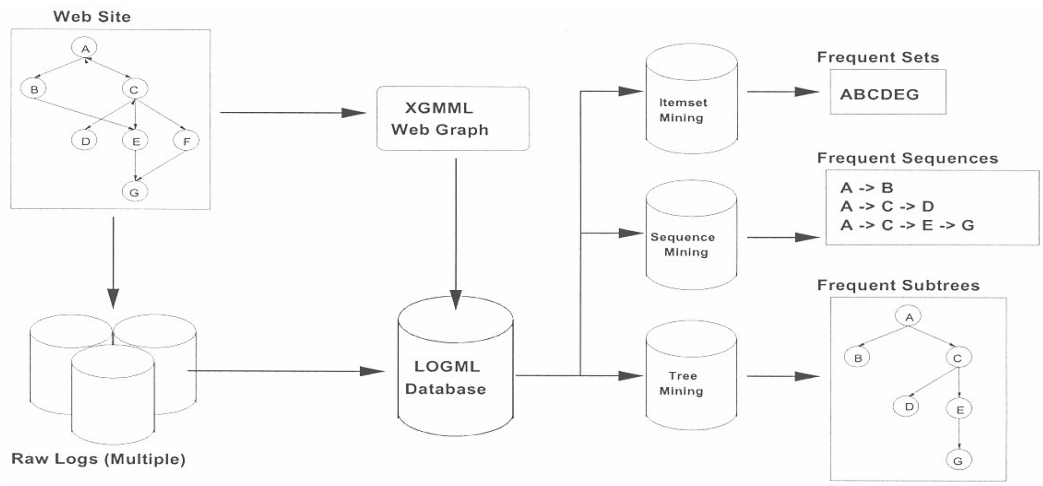


Figure 112 – Représentation des données en XGMML et LOGML

La seconde approche part du constat que les outils de fouilles de données générant des motifs séquentiels ne sont pas adéquats car ils ne permettent pas d'identifier des motifs de navigation qui satisfassent aux propriétés fournies par l'expert.

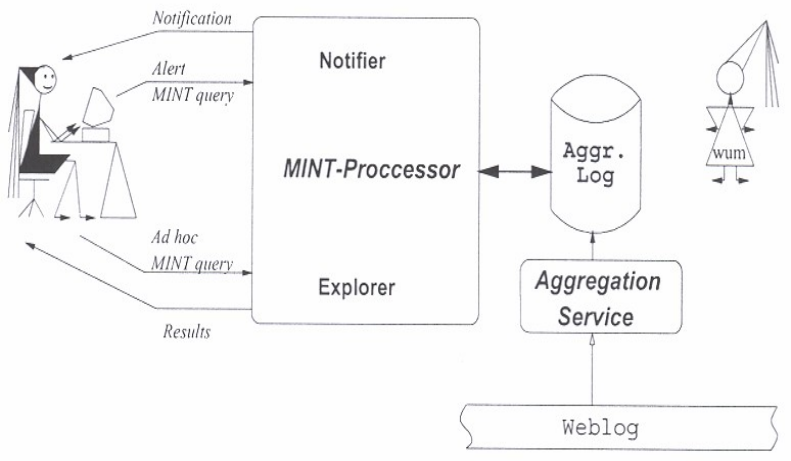


Figure 113 - L'architecture de WUM

Pour résoudre ce problème les auteurs de WUM : Web Utilisation Miner [SpFa98], proposent l'intégration d'un outil de requête, appelé MINT, directement à l'outil de fouille comme on peut le voir dans la Figure 113. MINT sert d'interface à l'expert du système. Deux principaux modules composent WUM : le service d'agrégation qui prépare les données (transactions des utilisateurs), et MINT pour la fouille (à intervalles périodiques pour générer éventuellement des alertes, ou à la demande). Le log agrégé est composé des parcours de l'utilisateur qui sont une séquence des pages demandées par le visiteur durant une transaction. Ces parcours nourrissent des arbres d'agrégation (trie) qui permettent d'extraire des motifs de navigation en utilisant un schéma UML de log agrégé. Les requêtes MINT sont évaluées selon ce schéma mais le processus peut être long et coûteux. En résumé, WUM se veut un outil proche de l'expert, nécessitant une implication directement en amont du processus de collecte d'information et donc de fouille.

Extraction de connaissances

Basés sur les données obtenues lors de l'étape précédente, de nombreux algorithmes ou approches ont été définis pour extraire de la connaissance des fichiers logs. Alors que les auteurs de [ZaXi98] et de [Dyre97] proposent d'utiliser des algorithmes classiques de fouilles de données (recherche de règles d'associations, prédiction, classifications), de nombreuses approches ont été proposées ces dernières années pour rechercher efficacement des comportements fréquents. Ainsi, les auteurs de WebLogMiner utilisent un algorithme de recherche de règles d'associations, similaire à celui de [AgSr94], et adapté aux motifs séquentiels. Un langage d'interrogation, basé sur l'utilisation de requête SQL est proposé et offre à l'utilisateur un meilleur contrôle sur le processus d'extraction. Il permet l'utilisation ou non de certaines règles d'extraction ou la sélection de motifs particuliers. Dans [MaTo97], un algorithme efficace pour la recherche de séquences d'événements, Minepi, est utilisé pour extraire directement des règles à partir de fichiers access log de l'université d'Helsinki. L'originalité de l'approche consiste à considérer chaque page consultée comme un événement et une fenêtre de temps similaire au paramètre Δt de [CoMo97] permet de regrouper les entrées suffisamment proches. Une approche assez comparable est également proposée dans [MaPo99a]. Enfin, plus récemment, [PeHa00] proposent : WAP-Tree une structure compressée et concise pour gérer de manière élégante ces séquences et WAP-Mine un algorithme récursif de fouilles de données basé sur les propriétés de cette structure. Cet algorithme est une adaptation de l'algorithme PrefixSpan que nous avons vu dans le Chapitre II.

2.2.2 Travaux autour de l'analyse de tendances

Il n'existe, à notre connaissance, que peu de travaux analysant les différentes tendances d'évolution des structures fréquentes au cours du temps. Cependant, de nombreux travaux existent pour analyser des tendances, notamment dans le cas de séries temporelles [HaKa01] (mouvements à long terme ou court terme, mouvements cycliques, mouvements aléatoires, …). Dans cette section, nous présentons les travaux qui sont les plus proches de notre problématique, i.e. la recherche de tendances dans des bases de données textuelles et dans le cadre du Web Content Mining.

Dans [LeAg97], les auteurs s'intéressent à la découverte de tendances au sein de bases de données textuelles concernant des dépôts de brevets. Le principe utilisé dans le système Patent Miner est représenté dans la Figure 114.

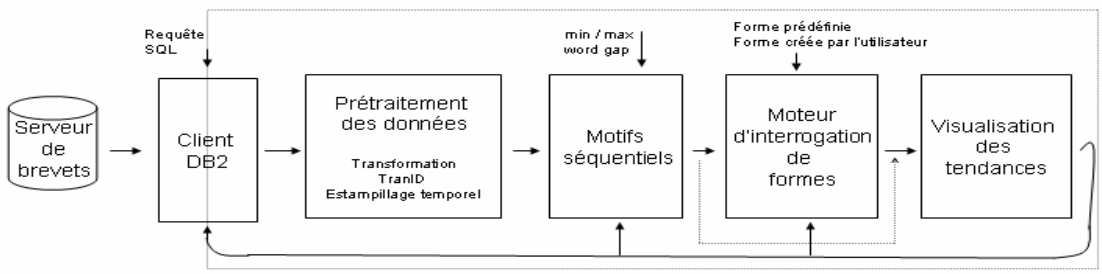


Figure 114 - Le système Patent Miner

Tout comme un processus classique d'extraction, les documents issus du serveur de brevets sont d'abord prétraités et stockés dans une base de données. Durant la phase de prétraitement les documents sont transformés pour être manipulables par un algorithme de recherche de motifs séquentiels, i.e. à chaque document est affecté une estampille temporelle et à chaque mot une ID. Chaque mot est considéré comme un item et le problème consiste à identifier la fréquence de chaque phrase (liste de mots) en utilisant les techniques de recherche de motifs séquentiels [AgSr95, SrAg96]. A l'aide des estampilles assignées aux documents du serveur, le corpus de brevet peut être partagé à l'aide d'une granularité (jour, mois, an...) spécifiée par l'utilisateur. L'utilisation de l'algorithme de fouille de données, sur chaque élément de la partition, permet de générer un ensemble de triplets (phrase, date, support) qui constitue l'historique des phrases fréquentes. A partir de cet historique, une requête sur les formes, i.e. les tendances, est effectuée, soit de manière interne par le système soit sur la demande d'un utilisateur. La requête de forme est réalisée à l'aide du langage de définition SDL [AgPs95].

Une approche d'extraction de nouvelles sur Internet est proposée dans [MeMo02]. Comme l'illustre la Figure 115, cette méthode reprend les principes d'un processus d'extraction de connaissances classique. La partie récupération des sources se charge dans un premier temps de récupérer sur Internet les nouvelles issues des différents sites concernés. Une première analyse de ces nouvelles permet d'éliminer les informations inutiles et d'identifier de nouvelles urls à explorer. Le processus de récupération des sources est activé de manière régulière (délai défini). La phase de prétraitement transforme l'information ainsi récupérée dans une représentation structurée. Cette représentation est composée de la source de l'information, de la date et d'une représentation formelle de son contenu. L'étape suivante, la généralisation, constitue le cœur de ce système. Les auteurs à l'aide d'une approche probabiliste définie dans [MoGe01], construisent une distribution probabiliste des sujets. Ensuite l'analyse de tendances est assimilable à une mesure probabiliste : *l'entropie relative*. Dans la partie analyse, l'utilisateur peut interagir avec le système et peut ainsi choisir, par exemple, la plage temporelle qui l'intéresse, i.e. la détection de tendances locales.

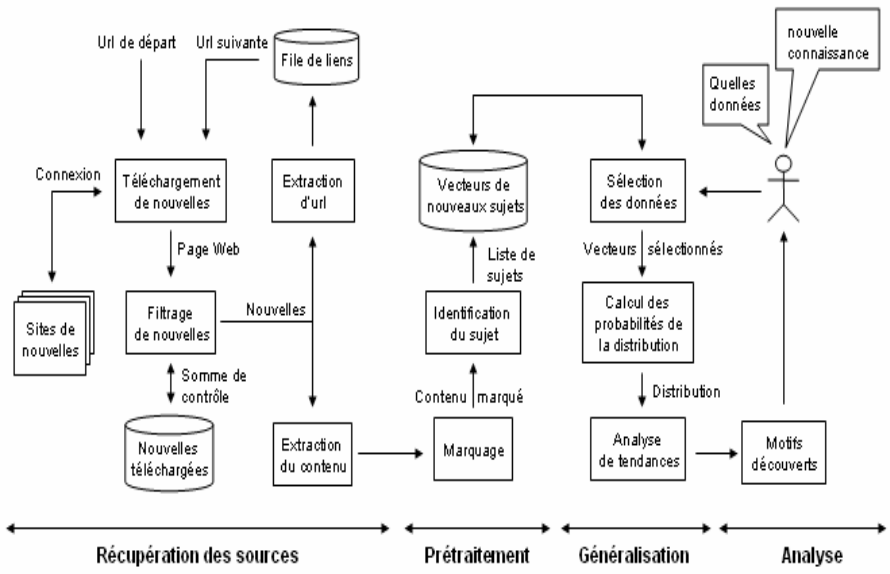


Figure 115 - Le processus de recherche de tendances dans des sites Web dynamiques

2.3 AUSMS-Web

Nous montrons, dans cette section, comment le système AUSMS a été adapté pour répondre aux différentes problématiques du Web Usage Mining et présentons les résultats d'expérimentations réalisées à l'aide de ce système sur des jeux de données réels.

2.3.1 Architecture fonctionnelle

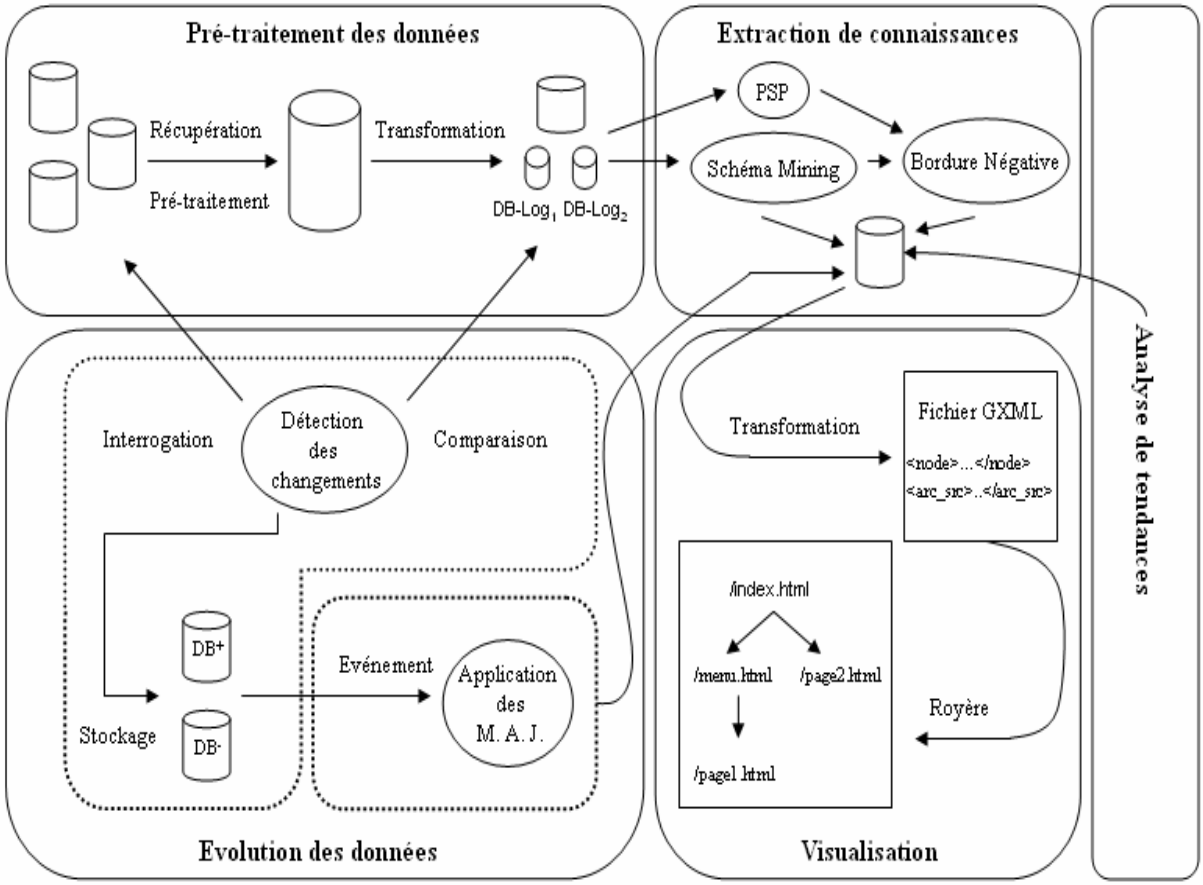


Figure 116 - AUSMS-Web

L'objectif de AUSMS-Web est de proposer un environnement de découverte et d'extraction de connaissances depuis un ou plusieurs serveurs Web. Ce système propose des solutions pour : la récupération de l'information, l'extraction, la mise à jour de connaissances extraites et enfin l'analyse des tendances [LaTe03b].
AUSMS-Web est une extension d'AUSMS pour pouvoir gérer les données issues des serveurs Web et réaliser l'analyse de tendances. Les principes généraux qui le régissent sont similaires à ceux de AUSMS. De manière à répondre aux deux problématiques présentées dans la section 2.1.1, deux types d'algorithmes d'extraction sont intégrés dans le système. Le premier algorithme d'extraction de motifs séquentiels est basé sur l'approche PSP décrite dans le Chapitre II. Dans cette section, nous ne détaillons pas les aspects liés à l'extraction de motifs séquentiels[2] et nous nous focalisons sur l'analyse de comportements contigus.

Comme nous l'avons annoncé précédemment, il existe une bijection entre le comportement d'un utilisateur sur un site Web, i.e. sa navigation sur le site, et un parcours de graphe. En effet, dans ce graphe, les arcs expriment la navigation à travers les différentes pages et un nœud correspond à un lien hypertexte.

[2] Le lecteur intéressé peut se reporter à [Mass02] où une description complète du système WebTool spécialement dédié à l'utilisation de motifs séquentiels dans le cadre du Web Usage Mining est proposée.

Exemple 55 :

Considérons le parcours d'un utilisateur sur un site décrit Figure 117. A partir d'un point d'entrée (/index.html), celui-ci s'est rendu sur la page (/index.html/page1.html), puis sur (/index.html/page2.html), est retourné sur (/index.html/page1.html) pour aller sur (/index.html/page3.html). Enfin après être repassé sur (/index.html/page1.html) et sur (/index.html), il s'est rendu sur (/index.html/page4.html) et (/index.html/page5.html).

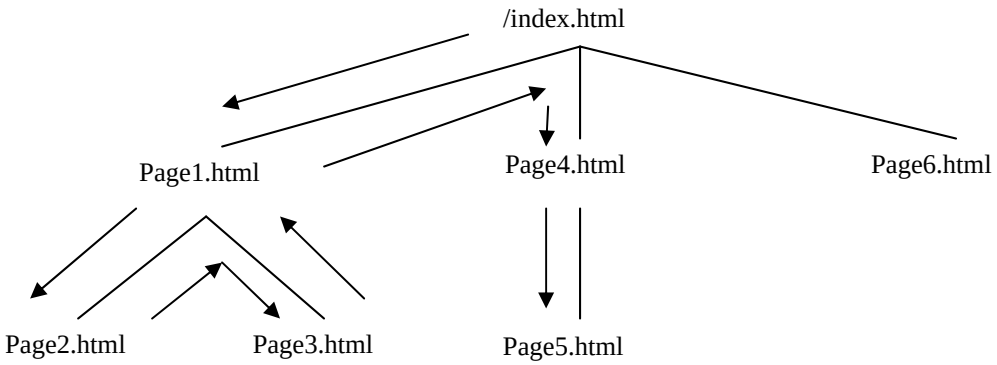


Figure 117 – Un exemple de parcours sur un site

Par rapport à la problématique d'extraction de connaissances que nous avons défini dans le Chapitre II, les navigations des utilisateurs peuvent donner lieu à des cycles comme l'illustre l'exemple précédent. Nous utilisons, dans AUSMS-Web, la méthode proposée dans [WaLi99] basée sur la réplication des sous fils partagés pour transformer ces graphes cycliques en graphes acycliques. Ainsi, à l'issue de la phase de prétraitement, nous disposons d'un ensemble de couples (*id*, *navigation*) où *id* représente l'identifiant unique d'un utilisateur (son numéro IP) et *navigation* représente la séquence de navigation transformée par l'algorithme *TreeToSequence*. Durant la phase de prétraitement, nous pouvons également intégrer la notion de session de travail, i.e. le temps pendant lequel un utilisateur est actif, à l'aide d'un paramètre temporel (dans le cadre de certaines expérimentations, nous avons spécifié un délai de 30 minutes à partir duquel nous considérons qu'il s'agit d'une nouvelle session de travail). Ce paramètre est bien entendu modifiable par l'utilisateur.

Exemple 56 :

A l'issue des prétraitements et des transformations par TreeToSequence, la navigation de l'exemple Exemple 55 devient : < (⊕/index.html$_1$) (⊕/index.html/page1.html$_2$) (⊕/index.html/page2.html$_3$) (⊕/index.html/page1.html$_4$) (⊕/index.html/page3.html$_5$) (⊕/index. html/page1.html$_6$) (⊕/index.html$_7$)(⊕/index.html/page4.html$_8$) (⊥/index.html/page5.html$_9$).

Dans un premier temps, les différents couples (*id*, *navigation*) sont stockés dans DB_{log} .Cette dernière est ensuite partitionnée en fonction des points d'entrée sur le site. DB_{log} est utilisée soit par l'algorithme d'extraction de motifs séquentiels (PSP) soit par l'algorithme PSP_{tree} avec la valeur de support minimal *minSupp* afin de rechercher les motifs fréquents et contigus fréquents sur toute la base. Les partitions DB_{log1}, DB_{log2}, … DB_{logn} sont utilisées par l'algorithme PSP_{tree} afin d'extraire les sous arbres fréquents dans chacune des partitions. Ces derniers correspondent bien entendu à des comportements contigus dans notre cas car nous n'autorisons pas de sous arbres qui ne respectent pas la topologie.

Comme nous l'avons vu dans la description d'AUSMS, il est possible à partir de ces extractions, de maintenir des connaissances dans le cas d'ajout de données dans les fichiers logs ou de modifications des valeurs de support.

2.3.2 Le module d'étude des tendances

Comme nous l'avons précisé dans la section 2.1.2, l'analyse des tendances consiste à rechercher quelles sont les évolutions des sous arbres fréquents au cours du temps. L'un des problèmes associé à cette analyse est le

stockage des données : comment trouver une structure efficace pour maintenir les connaissances extraites au fur et à mesure ? Etant donné le nombre de résultats intermédiaires, il est indispensable de trouver une structure de représentation adaptée. Le second problème lié aux tendances est de rechercher rapidement les mêmes structures afin de suivre leurs évolutions au cours du temps et en fonction des desiderata de l'utilisateur (tendance croissante, décroissante, cyclique, …).

Algorithm TrendAnalysis

Input : $(L^{DB}_{t1} + L^{DB}_{t2} + \ldots + L^{DB}_{tn}) = L^{DB}_{t}$, l'ensemble des structures fréquentes stockées à des dates t_1 ..... t_n avec leur support à ces dates.

Output : H_f, l'historique de chaque fréquent f sous la forme de couple (date, support).

```
1 : Foreach different frequent f in L^DB_t do
2 :         H_f = ∅;
3 :         Foreach frequent e ∈ L^DB_t do
4 :             If f = e then H_f = H_f + (e.date, e.support); endif
5 :         enddo
6 :         Return H_f;
7 : enddo
```

Algorithme 21 – L'algorithme TrendAnalysis

Le principe utilisé est actuellement le suivant. L'algorithme *FindSubStructure* décrit dans le Chapitre III est exécuté pour différentes valeurs de support. L'utilisateur peut spécifier la valeur du pas utilisé. Par exemple, l'utilisateur peut choisir de faire varier les valeurs de support d'un pas de 5% et dans ce cas l'analyse commence par 0.05, 0.10, 0.15, … Les résultats obtenus, i.e. les sous arbres fréquents, sont alors stockés. L'algorithme *TrendAnalysis* montre comment les historiques sont calculés à partir des sous arbres fréquents stockés à des dates différentes. Pour chaque sous arbre dont on désire connaître l'historique, nous cherchons aux différentes date $t_1, t_2, ..., t_n$ sa valeur de support. L'historique correspond alors à un ensemble de couples (date, support). A l'aide de cet historique, nous pouvons sélectionner ceux dont l'évolution du support correspond à un certain type de tendance (croissance, décroissance, …) au cours du temps. Les tendances décrivent simplement les évolutions qui respectent le choix de l'utilisateur avec les informations complémentaires sur les périodes de temps pendant lesquelles la tendance est vérifiée.

2.3.3 Expérimentations

De manière à valider notre approche, nous avons utilisé le système AUSMS-Web sur différents jeux de données. Dans la suite, nous présentons quelques expériences significatives sur différents jeux de données : la première expérience a été menée sur le site statique du Laboratoire LIRMM (Laboratoire d'Informatique, de Micro Electronique, de Robotique de Montpellier). La seconde expérience correspond à l'analyse du site statique des anciens élèves d'une école de chimie. Enfin, la dernière expérience concerne un site dynamique et multiserveur de e-commerce d'un opérateur de téléphonie. Pour chacun de ces jeux de donnés, nous présentons des résultats d'utilisation d'AUSMS-Web dans le cas de l'extraction de comportements contigus et de l'analyse de tendances. Les expériences menées sur la mise à jour des connaissances ont été décrites dans le chapitre précédent.

Jeu de données du LIRMM

Le premier jeu de données est issu du LIRMM et regroupe les différentes connexions réalisées sur le site Web du laboratoire du 1er septembre 1996 au 10 janvier 2000. Dans le cadre de cette expérience, nous avons souhaité travailler sur une longue période de manière à voir apparaître de nouveaux usages (nouveaux arrivants, nouvelles conférences, nouveaux cours, …). Le fichier d'access log initial a une taille de 200 MO et contient 968 identifiants de visiteurs différents. Le nombre de pages différentes étudiées est de 1196. Au cours des différentes navigations, le nombre moyen de pages visitées est de 22. L'intégration de ce jeu de donnés au sein de AUSMS-Web a nécessité l'écriture d'analyseurs lexicaux spécifiques afin d'effectuer un prétraitement des données. Dans l'expérience que nous reportons, nous avons conservé, lors du prétraitement, les différentes images envoyées au navigateur.

Après avoir effectué les différents traitements, les données ont été stockées dans DB_{log} qui a lui-même été partitionné en fonction des différents points d'entrée. Nous avons pu constater que dans le fichier log de très

nombreuses personnes passaient directement par les pages personnelles sans passer directement par la racine. Par souci de confidentialité, nous notons par la suite $PE_{i,}$ les différents points d'entrée.

La première expérience a été de montrer que les motifs séquentiels ne fournissent qu'une information partielle par rapport aux comportements contigus. Pour cela, nous avons appliqué dans un premier temps l'algorithme de recherche de motifs séquentiels sur le jeu de données. La Figure 118 illustre une partie des résultats. Dans un second temps, nous avons appliqué l'algorithme PSP_{tree} sur DB_{log} avec les mêmes variations de support. La Figure 119 illustre une partie des résultats. Nous constatons ainsi que pour 2%, nous obtenons les comportements contigus suivants : < ([⊕, PE_5, 1]) ([⊕, /, 2]) ([⊥, /robots.txt, 3]) ([⊕, /, 2]) ([⊥, /robots.txt, 3]) > et < ([⊕, PE_6, 1]) ([⊗, /images/html.gif, 2]) ([⊥, /images/memoire.gif, 3]) ([⊥, /images/molecule.gif, 3]) ([⊥, /images/mail.gif, 3]) ([⊕, /images2/search.gif, 4]) **>**. Dans le dernier résultat, nous pouvons constater qu'étant donné que les images sont associées à une page, leur profondeur est similaire et que le comportement de 2% des utilisateurs a été de rentrer sur le site par la page PE6 et de se rendre ensuite sur la page image dans laquelle les images html.gif, memoire.gif, molecule.gif et mail.gif étaient présentes et qu'ensuite ils sont allés sur la page images2 dans laquelle l'image search.gif etait présente.

Support	**Motifs séquentiels**
25%	* < (PE_1) (/moteurs.html) >
20%	* < (PE_1) (/moteurs.html) > * < (PE_2) (/web/web.css) >
15%	* < (PE_1) (/moteurs.html) > * < (PE_2) (/web/web.css) >
10%	* < (PE_1) (/moteurs.html) > * < (PE_2) (/web/web.css) > * < (PE_2) (/web/web.html) > * < (/web/node14.html) > * < (/test_form.html) > * < (/cgi-bin/search.cgi) > * < (/web/image1.html) > .....
5%	* < (PE_3) (/images/calvinGrey.gif) (/images/mail.gif) > * < (PE_4) (/robots.txt) ([/robots.txt) > * < (PE_1) (/moteurs.html) (/cgi-bin/search.cgi) **>** * < (/cgi-bin/search.cgi) (/images/protein-cro.gif) **>** * < (/images/memoire.gif) (/images/protein-cro.gif) **>** * < (/web/node44.html) **>** * < (/web/web.css) (/web/image1.html) > * < (/web/web.css) (/web/node14.html) > ……
2%	* < (PE_5) (/) (/robots.txt) (/) (/robots.txt) > * < (PE_6) (/images/html.gif) (/images/memoire.gif) (/images/molecule.gif) (/images/mail.gif) (/images/search.gif) **>** * < (/cgi-bin/search.cgi) (/images/protein-cro.gif) (/images/emploi.gif) **>** * < (/images/memoire.gif) (/images/protein-cro.gif) (/images/mail.gif) **>** * < (/web/web.css) (/web/node44.html) **>** * < (/web/web.css) (/web/image1.html) > * < (/web/web.css) (/web/node14.html) > * < (/) (/BioBdSlides/sld044.htm) (/BioBdSlides/sld039.htm) > …..

Figure 118 – Exemple de motifs séquentiels sur le jeu de données du LIRMM

Comme nous nous y attendions, un moins grand nombre de résultats apparaissent dans le cas des sous arbres extraits et ceux-ci sont, bien entendu, inclus dans les séquences fréquentes obtenues par PSP. Par contre, les résultats obtenus montrent bien que pour avoir les mêmes résultats entre PSP et PSP_{tree}, il serait indispensable de proposer un post traitement. Outre, le fait que ce dernier nécessite une importante quantité de calculs (il faut parcourir chaque séquence et vérifier qu'elle correspond bien à un arbre par rapport au site), il ne faut pas oublier

que pour obtenir les motifs, un plus grand nombre de candidats a été généré (C.f. remarque sur l'utilisation de motifs séquentiels dans les chapitres précédents).
Enfin, nous avons appliqué PSP_{tree} sur les différentes partitions par rapport aux points d'entrée. Nous voulions vérifier dans ce cas qu'un plus grand nombre de comportements contigus apparaissaient et validaient ainsi le choix de s'intéresser aux points d'entrée pour un gestionnaire de site. La Figure 120 présente les caractéristiques des différents points d'entrée. La Figure 121 illustre quelques uns des résultats obtenus lors de l'extraction de comportements contigus.

Support	**Sous arbres extraits**
25%	* < ([⊕, PE_1, 1]) ([⊕, /moteurs.html, 2]) >
20%	* < ([⊕, PE_2, 1]) ([⊗, /web/web.css, 2]) > * < ([⊕, PE_1, 1]) ([⊕, /moteurs.html, 2]) >
15%	* <([⊕, PE_2, 1]) ([⊗, /web/web.css, 2]) > * <([⊕, PE_1, 1]) ([⊕, /moteurs.html, 2]) >
10%	* < ([⊕, PE_1, 1]) ([⊕, /moteurs.html, 2]) > * < ([⊕, PE_2, 1]) ([⊗, /web/web.css, 2]) > * < ([⊕, PE_2, 1]) ([⊗, /web/web.html, 2]) > .....
5%	* < ([⊗, PE_3, 1]) ([⊥, /images/calvinGrey.gif, 2]) ([⊥, /images/mail.gif, 2]) > * < ([⊕, PE_4, 1]) ([⊥, /robots.txt, 2]) ([⊥, /robots.txt, 2]) > * < ([⊗, PE_1, 1]) ([⊕, /moteurs.html, 2]) ([⊥, /cgi-bin/search.cgi, 3]) > ……
2%	* < ([⊕, PE_5, 1]) ([⊕, /, 2]) ([⊥, /robots.txt, 3]) ([⊕, /, 2]) ([⊥, /robots.txt, 3]) > * < ([⊕, PE_6, 1]) ([⊗, /images/html.gif, 2]) ([⊥, /images/memoire.gif, 3]) ([⊥, /images/molecule.gif, 3]) ([⊥, /images/mail.gif, 3]) ([⊕, /images2/search.gif, 4]) > .....

Figure 119 – Partie des sous arbres extraits sur les données du LIRMM

Point d'entrée	**Taille en % du fichier original**	**Nombre en % de tid par rapport au fichier original**
PE_1	50	28
PE_2	39	24
PE_3	48	7
PE_4	25	7
PE_5	66	11
PE_6	47	6

Figure 120 – Caractéristiques des points d'entrée

Comme nous nous y attendions, l'analyse des comportements contigus uniquement par rapport aux points d'entrée offre une information plus complète que sur l'ensemble de la base. En effet, les jeux de données étant plus réduits, il est évident que la recherche de comportements fréquents fait émerger un plus grand nombre de résultats. Le fait d'utiliser PSP_{tree} sur ces jeux de données offre en outre une analyse plus fine du comportement des utilisateurs uniquement dans la mesure où, pour chaque point d'entrée, l'utilisateur final dispose d'une information complète sur les parcours contigus des utilisateurs. Par exemple, nous voyons que pour l'entrée PE6,

nous obtenons le parcours contigus suivant : < ([⊕, PE_6, 1]) ([⊗, /images/html.gif, 2]) ([⊥, /images/memoire.gif, 3]) ([⊥, /images/molecule.gif, 3]) ([⊥, /images/mail.gif, 3]) ([⊕, /images2/search.gif, 4]) ([⊥, /images2/molecule.gif, 3]) ([⊥, /images2/lirmm.gif, 3])> qui n'apparaissait pas dans le cas de l'analyse de toute la base.

Point d'entrée	**Support**	**Comportement contigu**
PE_1	20%	* < ([⊕, PE_1, 1]) ([⊕, /moteurs.html, 2]) ([⊥, /cgi-bin/search.cgi, 3]) > * < ([⊕, PE_1, 1]) ([⊕, /moteurs.html, 2]) ([⊕, /moteurs.html, 2]) >
PE_2	10%	* < ([⊕, PE_2, 1]) ([⊕, /web/node44.html, 2]) ([⊕, /web/image1.gif, 3]) > * < ([⊕, PE_2, 1]) ([⊕, /web/node7.html, 2]) ([⊗, /web/web.css, 2]) ([⊕, /web/node14.html, 2]) > …
PE_3	30%	* < ([⊗, PE_3, 1]) ([⊥, /images/calvinGrey.gif, 2]) ([⊥, /images/mail.gif, 2]) ([⊥, /images/protein-cro.gif, 2]) ([⊥, /images/star.gif, 2]) > * < ([⊗, PE_3, 1]) ([⊥, /images/calvinGrey.gif, 2]) ([⊥, /images/interet.gif, 2]) ([⊥, /images/protein-cro.gif, 2]) ([⊥, /cgi-bin/eureka ?MtAdmin=MtInstances&MTclassID=407, 3]) > …
PE_4	20%	* < ([⊕, PE_4, 1]) ([⊥, /robots.txt, 2]) ([⊥, /robots.txt, 2]) ([⊥, /robots.txt, 2]) > * < ([⊕, PE_4, 1]) ([⊥, /robots.txt, 2]) ([⊗, /web/web.html, 2]) ([⊕, /web/node41.html, 3]) > …
PE_5	10%	* < ([⊕, PE_5, 1]) ([⊕, /, 2]) ([⊥, /robots.txt, 3]) ([⊕, /, 2]) ([⊥, /robots.txt, 3]) ([⊕, /, 2]) ([⊥, /robots.txt, 3]) ([⊕, /, 2]) > * < ([⊕, PE_5, 1]) ([⊕, /, 2]) ([⊕, /exemple.html, 3]) ([⊗, /web/web.html, 2]) ([⊥, /Javascriptessai.html, 3]) ([⊕, /moteurs.html, 2]) > …
PE_6	25%	* < ([⊕, PE_6, 1]) ([⊗, /images/html.gif, 2]) ([⊥, /images/memoire.gif, 3]) ([⊥, /images/molecule.gif, 3]) ([⊥, /images/mail.gif, 3]) ([⊕, /images2/search.gif, 4]) ([⊥, /images2/molecule.gif, 3]) ([⊥, /images2/lirmm.gif, 3])> * < ([⊕, PE_6, 1]) ([⊗, /images/html.gif, 2]) ([⊥, /images/protein-cro.gif, 3]) ([⊥, /images/lirmm.gif, 3]) > …

Figure 121 – Comportements contigus pour des points d'entrée

Afin de pouvoir valider notre approche sur l'étude des tendances, nous avons divisé le fichier log en dix périodes pour pouvoir observer ces tendances. Ainsi nous obtenons des fichiers log représentant les navigations des utilisateurs sur différentes périodes. Le premier fichier log contient les informations relatives aux parcours entre le 1er septembre 1996 et le 10 janvier 1997, le second entre le 1er septembre 1996 et 20 mai 1997, et ainsi de suite jusqu'au 31 mars 2000.

Le tableau de la Figure 122 représente la taille de dix fichiers log ainsi générés ainsi que le nombre moyen de pages visitées. En première analyse, nous constatons que globalement le trafic sur le site est relativement régulier. Par contre, le nombre moyen de pages visitées décroît fortement lors des premières périodes et se stabilise peu à peu dès la 6ième période entre 20 et 30 pages pour la même visite.

Fichier	Taille	Nombre de pages visitées en moyenne
01/09/96 – 10/01/97	17 Mo	120
01/09/96 – 20/05/97	36 Mo	65
01/09/96 – 30/09/97	56 Mo	45
01/09/96 – 10/01/98	77 Mo	37
01/09/96 – 20/05/98	101 Mo	31
01/09/96 – 30/09/98	127 Mo	28
01/09/96 – 10/01/99	153 Mo	25
01/09/96 – 20/05/99	168 Mo	22
01/09/96 – 30/09/99	187 Mo	21
01/09/96 – 10/01/00	201 Mo	22

Figure 122 – Taille des fichiers log

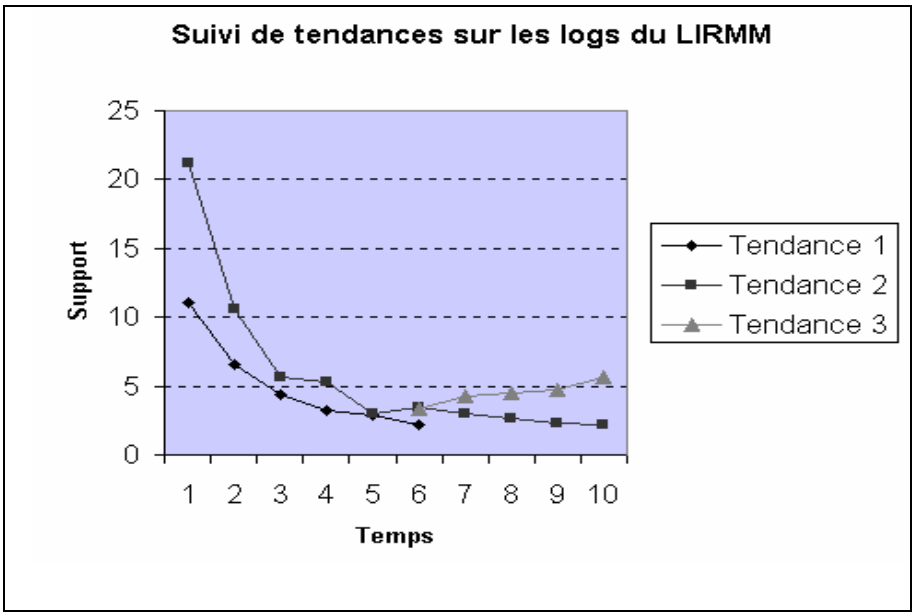


Figure 123 – Exemple de tendances

Nous avons appliqué l'algorithme d'analyse de tendance sur ces jeux de données. La Figure 123 illustre un exemple de tendances extraites. Nous remarquons que la tendance 1 qui correspond à l'arbre suivant : < ([⊗, PE_6, 1]) ([⊗, /images/html.gif, 2]) ([⊥, /images/calvinGrey, 3]) ([⊥, /images/mail.gif, 3]) ([⊥, /images/calvinGrey.gif, 3]) **>** décroît avec le temps. Nous remarquons aussi que cette tendance disparaît complètement durant la 6ième période. En effet, dus à une valeur de support trop faible, nos algorithmes ne sont pas en mesure de fournir un résultat à ce point. Une tendance décroissante est aussi mise en évidence pour la tendance 2 qui représente : < ([⊗, PE_3, 1]) ([⊥, /images/memoire.gif, 2]) ([⊥, /images/molecule.gif, 2]) ([⊥, /images/mail.gif, 2]) ([⊥, /images/emploi.gif, 2]) **>**. Finalement, la tendance 3 : < ([⊕, PE_4, 1]) ([⊥, /robots.txt, 2]) ([⊥, /robots.txt, 2]) > évolue dans le temps et correspond en fait au nombre de fois que ces pages sont atteintes par un moteur de recherche.

Jeu de données de l'Ecole de Chimie

Le deuxième jeu de données est issu du serveur Web de l'association des anciens élèves de l'Ecole Nationale Supérieure de Chimie de Montpellier pour la période allant du 7 juillet 2002 au 15 septembre 2002. Le fichier d'access log utilisé a une taille de 9,53 Mo et contient 3736 identifiants de visiteurs différents. Contrairement au précédent, le fichier sur lequel les expériences ont été menées a été construit par la concaténation des fichiers log hebdomadaires générés par le serveur.

Le nombre de pages différentes étudiées est de 979 pages. Le nombre de pages visitées en moyenne par navigation est de 29. Tout comme le jeu de données précédent, l'intégration de ce jeu de données au sein de AUSMS-Web a nécessité l'écriture d'analyseurs lexicaux spécifiques afin de prétraiter les données. Les différents points d'entrée sont notés PE_i. Dans un premier temps nous avons réalisé différentes extractions de connaissances pour des supports variant entre 4 et 40 %.

Support	**Quelques structures extraites**
40%	* < ([⊕, PE_1, 1]) ([⊥, /images/backgrd3.gif, 2]) > * < ([⊕, PE_1, 1]) ([⊥, /images/meter050.gif, 2]) >
30%	* < ([⊕, PE_1, 1]) ([⊥, /images/backgrd3.gif, 2]) ([⊥, /images/meter050.gif, 2]) > * < ([⊕, PE_2, 1]) ([⊗, /css/00.css, 2]) >
20%	* < ([⊕, PE_1, 1]) ([⊥, /images/backgrd3.gif, 2]) ([⊥, /images/meter050.gif, 2]) > * < ([⊕, PE_2, 1]) ([⊗, /css/00.css, 2]) ([⊥, /images/menu016c.gif, 3]) > * < ([⊕, PE_2, 1]) ([⊗, /css/00.css, 2]) ([⊥, /images/menu005c.gif, 3]) ([⊥, /images/menu017c.gif, 3]) > …..
10%	* < ([⊕, PE_3 1]) ([⊗, /, 2]) ([⊥, /images/menu003.gif, 2]) ([⊥, /images/logotram.jpg, 3]) > * < ([⊕, PE_2, 1]) ([⊗, /css/03.css, 2]) ([⊥, /images/menu002a.gif, 3]) ([⊥, /images/menu003d.gif, 3]) ([⊥, /images/menu005c.gif, 3]) ([⊥, /images/menu017d.gif, 3])> * < ([⊕, PE_4, 1]) ([⊗, /societe/pharma.htm, 2]) > .....
7%	* < ([⊕, PE_3 1]) ([⊗, /, 2]) ([⊥, /images/menu003.gif, 2]) ([⊥, /images/logotram.jpg, 3]) > * < ([⊕, PE_2 1]) ([⊗, /css/03.css, 2]) ([⊥, /images/menu020a.gif, 3]) ([⊥, /images/menu003d.gif, 3]) ([⊥, /images/menu005c.gif, 3]) ([⊥, /images/menu017c.gif, 3]) > * < ([⊕, PE_5, 1]) ([⊕, /forum/forum00.htm, 2]) > ……
4%	* < ([⊕, PE_3 1]) ([⊗, /, 2]) ([⊗, /css/00.css, 2]) ([⊗, /css/01.css, 2]) ([⊥, /images/meter150.gif, 3]) > * < ([⊕, PE_2 1]) ([⊗, /css/00.css, 2]) ([⊥, /images/backgrd3.gif, 3]) ([⊥, /images/menu004a.gif, 3]) ([⊥, /images/meter050.gif, 3]) ([⊥, /images/menu003d.gif, 3]) > * < ([⊕, PE_6, 1]) ([⊕, /societe/societes.htm, 2]) ([⊗, /societe/pharma.htm, 3]) > .....

Figure 124 – Extrait de résultats sur les données d'AAE

Le tableau de la Figure 124 illustre un partie des sous arbres extraits pour différentes valeurs de support.

Le fait d'obtenir de manière hebdomadaire les informations des serveurs a permis de réaliser plusieurs expériences lors de l'analyse des tendances. Nous considérons pour la suite que le fichier AAE (1) correspond au fichier log hebdomadaire, alors que le fichier AAE (2) correspond au cumul des différentes informations, i.e. au cumul de tous les fichiers AAE (1) sur la période. L'avantage de comparer un fichier régulier à un fichier cumulé est que, dans le cas d'un fichier régulier, un comportement nouveau peut devenir fréquent alors qu'il risque d'être « noyé » par les autres comportements dans le cas du fichier cumulé. La Figure 125 indique les différentes tailles de fichiers hebdomadaires ainsi que celle du fichier cumulé.

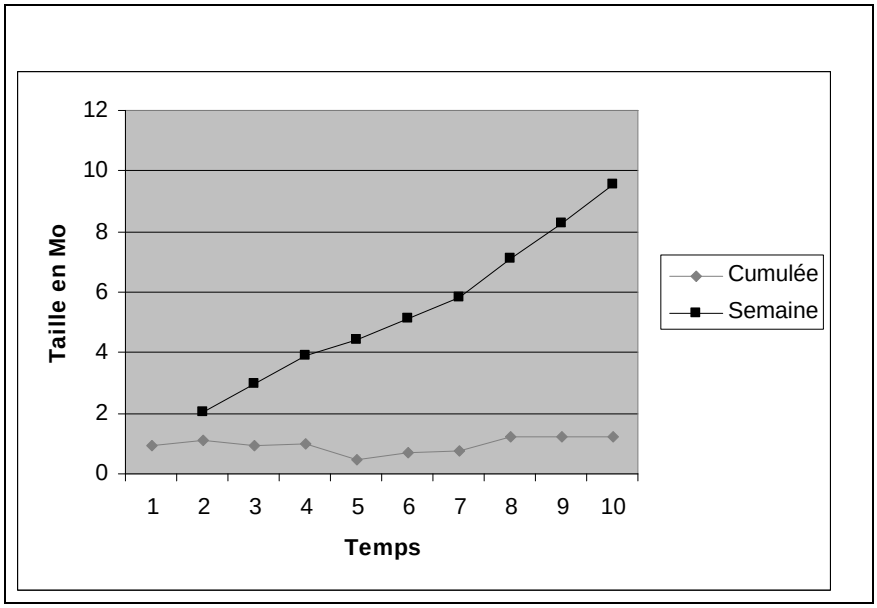


Figure 125 - Taille des jeux de données

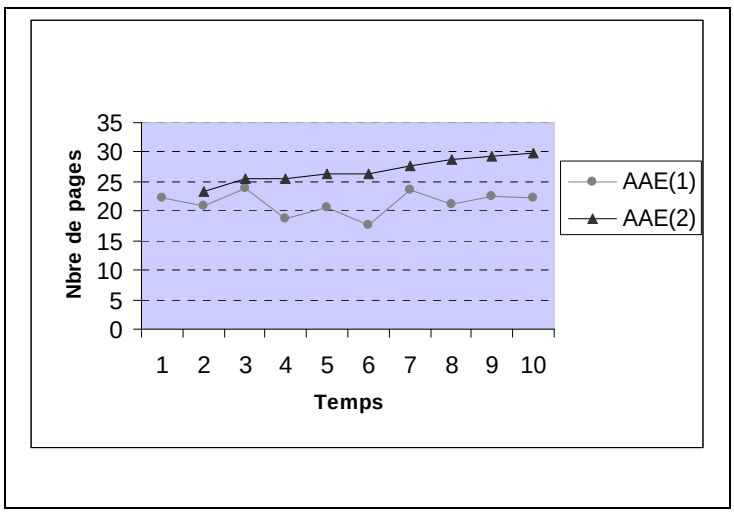


Figure 126 - Nombre de pages visitées en moyenne

Etant donné que nous souhaitions, dans le cadre de l'analyse des tendances, nous intéresser à des parcours généraux croissants ou décroissants, il était nécessaire d'avoir un jeu de données dans lequel le trafic soit suffisamment régulier. En outre, nous étions intéressés par les tendances dans les parcours des utilisateurs et non dans les tendances d'utilisation du serveur Web. La Figure 126 indique le nombre de pages visitées en moyenne et nous pouvons ainsi constater que le trafic sur ce site est relativement régulier.

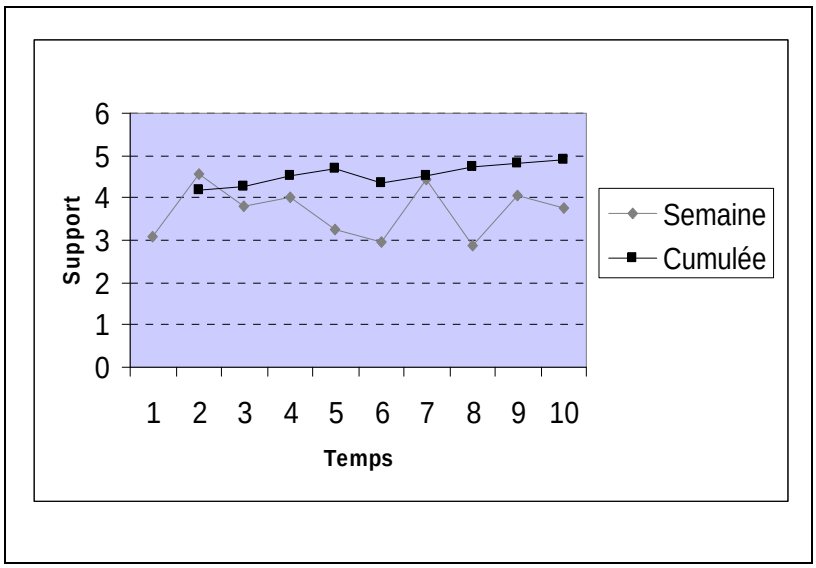


Figure 127 - Un exemple de tendance

De manière à illustrer la différence entre les deux types de fichiers, considérons la Figure 127. Elle illustre le comportement < (/([⊕, *PE*$_4$, 1]) ([⊗, /societe/pharma.htm, 2]) ([⊥, /images/menu02.gif, 3]) ([⊥, /images/menu03x.gif, 3]) ([⊥, /images/menu91.gif, 3]) > qui représente une navigation fréquente des usagers
Dans le cas des données cumulées, nous pouvons constater que ce comportement est globalement croissant. L'ordonnée représentant le support, nous constatons en effet, qu'entre le début et la fin de l'analyse, de plus en plus de comportements d'usagers correspondent à ce motif. Par contre, si l'analyse est effectuée semaine après semaine, nous pouvons considérer que ce comportement n'est pas aussi croissant qu'il le semblait sur le cumul. En effet, nous constatons qu'entre les semaines 2 et 6 il était même plutôt décroissant et qu'un pic d'utilisation de ces urls a eu lieu lors de la semaine 7. La conclusion attendue se vérifie sur cette courbe. L'analyse globale ne permet pas de repérer tous les comportements, certains se retrouvent alors « noyés » dans des comportements plus globaux. Dans ce cas de figure, contrairement aux approches traditionnelles nous pouvons repérer ce type de comportement.

Considérons à présent les figures suivantes qui représentent des analyses de tendances sur une longue période de temps.

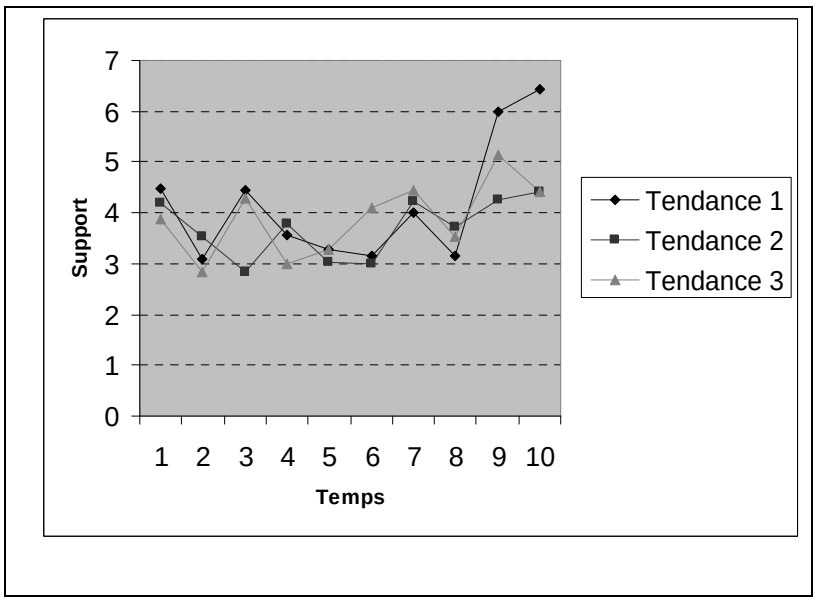


Figure 128 - Suivi de tendances sur les logs de AAE (1)

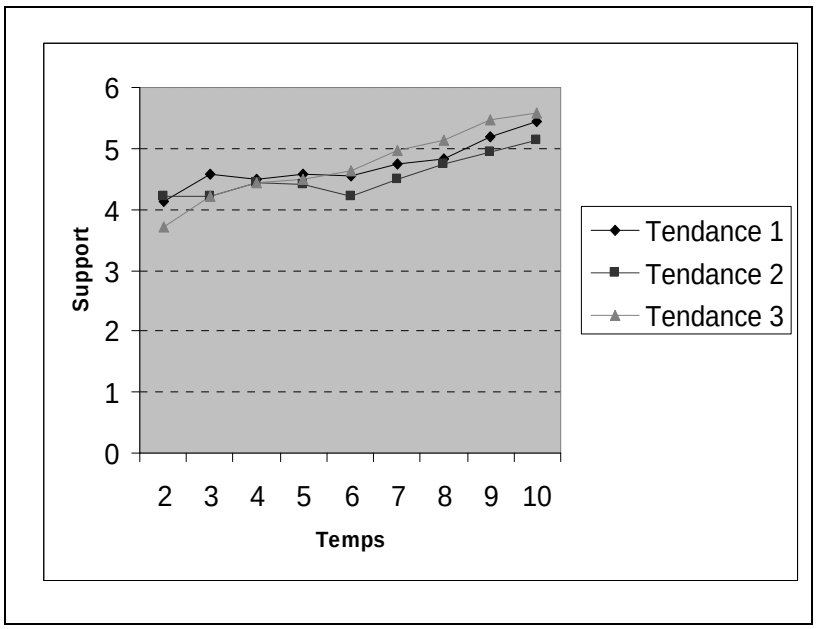


Figure 129 - Suivi de tendances sur les logs de AAE (2)

La Figure 128 et la Figure 129 illustrent des tendances correspondant respectivement à :

Tendance 1 : < ([⊕, PE_3 1]) ([⊗, /, 2]) ([⊗, /css/00.css, 2]) ([⊗, /css/01.css, 2]) ([⊥, /images/meter150.gif, 3]) >

Tendance 2 : < ([⊕, PE_1, 1]) ([⊥, /images/backgrd3.gif, 2]) ([⊥, /images/menu02.gif, 2]) ([⊥, /images/creuset.gif, 2]) ([⊥, /images/menu91.gif, 2]) ([⊥, /images/menu90.gif, 2]) ([⊥, /images/menu92.gif, 2]) >

Tendance 3 : < ([⊕, PE_2 1]) ([⊗, /css/00.css, 2]) ([⊥, /images/backgrd3.gif, 3]) ([⊥, /images/menu004a.gif, 3]) ([⊥, /images/meter050.gif, 3]) ([⊥, /images/menu003d.gif, 3]) >

Comme précédemment, nous pouvons constater dans la Figure 129 que les différentes tendances sont croissantes au cours du temps. Cependant, une analyse fine de la Figure 128 montre que, malgré les apparences, le comportement au cours du temps des utilisateurs n'est pas aussi croissant qu'il y parait et qu'il existe même des semaines pour lesquelles celui-ci devient décroissant (tendance 2).

Jeux de données d'un opérateur de téléphonie

Les derniers jeux de données concernent le commerce électronique et sont issus d'un opérateur de téléphonie mobile. Le premier jeu est récupéré directement depuis les fichiers access log et le second est obtenu à l'aide du système LISST [Bour03].
Le premier jeu de données est issu des fichiers log sur la période comprise entre le 14 et le 20 mars 2003. Il a une taille de 400 Mo, il contient 12000 identifiants de visiteurs différents, concerne 890 pages différentes et en moyenne le nombre de navigation est de 6 pages. Ce jeu de données est constitué par l'agrégation de deux fichiers logs. En effet le site, sur lequel nous avons effectué nos analyses, concerne la vente et la gestion de compte de téléphonie prépayé : le propriétaire d'un compte y vient pour consulter son compte, le recharger, bénéficier d'offres promotionnelles, etc … Ce site est réparti sur deux serveurs : le premier est consacré aux opérations courantes, et le second est utilisé pour la partie paiement sécurisé par carte bancaire. Chaque jour, en moyenne, un fichier log de 56 Mo est généré pour le premier et 5 Mo pour le second. De la même manière que dans les expériences précédentes, l'intégration a nécessité l'écriture d'analyseur lexicaux spécifiques afin de prétraiter les données. Nous avons, dans ce cadre, filtré l'intervention de processus automatique lancé par les serveurs eux mêmes (IP spécifique). Comme précédemment, nous avons, dans un premier temps, réalisé différentes extractions de connaissances pour des supports variant entre 2 et 30 %. Une partie des résultats de l'extraction est décrite dans la Figure 130.

En ce qui concerne les tendances, nous considérons par la suite que le fichier JOUR correspond au fichier log journalier, alors que le fichier CUMUL correspond au cumul des différentes informations, i.e. au cumul de tous les fichiers JOUR sur la période.

Support	Quelques structures extraites
30%	* < ([⊕, PE_1, 1]) ([⊗, /default.asp, 2]) >
20%	* < ([⊕, PE_1, 1]) ([⊗, /default.asp, 2]) ([⊗, Cob/lesminutes/EMinutes/Account/Manage.asp, 3]) >
15%	* < ([⊕, PE_1, 1]) ([⊗, /default.asp, 2]) ([⊗, Cob/lesminutes/EMinutes/Account/Manage.asp, 3]) > * < ([⊕, PE_2, 1]) ([⊗, Cob/lesminutes/EMinutes/Account/Manage.asp, 2]) ([⊕, lesminutes/Payment/SecurePay.asp, 3]) ([⊥, lesminutes/Payment/SecurePayAtLeast.asp, 4]) >
10%	* < ([⊕, PE_3, 1]) ([⊗, /Cob/lesminutes/EMinutes/Buy/PayBox.asp, 2]) ([⊕, lesminutes/Payment/SecurePay.asp, 3]) ([⊥, lesminutes/Payment/SecurePayAtLeast.asp, 4]) ([⊥, lesminutes/Payment/RedirectSite.asp, 4])> * < ([⊕, PE_1, 1]) ([⊗, /default.asp, 2]) ([⊕, lesminutes/Payment/SecurePay.asp, 3]) ([⊥, lesminutes/Payment/SecurePayAtLeast.asp, 4]) > * < ([⊕, PE_4, 1]) ([⊗, /Cob/lesminutes/Offer/TYPE_9/Offer.asp, 2]) > .....
5%	* < ([⊕, PE_1, 1]) ([⊗, /default.asp, 2]) ([⊗, Cob/lesminutes/EMinutes/Account/Manage.asp, 3]) ([⊗, /Cob/lesminutes/EMinutes/Buy/PayBox.asp, 4]) ([⊕, lesminutes/Payment/SecurePay.asp, 5]) ([⊥, lesminutes/Payment/SecurePayAtLeast.asp, 6) ([⊥, lesminutes/Payment/RedirectSite.asp, 6])> * < ([⊕, PE_1, 1]) ([⊗, /default.asp, 2]) ([⊗, Cob/lesminutes/EMinutes/Account/Manage.asp, 3]) ([⊥, /Cob/lesminutes/EMinutes/Account/ShowBillingCharge.asp, 4]) > ……
2%	* < ([⊕, PE_1, 1]) ([⊗, /default.asp, 2]) ([⊗, Cob/lesminutes/EMinutes/Account/Manage.asp, 3]) ([⊗, /Cob/lesminutes/EMinutes/Buy/PayBox.asp, 4]) ([⊕, lesminutes/Payment/SecurePay.asp, 5]) ([⊥, lesminutes/Payment/SecurePayAtLeast.asp, 6) ([⊥, lesminutes/Payment/RedirectSite.asp, 6]) ([⊗, Cob/lesminutes/EMinutes/Account/Manage.asp, 3])> * < ([⊕, PE_4, 1]) ([⊗, /Cob/lesminutes/Offer/TYPE_9/Offer.asp, 2]) ([⊗,/Cob/lesminutes/Offer/TYPE_9/OfferDelivery.asp, 3]) > .....

Figure 130 – Extrait de résultats sur les données de l'opérateur de téléphonie

La Figure 131 et la Figure 132 illustrent quelques exemples de tendances obtenues.

Tendance 1 : < ([⊕, PE_2, 1]) ([⊗, Cob/lesminutes/EMinutes/Account/Manage.asp, 2]) ([⊗, /Cob/lesminutes/EMinutes/Buy/PayBox.asp, 3]) ([⊕, lesminutes/Payment/SecurePay, 4]) >
La tendance 1 correspond en fait à une page de gestion du compte où l'utilisateur recharge son compte et arrive sur la page de paiement.

Tendance 2 : < ([⊕, PE_1, 1]) ([⊗, /default.asp, 2]) ([⊗, Cob/lesminutes/EMinutes/Account/ Manage.asp, 3]) ([⊗, /Cob/lesminutes/EMinutes/Buy/PayBox.asp, 4]) ([⊕, lesminutes/Payment/SecurePay.asp, 5]) ([⊥, lesminutes/Payment/SecurePayAtLeast.asp, 6) ([⊥, lesminutes/Payment/RedirectSite.asp, 6]) >.
Cette tendance correspond à une gestion de compte pour laquelle le client recharge son compte et paye. Le paiement est accepté et ensuite l'utilisateur est redirigé.

Tendance 3 : < ([⊕, PE_1, 1]) ([⊗, /default.asp, 2]) ([⊗, Cob/lesminutes/EMinutes/Account/Manage.asp, 3]) ([⊗, /Cob/lesminutes/EMinutes/Buy/P ayBox.asp, 4]) ([⊕, lesminutes/Payment/SecurePay.asp, 5]) ([⊥, lesminutes/Payment/ SecurePayAtLeast.asp, 6) ([⊥, lesminutes/Payment/RedirectSite.asp, 6]) ([⊗, Cob/lesminutes/EMinutes/Account/Manage.asp, 3])>
Cette dernière tendance est assez similaire à la précédente mais dans ce cas, les utilisateurs vont également consulter leur compte pour vérifier que le rechargement du compte s'est bien déroulé.

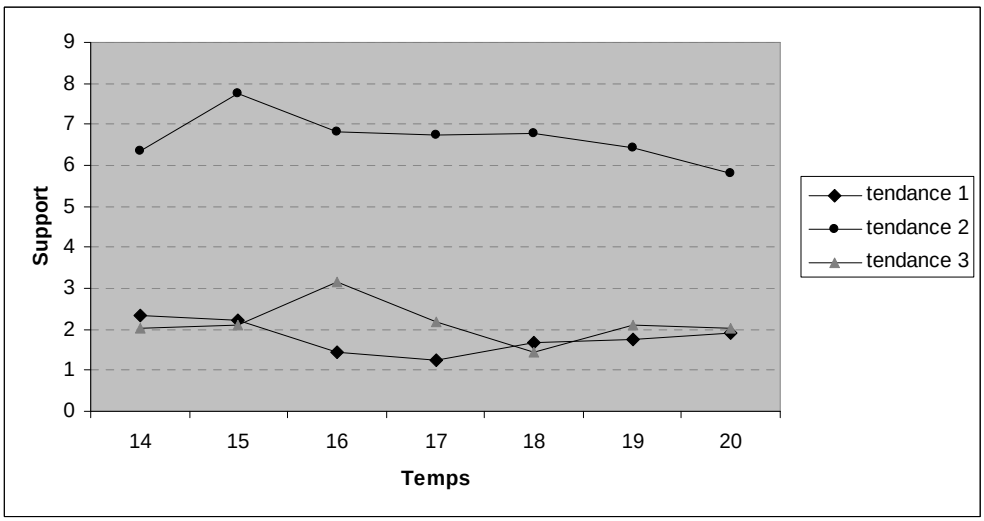


Figure 131 - Tendances sur le fichier JOUR

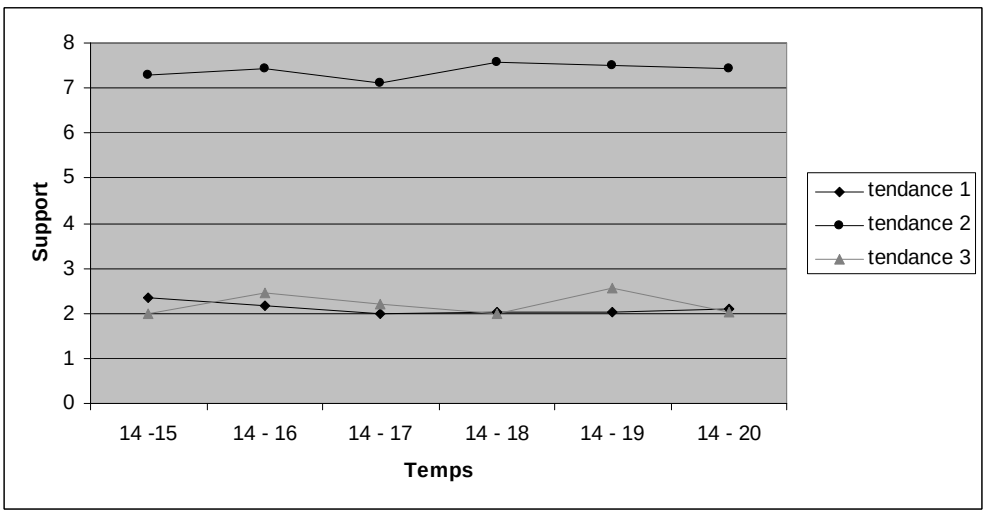


Figure 132 - Tendances sur le fichier CUMUL

La deuxième partie du jeu de données est issue d'une architecture logicielle LISST permettant de suivre le comportement d'un utilisateur en notant son passage sur des emplacements sémantiques prédéfinis sur un site Web.

Cette architecture représentée dans la Figure 131 est complémentaire au module de pré traitement d'AUSMS-Web. De manière générale, cette architecture permet d'étendre l'analyse des utilisateurs sur plusieurs sites distants (parcours inter sites et multi serveurs) afin d'obtenir les parcours les plus complets possibles [Bour03]. Le fichier de log résultant analysé concerne les parcours de 406 clients sur ce site le jeudi 22 Mai de 16h35 à 19h05. Le nombre de pages différentes durant ces navigations est de 163 et le nombre de pages visitées en moyenne est de 8. La taille du fichier est de 9Mo. La Figure 134 illustre quelques uns des résultats obtenus.

Etant donné que les parcours des utilisateurs sont plus longs (multi sites), nous pouvons constater que les comportements contigus sont plus significatifs.

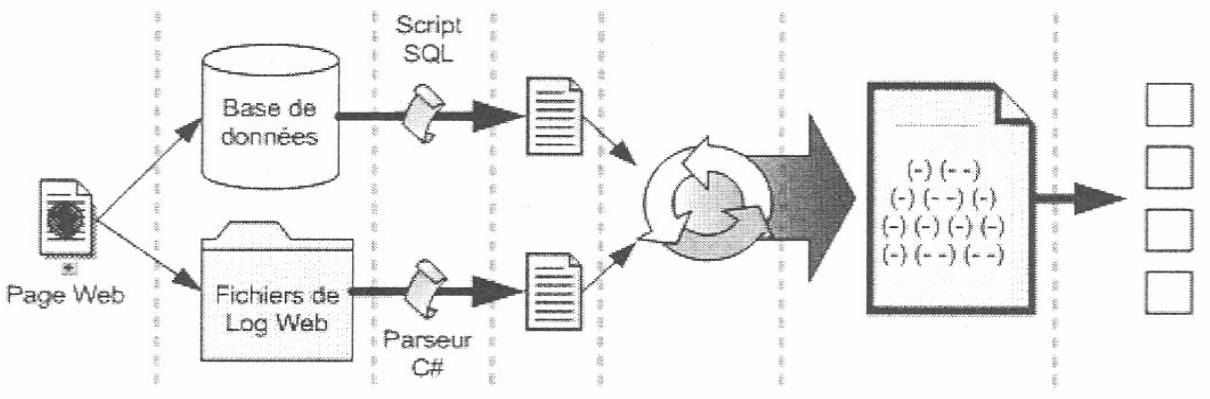


Figure 133 – Architecture LISST

Support	**Quelques structures extraites**
50%	* < ([⊕, PE_1, 1]) ([⊗, /default.asp, 2]) >
40%	* < ([⊕, PE_1, 1]) ([⊗, /default.asp, 2]) > * < ([⊕, PE_2, 1]) ([⊗, /cob/lesminutes/eminutes/account/manage.asp, 2]) >
30%	* < ([⊕, PE_1, 1]) ([⊗, /default.asp, 2]) ([⊥,/redirect/compteexistant.asp, 3]) ([⊗, /cob/lesminutes/eminutes/account/manage.asp, 3])>
20%	* < ([⊕, PE_1, 1]) ([⊗, /default.asp, 2]) ([⊥,/redirect/compteexistant.asp, 3]) ([⊗, /cob/lesminutes/eminutes/account/manage.asp, 3])> * < ([⊕, PE_1, 1]) ([⊗, /default.asp, 2]) ([⊥, /redirect/compteexistant.asp, 3]) > * < ([⊕, PE_1, 1]) ([⊗, /default.asp, 2]) ([⊗, /default.asp, 2]) > …
15%	* < ([⊕, PE_1, 1]) ([⊗, /default.asp, 2]) ([⊥,/redirect/compteexistant.asp, 3]) ([⊗,/cob/lesminutes/eminutes/buy/paybox.asp, 3])> * < ([⊕, PE_1, 1]) ([⊗, /default.asp, 2]) ([⊥,/redirect/eminutes.asp, 3]) ([⊥,/redirect/eminutes.asp, 3]) > * < ([⊕, PE_4, 1]) ([⊗,/cob/lesminutes/eminutes/dest.asp, 2]) > …
10%	* < ([⊕, PE_1, 1]) ([⊗, /default.asp, 2]) ([⊥, /redirect/compteexistant.asp, 3]) ([⊕, /lesminutes/cob/lesminutes/payment/securepay.asp, 3]) ([⊕, /les minutes/cob/lesminutes/payment/securepayatleast.asp, 4]) ([⊥, /redirect/ compteexistant.asp, 4]) ([⊗, /cob/lesminutes/eminutes/account/manage.asp, 2])> * < ([⊕, PE_3, 1]) ([⊗, /lesminutes/cob/lesminutes/payment/redirectsite.asp, 2]) ([⊥, /redirect/compteexistant.asp, 3]) ([⊗, /cob/lesminutes/eminutes/ account/manage.asp, 2]) ([⊗, /cob/lesminutes/eminutes/buy/paybox.asp, 3]) > ...
2%	* < ([⊕, PE_4, 1]) ([⊗,/cob/lesminutes/eminutes/dest.asp, 2]) ([⊥,/redirect/home.asp, 3]) ([⊗, /default.asp, 2]) > * < ([⊕, PE_1, 1]) ([⊗, /default.asp, 2]) ([⊥, /lesminutes/cob/lesminutes/ payment/redirectsite.asp, 3]) ([⊗, /cob/lesminutes/eminutes/ account/manage.asp, 2]) ([⊥, /redirect/compteexistant.asp, 3]) ([⊗, /cob/ lesminutes/eminutes/account/manage.asp, 2]) ([⊗, /cob/lesminutes/ eminutes/buy/paybox.asp, 3]) ([⊗, /cob/lesminutes/eminutes/buy/ payboxdelivery.asp, 4]) ([⊗, /cob/lesminutes/eminutes/account/account.asp, 2]) > …

Figure 134 - Exemples de structures

3 AUSMS un outil pour la recherche de modèles de vues fréquentes

Dans cette section, nous montrons comment nous avons adapté AUSMS pour aider le concepteur à définir un modèle de vues sous XML.

Dans une première partie, nous rappelons la problématique de l'intégration de données hétérogènes. Nous présentons dans la section 3.2, les différents travaux existants dans ces domaines. Enfin, dans la section 3.3, nous décrivons comment le système AUSMS peut être intégré à un système de définition de vues.

3.1 Problématique de l'intégration de données hétérogènes

Le langage XML a été proposé par le W3C (*World Wide Web Consortium*) comme format d'échange pour le Web. Il s'agit d'un langage de balisage qui permet de représenter des données semi structurées, i.e. des données dont la structure est irrégulière et auto décrite. Le succès d'XML et la popularité du Web font que la quantité de données disponibles dans ce langage est en constante augmentation. Comme nous l'avons vu précédemment dans le cas des données semi structurées, les données XML peuvent également ne pas être validées par une DTD ou un *XML-Schema*. Il existe donc souvent des similitudes dans la structure des éléments d'un document XML. L'analyse de ces régularités fournit des informations importantes qui peuvent être utilisées dans un processus d'aide à l'intégration de documents XML.
L'intégration de données consiste à fournir une vue unifiée de sources hétérogènes [ChGa94]. Pour cela, on utilise généralement un mécanisme de vues qui permet de restructurer les données des sources afin de construire un schéma médiateur. XML est souvent utilisé comme modèle commun pour l'intégration. La recherche de régularités dans la structure de sources XML permet alors construire des vues sur les sources, dans le but de permettre leur intégration. Le processus d'intégration n'est pas automatique mais la recherche de régularité permet de proposer une structure pour la vue qui sera personnalisée par le concepteur.

Les systèmes d'intégration de données fournissent une vue unifiée de sources hétérogènes à travers un schéma médiateur. Deux approches existent pour la construction de ce schéma (*GAV* et *LAV*). Elles reposent sur des mécanismes de vues [Hale03]. Avec l'approche *GAV* (*Global As View*) le schéma médiateur est défini comme un ensemble de vues sur les sources. A l'inverse, avec l'approche *LAV* (*Local As View*), les sources sont définies comme des vues du schéma médiateur. Le choix de l'approche à utiliser détermine des caractéristiques importantes du système. Avec *GAV* le traitement des requêtes est plus facile car il se fait de manière plus directe qu'avec *LAV* dans lequel la localisation des sources à utiliser est problématique. A l'inverse, l'ajout de nouvelles sources à intégrer est plus facile avec *LAV* car elle ne modifie pas le schéma médiateur.

Nous voyons donc la nécessité d'un modèle de vues pour XML. Tout d'abord, cela permet d'intégrer des sources XML. De plus, comme dans les SGBD, le mécanisme de vues peut être utilisé pour personnaliser des données, résoudre des problèmes de confidentialité, etc. Dans [BaBe01], les auteurs proposent un modèle de vues pour l'intégration : VIMIX (*View Model for Integration of Xml sources*). Cependant, la définition de vues XML est souvent difficile car la structure des données n'est pas toujours connue à l'avance. La problématique de l'intégration de sources de données hétérogènes consiste, à partir d'un ensemble de documents XML à rechercher les régularités structurelles pour aider le concepteur dans la définition de ses vues.

3.2 Aperçu des travaux antérieurs

Les systèmes d'intégration de données sont souvent basés sur des langages déclaratifs qui permettent de spécifier des vues sur les données des sources. A notre connaissance, il n'existe que peu de propositions pour automatiser ce processus. Les travaux existants dans ce domaine utilisent des techniques de traitement automatique du langage naturel qui permettent d'intégrer les données en étudiant leur sémantique. Un panorama de ces techniques est présenté dans [Gard02]. Ces approches utilisent généralement des fonctions lexicales qui permettent de calculer la distance sémantique entre les termes du schéma de la source et du schéma médiateur. Le système Xylème est un entrepôt de données qui permet d'intégrer les données XML du Web. Il comporte un module qui permet de générer des *mappings* entre la DTD « concrète » d'un document et une DTD « abstraite » modélisant un domaine d'intérêt [RaSi01].

Cependant, notre approche n'est pas directement comparable à ces travaux, principalement parce que notre problématique consiste à analyser la structure des données, sans se soucier de leur sémantique. Notre objectif n'est pas d'automatiser le processus d'intégration, mais de le faciliter en proposant de spécifier les données à extraire dans une source. Nous pensons que l'étape de restructuration des données doit être réalisée par un expert du domaine.

3.3 AUSMS-View : un système d'aide à la définition de vues

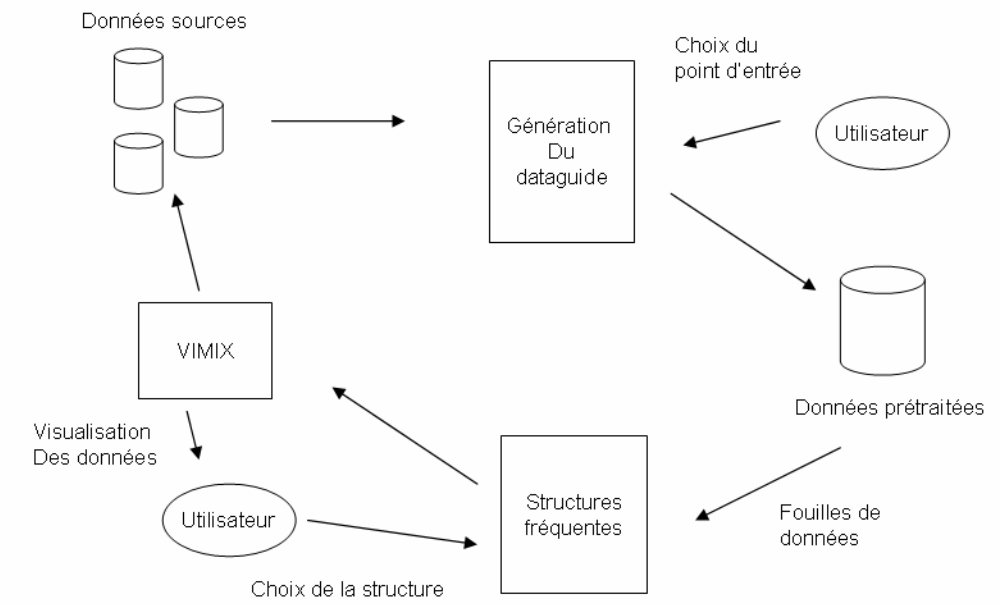


Figure 135 - Architecture fonctionnelle

La Figure 135, illustre le système que nous avons mis en place [LaBa04]. Les sources de données hétérogènes semi structurées, sous la forme de documents XML, sont, dans un premier temps, examinées pour permettre la construction automatique d'un *dataguide*. A partir de celui-ci l'utilisateur choisit un point d'entrée. Le système génère alors les différentes structures à étudier à partir de ce point d'entrée et des données sources. L'algorithme d'extraction détermine pour différentes valeurs de support quelles sont les différentes structures typiques. L'utilisateur choisit parmi ces différents motifs, celui qui va permettre la définition d'un motif sur une source pour construire une vue. Cette définition est envoyée au module VIMIX qui permet à l'utilisateur de visualiser les documents des sources après application de celle-ci.

Dans les sections suivantes, nous détaillons la définition des vues VIMIX ainsi que la construction de motifs sur les sources.

3.3.1 Le modèle de vues VIMIX

Le modèle de vues pour XML que nous utilisons permet de restructurer les données des sources pour les intégrer. Ce modèle appelé VIMIX (*VIew Model for Integration of Xml sources*) est présenté plus en détail dans [Bari03]. La définition de vues VIMIX s'effectue en 3 grandes étapes, illustrées par la figure suivante.

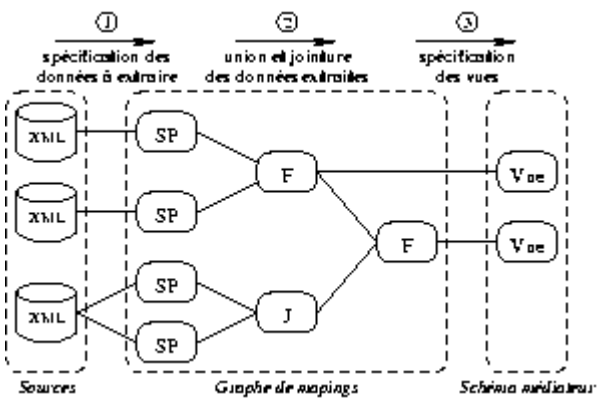


Figure 136 - Définition d'un schéma médiateur avec VIMIX

La première étape intitulée « spécification des données à extraire » consiste à définir des motifs sur les sources qui permettent d'associer des variables aux données des sources.
Les données à extraire sont spécifiées en utilisant une technique généralement appelée *pattern matching*. Cette technique consiste à spécifier un motif à retrouver dans une source pour définir des variables. L'instanciation de ces variables permettra d'extraire les données de cette source. Pour spécifier un motif sur une source avec VIMIX, on utilise :

- des axes de recherche qui permettent d'appliquer une fonction de navigation dans la structure des données,

- des nœuds qui permettent de décrire les éléments ou les attributs à rechercher dans la source.

Par exemple, le motif suivant permet de définir deux variables sur le *nom* et le *prénom* d'un auteur dans une source de données bibliographiques. L'utilisation des axes de recherche et des nœuds sur les sources permet de spécifier une forme à retrouver dans la structure du document 'bib.xml'. L'axe racine utilise la fonction *descendant* pour spécifier n'importe quel nœud '*auteur*' dans le document. L'axe de recherche du nœud *auteur* utilise la fonction *children* pour spécifier que l'élément doit avoir deux nœuds fils respectivement *nom* et *prénom*. Enfin, ces nœuds sont liés aux variables *n* et *p* pour extraire les données correspondantes dans le document source.

```
<source-pattern name='sp_auteurs' source='bib.xml'>
    <search-axis function='descendant'>
        <source-node reg-expression='auteur'>
            <search-axis function='children'>
                <source-node reg-expression='nom' bindto='n' />
                <source-node reg-expression='prenom' bindto='p' />
            </search-axis>
        </source-node>
    </search-axis>
</source>
```

D'un point de vue logique, les données extraites par un motif sur une source peuvent être représentées par une relation. Chaque variable définie par le motif est représentée par une colonne de la relation. Chaque instanciation de la forme des variables correspond à un tuple.

Cette représentation en relation permet de définir des opérations d'union et de jointure pour restructurer les données extraites. Enfin, la spécification du résultat de la vue permet de définir le schéma et les données.

3.3.2 Génération de motifs sur les sources

La construction de motifs sur les sources est souvent difficile car elle nécessite de connaître la structure des données. Pour cela, dans [BaBe01], un mécanisme d'aide basé sur la DTD d'une source (si elle existe), ou un *dataguide* [GoWi97] a été proposé. Un *dataguide* est un résumé de la structure des données dans lequel chaque chemin apparaît une seule fois. On peut donc utiliser ce dataguide (ou une DTD) pour proposer au concepteur de la vue tous les sous éléments ou attributs possibles pour un élément donné.

L'ensemble des structures à analyser est construit à partir d'une source de données XML en choisissant un point d'entrée dans le dataguide. La figure suivante illustre ce mécanisme.

L'exemple est basé sur une source de données bibliographiques. Le point d'entrée choisi dans le dataguide est l'élément auteur. Cela permettra de générer un motif sur la source décrivant les auteurs. Ce motif pourra être personnalisé par le concepteur de la vue après le processus de génération.

Une fois ce point d'entrée choisi, l'ensemble des éléments contenus dans la source de données est transformé en une collection de structures, i.e. une forêt d'arbres, sur laquelle nous allons appliquer l'algorithme PSP_{tree}.
Pour appliquer cet algorithme, l'utilisateur doit spécifier une valeur de support. Cette valeur permettra de déterminer quelles sont les structures fréquentes ou non. S'il n'a aucune idée de la valeur de support à choisir, l'utilisateur peut choisir une plage de support et un pas. Le système effectuera les différentes recherches de structures fréquentes pour les supports de cette plage. L'utilisateur disposera alors de plus de possibilités pour

choisir la structure typique qui sera utilisée pour spécifier la vue. Une fois les différents calculs terminés l'utilisateur sélectionne parmi les résultats un des motifs fréquents. Ce choix permettra après conversion de spécifier un motif sur la source pour définir une vue VIMIX. La phase de transformation est assez aisée. En effet, il suffit de traduire la structure fréquente choisie par l'utilisateur dans la syntaxe VIMIX.

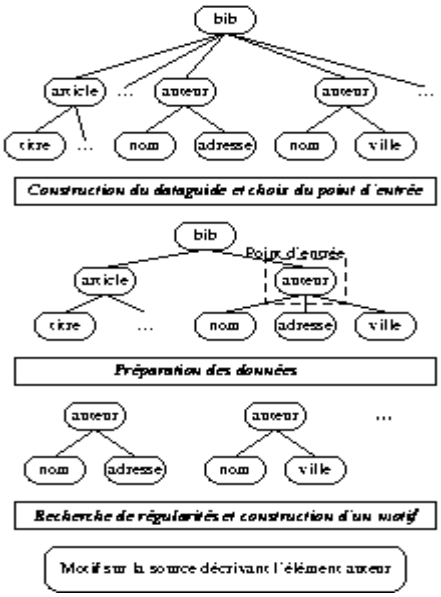


Figure 137 – Génération d'un motif sur une source

3.3.3 Expérimentation

Pour présenter l'utilité de notre approche, nous reprenons l'exemple de la base de données canadienne du Chapitre III. Cette base, disponible à l'adresse http://daryl.chin.gc.ca:8001/basisbwdocs/sid/title1f.html a été créée pour répondre aux besoins des gestionnaires des ressources culturelles chargés de l'information sur les épaves archéologiques. Elle contient des renseignements sur des navires qui ont été immatriculés au Canada ou qui ont navigué dans les eaux canadiennes. Dans un premier temps, nous avons converti les différentes sous bases sous la forme de source de données XML à l'aide d'un encodeur programmé en java. Une fois cette base intégrée à notre système, nous avons généré le dataguide associé et choisi le point d'entrée « navire » afin d'effectuer la recherche de structures fréquentes. Nous avons lancé ces recherches pour différentes valeurs de support. L'utilisateur peut alors choisir un motif VIMIX, le personnaliser et l'utiliser pour construire une vue utilisant cette source. La figure ci-dessous illustre un motif fréquent trouvé ainsi que sa génération automatique en VIMIX.

Support	Motif fréquent	Motif généré pour VIMIX
80%	{Navire:{NumCoque, Enregistrement: {DateEnr, PortEnr}, DescriptionNavire:{TypePoupe}}	<source-pattern name='sp-navires' source='navires.xml'> <search-axis function='children'> <source-node reg-expression='Navire'> <search-axis function='children'> <source-node reg-expression='NumCoque'/> <source-node reg-expression='Enregistrement'> <search-axis function='children'> <source-node reg-expression='DateEnr'/> <source-node reg-expression='PortEnr'/> </search-axis> </source-node> <Source-node reg-expression='DescriptionNavire> <search-axis function='children'> <source-node reg-expression='TypePoupe'/> </search-axis> </source-node> </search-axis> </source-node> </search-axis> </source-pattern>

4 Discussion

En illustrant deux problématiques différentes nous avons voulu montrer que l'aide à la détection de structures typiques pouvaient aider le concepteur dans ses choix et surtout qu'il existait de nombreux domaines d'application de telles techniques.

Ainsi, au cours de ce chapitre nous avons présenté comment les différents algorithmes décrits précédemment ont été intégrés et utilisés sur deux domaines d'applications différents [LaTe03c]. En ce qui concerne le Web Usage Mining, nous avons vu qu'il était possible avec le système AUSMS-Web de disposer d'un parcours beaucoup plus précis des utilisateurs. Bien entendu, ce parcours peut également être obtenu par un post traitement des résultats d'algorithmes de motifs séquentiels. Cependant, comme nous l'avons vu dans le Chapitre III, ce post traitement pénalise les temps de réponse. En privilégiant, les deux aspects : motifs séquentiels et motifs contigus, nous offrons à l'utilisateur la possibilité de s'intéresser aux deux problématiques en fonction de ses desiderata. Lors de l'analyse des comportements, nous avons vu qu'il était également possible d'examiner les tendances des différents utilisateurs. Pour l'instant, le système AUSMS-Web détermine tous les types de tendances sans les différencier. Nous reviendrons, dans le chapitre de conclusion, sur les différentes améliorations que nous souhaitons apporter au système pour qu'il puisse répondre à certains types de tendance. En outre, nous avons mis en évidence, qu'il devient indispensable pour suivre le comportement au cours du temps des utilisateurs, de mettre en œuvre une méthodologie d'analyse pour éviter que des comportements ne disparaissent au cours du temps car cachés dans la globalité des fichiers logs. Ainsi, dans notre cadre d'expérimentation, la méthodologie a consisté à ne pas prendre les fichiers logs dans leur globalité mais plutôt de manière régulière.

Dans le cas de l'aide à la définition de vues, nous avons illustré, au travers de ce chapitre, qu'il était possible d'aider le concepteur à spécifier les vues en lui proposant les parties de son schéma qui possèdent suffisamment de régularité. En rendant la génération des vues automatique, nous laissons donc le concepteur choisir les vues qui lui semblent les plus intéressantes dans son contexte d'application. Par manque de temps (pour chaque jeu de données, il est indispensable de définir un parseur spécifique), nous n'avons pas pu pousser plus en avant les expériences sur d'autres jeux de données. Cependant, les résultats obtenus à partir d'un jeu de données ont permis de montrer la faisabilité de l'approche. Bien entendu, un tel mécanisme est également applicable dans le cas de la génération d'index. Ainsi, en fonction de l'analyse du contenu d'un site Web ou de sa fréquence d'utilisation, il devient désormais possible d'automatiser la génération d'index et ainsi d'optimiser l'utilisation d'un site.

Chapitre VI - Conclusions et Perspectives

Le chapitre est organisé de la manière suivante. Dans la section 1, nous rappelons les principales contributions à la recherche de structures typiques et revenons sur les discussions réalisées au cours du mémoire. Nous présentons au cours de la section 2 les perspectives associées à ce travail.

1 Contributions

Au cours de ce mémoire nous avons abordé la problématique de la recherche de structures typiques au sein de grosses bases de données d'objets semi structurées.
Après avoir défini plus en détail au sein du Chapitre II, le type de données semi structurées que nous souhaitions rechercher, nous avons voulu également prendre en compte le problème lié à la pérennité des résultats obtenus. Ce problème, également connu sous le nom de mise à jour des connaissances extraites ou même fouille incrémentale est à l'heure actuelle un problème difficile quelles que soient les données manipulées. Au cours de ce chapitre, nous avons ainsi examiné les différents travaux existants autour de ces problématiques. Nous en avons également profité pour faire un rappel sur les travaux autour des règles d'association ainsi que sur les motifs séquentiels dans la mesure où la plupart des approches actuelles sont basées sur ces dernières. En ce qui concerne les motifs séquentiels, nous avons pu constater qu'ils semblaient être assez proches de notre problématique d'extraction. En effet, les données semi structurées peuvent être considérées comme des graphes qui eux-mêmes peuvent être vus comme des séquences particulières. Dans le Chapitre III, nous avons d'ailleurs montré que ces deux types de représentation étaient tout à fait bijectives. Malheureusement, nous avons également montré que les algorithmes de recherche de motifs séquentiels n'étaient pas adaptables à la recherche de sous structures fréquentes dans la mesure où les différents niveaux d'imbrications des données complexes ne sont pas pris en compte et ou la présence des sous arbres frères est délicate à appréhender. Il existe bien sûr quelques types de jeux de données (structures avec peu d'imbrication et surtout dans lesquelles il y a très peu de « remontées », i.e. il s'agit d'arbres dont les nœuds possèdent peu de frères) pour lesquels les motifs séquentiels sont adaptés et nous avons, pour ce type d'application, obtenu de bons résultats avec des algorithmes de recherche de motifs [LaMa00a, LaMa00b]. Cependant, dans le cas général, outre les problèmes énoncés précédemment, les approches des motifs généreraient un trop grand nombre de candidats, ce qui pénaliserait les performances de l'algorithme.
Ces constats nous ont amené à proposer une nouvelle approche appelée PSP_{tree} dont les principes généraux sont inspirés des algorithmes par niveaux : génération de candidats et test de ces candidats sur la base de données. Cependant, notre approche tient compte des types de données manipulées et optimise la génération des candidats de deux manières principales : utilisation d'une structure d'arbre préfixée et création de règles d'extension en tenant compte uniquement des extensions possibles pour un nœud de l'arbre. Nous avons montré au cours de différentes expérimentations que l'approche proposée était tout à fait adaptée à divers domaines d'applications. En effet, PSP_{tree} a été utilisé soit pour analyser le contenu de sites Web, i.e. dans un contexte de Web content mining, mais également pour analyser le parcours d'utilisateur de sites Web, i.e. Web usage mining. Pour analyser les performances de l'algorithme, nous avons également développé un générateur de bases de données semi structurées. Ce dernier a été utilisé pour générer différents jeux de données plus ou moins complexes (variation de la profondeur, du nombre de fils d'un nœud, etc.) et a permis de montrer les performances de notre algorithme notamment dans le cas de l'augmentation du nombre de données.
A l'issue de ce travail, nous avons souhaité relâcher, pour rendre notre approche plus générique, certaines contraintes inhérentes aux données manipulées. Dans ce cadre, nous avons jugé qu'il serait intéressant de rechercher des structures imbriquées quelque soit le niveau de profondeur de celles-ci. Cependant, nous avons considéré que les données recherchées devaient toujours respecter la même topologie au sein des arbres. En ce sens, notre approche est différente de celle proposée par M. Zaki dans TreeMiner [Zaki02]. La discussion du Chapitre III revient d'ailleurs sur ces différences. Pour prendre en compte le fait de libérer des contraintes, nous avons étendu PSP_{tree} pour proposer l'approche PSP_{tree}-GENERALISE. L'idée principale est d'utiliser la profondeur relative à la place de la profondeur réelle pour regrouper les règles d'extension possible d'un nœud. De la même manière que précédemment, nous avons étudié l'utilisation de l'algorithme PSP_{tree}-GENERALISE dans le cas de jeux de données réelles mais également ses performances dans le cas de jeux de données générées.

Nous avons ainsi pu constater qu'en relâchant les contraintes le comportement de l'algorithme possédait un comportement similaire au précédent (montée en charge, temps de réponse).

Etant donné que les données semi structurées peuvent évoluer au cours du temps, nous nous sommes posé la question de la maintenance des connaissances extraites. En effet, les jeux de données que nous avons utilisés pour expérimenter nos algorithmes étant issus du Web, il est évident que ceux-ci évoluent fréquemment. Dans le cas du Web usage mining, ce constat est encore plus flagrant étant donné que le nombre de visiteurs d'un site évolue régulièrement en fonction des usages. A partir de ce constat, le problème était de proposer une solution pour éviter de relancer les algorithmes précédents à chaque mise à jour. Nous avons pu constater, au cours du Chapitre III, que la recherche de structures typiques était coûteuse en temps de réponse. Dans le Chapitre II, nous avons recensé que les principales approches existantes pour maintenir la connaissance étaient dédiées soit aux règles d'association soit aux motifs séquentiels. Il n'existe rien à l'heure actuelle pour maintenir les connaissances sur des structures typiques malgré l'importance de ce domaine. Ainsi, dans le Chapitre IV, après avoir présenté les différents types de mise à jour possibles, nous avons proposé une approche basée sur l'utilisation de la bordure négative. Cette approche offre l'avantage de stocker la limite à partir de laquelle il est intéressant de relancer les algorithmes précédents. Au cours de ce chapitre, nous avons proposé un ensemble d'algorithmes permettant de maintenir la connaissance quelle que soit la mise à jour envisagée sur les données sources. Ces différents algorithmes ont été testés sur différents jeux de données afin de prouver que le maintien d'une bordure négative pouvait considérablement améliorer les temps de calcul pour obtenir une connaissance à jour. Au cours de la discussion du Chapitre II, nous avons également vu qu'il n'existait pas de solution parfaite pour maintenir la connaissance dans la mesure où de trop nombreux paramètres entrent en jeu (support, limite à partir de laquelle les éléments peuvent devenir fréquents, …). Les expériences menées avec nos algorithmes ont montré que globalement le maintien d'une bordure négative permettait d'améliorer considérablement les temps de réponse et offrait ainsi une solution à la prise en compte des modifications des données sources. Elles ont également montré les limites d'une telle approche et notamment que pour certains types de données, le maintien de la bordure n'était pas suffisant pour améliorer les performances. En ce sens, les conclusions du Chapitre II sont confirmées et montrent bien que l'utilisation d'algorithmes incrémentaux est très dépendante des jeux de données et des paramètres utilisés. Malgré cela, il est important de constater que dans la plupart des cas (les différentes expériences menées le confirment) une approche incrémentale est plus efficace que de recommencer l'extraction à partir de zéro.

Au cours du Chapitre V, nous avons montré comment les différents algorithmes précédents ont été intégrés au sein d'un prototype appelé AUSMS. Dans ce cadre, nous proposons au décideur un outil complet allant de la récupération de l'information à l'extraction des connaissances dans des bases de données semi structurées. Bien entendu, les algorithmes concernant les mises à jour des données sources sont pris en compte dans la mesure où ils sont également utilisés dans le cas d'un apprentissage automatique (utilisation des extractions effectuées précédemment dans le cas de modification de support). Deux domaines d'application privilégiés ont été plus particulièrement étudiés. Dans le premier cas, le Web usage mining, le système a été utilisé pour analyser le comportement des utilisateurs sur un ou plusieurs serveurs Web. Il a également été étendu à la prise en compte des comportements contigus et à l'étude des tendances des utilisateurs au cours du temps. Ainsi, nous avons pu démontrer qu'avec une étude du comportement contigu, notamment via la prise en compte des points d'entrée, le décideur dispose d'informations pertinentes pour améliorer les performances de son site. De la même manière, via la composante tendance, le décideur est à même de mieux comprendre comment évolue son site au cours du temps. De telles informations sont nécessaires à une amélioration d'un site et surtout à une adaptation efficace dans un cadre de e-commerce. Nous avons également montré au cours du Chapitre V que la création d'index ou de vues décrites dans le chapitre d'introduction était réalisable via AUSMS et que les informations proposées par notre système pouvaient faciliter la tâche du décideur.

Dans le cadre de ce mémoire, nous avons pu proposer de nouvelles approches qui sont efficaces pour extraire, maintenir et manipuler des données semi structurées. Les travaux menés ouvrent de nouvelles perspectives et de nombreuses pistes de recherche. Nous abordons quelques une de ces perspectives dans la section suivante.

2 Perspectives

2.1 Prise en compte d'un caractère joker

Dans [WaLi98, WaLi99], (C.f. Chapitre II) les auteurs s'intéressant à la recherche de structures typiques ont proposé l'utilisation d'un caractère joker pour ne pas tenir compte du niveau d'imbrication des sous arbres et pour étendre la problématique. Un tel caractère semble effectivement pertinent dans notre contexte dans la

mesure où il permet de rendre encore plus générique l'approche proposée. Si nous examinons plus attentivement par rapport à nos propositions les conséquences, il est évident que dans un premier temps les extensions proposées via l'algorithme PSPtree-GENERALISE sont adéquates dans la mesure où nous faisons également abstraction de la profondeur. Par contre, la prise en compte d'un joker induit également de prendre en compte le sens dans le cas de la filiation (actuellement de type père-fils uniquement étant donné le respect de le topologie) mais également, dans le cadre du nombre de niveaux, i.e. de fils, que l'on s'accorde à traverser. Un premier constat évident est que plus le nombre de jokers pris en compte par arbre est important plus le nombre de tests est également important et donc plus les temps de réponse sont étendus. Le second constat est que dans le cadre de notre proposition, seules les fonctions d'extension (génération des candidats) et de comptage (vérification des candidats) doivent être reconsidérées. De manière à illustrer les problématiques engendrées à la prise en compte de tels caractères, considérons l'exemple suivant :

Exemple 57 :

Considérons la base de données de la Figure 138, l'utilisation d'un caractère joker nous permettrait d'obtenir la séquence suivante : <(personne$_1$) (?) (nom$_3$) (prenom$_3$)>. Par ce principe, nous aimerions faire abstraction du niveau de profondeur de l'arbre et ainsi extraire toutes les séquences qui respectent ce critère.

Arbre_id	*Arbre*
T_1	*(personne$_1$) (identite$_2$) (nom$_3$) (prénom$_3$)*
T_2	*(personne$_1$) (etat_civil$_2$) (prénom$_3$) (nom$_3$)*
T_3	*(personne$_1$) (identite$_2$) (civil$_3$) (nom$_3$) (prénom$_3$)*

Figure 138 – Un exemple de base de données

En examinant les possibilités d'extension, nous obtiendrions le résultat suivant par rapport à PSPtree-GENERALISE :

Item	*Fils*	*Frères*	*Ancêtres*	*Joker*
Personne	*Identité (+1), etat_civil (+1)*			*Nom (?), Prénom (?)*
Identité	*Nom (+1), Prénom (+1)*			
Nom		*Prénom (0)*		
Prénom				
Etat_civil	*Nom (+1), Prénom (+1)*			
Civil	*Nom (+1), Prénom (+1)*			

Une première analyse rapide montre que les résultats obtenus dans ce cadre ressemblent à ceux proposés par [Zaki02], cependant, une étude plus approfondie est nécessaire de manière à examiner quelles sont toutes les conséquences sur nos algorithmes.

2.2 Optimisation de la gestion des évolutions

Nous avons vu dans le Chapitre V que la gestion des évolutions des données était prise en compte par l'utilisateur à un instant donné. La première limite de cette approche est que pendant un laps de temps qui peut être important les schémas extraits ne sont plus forcément représentatifs des données sources si celles-ci ont été modifiées. La seconde limite est liée au temps nécessaire pour prétraiter les données sources (téléchargement, transformation, …) de manière à vérifier s'il y a eu des modifications. L'une des perspectives de recherche serait d'automatiser le plus possible ce processus de manière à optimiser ainsi les traitements.
Pour résoudre le premier problème, une solution possible consiste à examiner de manière régulière, i.e. à intervalle régulier, et en arrière plan les données sources de manière à vérifier s'il y a eu des modifications. Etant donné que nous gérons la bordure négative, l'idée générale est de définir une structure qui puisse maintenir les schémas qui ne sont pas fréquents mais pour lesquels il suffit d'avoir peu de modifications pour devenir fréquents. Ainsi, en comparant les variations des données sources, nous pouvons automatiquement appliquer les calculs dès qu'un schéma devient fréquent (resp. non fréquent). Cependant, cette approche même si elle semble applicable ne tient pas compte du nombre et du type de modifications qui peuvent intervenir sur les données sources. En effet, si ces évolutions sont prises directement elles peuvent avoir un impact sur les connaissances extraites. Par contre, il se peut que sur un plus long terme elles s'annulent et ne nécessitent pas de modifier les connaissances extraites. L'exemple suivant illustre ce cas de figure.

Exemple 58 :

Considérons t, t_1, t_2 et t_3 des dates distinctes ordonnées. La date t représente la prise en compte pour la première fois des informations relatives à une source, les dates t_1 et t_2 représentent deux dates auxquelles nous avons recherché les évolutions. Considérons le sous arbre représenté par la séquence suivante au temps t : <(⊗Navire$_1$) (⊥NumOfficiel$_2$) (⊥Nationalité$_2$) (⊗Construction$_2$) (⊥VilleCons$_3$) (⊥ProvinceCons$_3$) (⊗Enregistrement$_2$) (⊥DateEnr$_3$) (⊥VilleEnr$_3$)(⊗DescrNavire$_2$) (⊥FigProue$_3$)>. A la date t_1, considérons l'ajout de (⊥PaysEnr$_3$) nous avons alors : <(⊗Navire$_1$) (⊥NumOfficiel$_2$) (⊥Nationalité$_2$) (⊗Construction$_2$) (⊥VilleCons$_3$) (⊥ProvinceCons$_3$) (⊗Enregistrement$_2$) (⊥DateEnr$_3$) (⊥VilleEnr$_3$) (⊥PaysEnr$_3$) (⊗DescrNavire$_2$) (⊥FigProue$_3$)>. Lorsque nous analysons la source à la date t_2, nous détectons la suppression de ce même (⊥PaysEnr$_3$). La séquence redevient alors dans son état initial. Sans analyse préalable, il sera nécessaire de prendre en compte d'une part l'ajout de (⊥PaysEnr$_3$) puis sa suppression. Or une analyse de la séquence à la date t_3 de l'application des mises à jour montre que celle-ci est identique à la séquence initiale.

Une piste de recherche possible serait alors de s'inspirer des travaux menés autour des « delta relations » des règles actives [ChAb98]. L'idée générale consiste en fait à analyser régulièrement les sources et à ne refléter que les effets de bords des modifications de la structure, i.e. de minimiser les opérations de mise à jour. Dans un premier temps, nous récupérons les delta relations à deux dates différentes ordonnées. Ainsi, pour un document donné, chacune des delta relations contient soit des ajouts, soit des suppressions. Dès que l'on souhaite répercuter les modifications, nous examinons les delta relations associées à un document. S'il existe une seule delta relation, alors il suffira de prendre en compte l'information contenue et de la répercuter. Par contre s'il existe plusieurs delta relations, celles-ci sont combinées dans une seule delta relation qui gère les effets de bords.

2.3 Optimisation de la maintenance des connaissances

Dans le cadre des opérations de mise à jour, nous avons pu constater que les travaux menés concernaient principalement certains nœuds de l'arbre des résultats. En effet, globalement les mises à jour se limitent à des mises à jour sur des nœuds très spécifiques. Nous devons ainsi soit supprimer les fils de ce nœud, dans ce cas les opérations sont assez simples et se résument à l'élimination des pointeurs et à la restitution de la mémoire. Mais dans le cas de l'extension d'un nœud, le processus est plus complexe et fait appel aux algorithmes décrits dans le Chapitre III. Nous avons réalisé quelques expériences de parallélisation qui consistaient à stocker les nœuds de l'arbre sous la forme de threads. Même si pour l'instant les opérations sont limitées à de la simulation (même machine et même processeur), nous avons pu constater qu'il est possible de répartir l'arbre sur différents processeurs, donc éventuellement sur différentes machines. Les premiers résultats ont montré qu'une telle solution permettait d'améliorer les performances des algorithmes. Il s'agit peut être d'une solution à la prise en compte des mises à jour qui offrirait une réponse plus optimisée, i.e. qui étendrait le champ d'application, aux différents problèmes que nous avons pu rencontrer.

2.4 Vers une analyse plus fine des tendances des utilisateurs au cours du temps

Dans le cadre du système AUSMS-Web, l'analyse de tendance définie ne permet pas, pour l'instant, à l'utilisateur de spécifier le choix de ses tendances. Une perspective possible serait de proposer un langage de description pour les tendances afin de proposer des résultats qui seraient plus en adéquation avec les desiderata de l'utilisateur. Si nous examinons attentivement les travaux de [LeAg97], nous pouvons constater que la possibilité de rechercher différents types de tendances (cycliques, répétitives, …) est basée sur l'utilisation d'un langage particulier de description (SHAPE) pour extraire les tendances. Il est clair qu'un tel langage est tout à fait adapté à notre contexte. Etant donné qu'à l'issue du processus, nous disposons de toutes les informations nécessaires, l'utilisation de ce langage permettrait d'interroger de manière plus précise les résultats et offrirait ainsi au décideur les tendances correspondant à ses critères.

2.5 Vers une nouvelle structure de données : quid des vecteurs de bits

Nous avons vu dans le Chapitre II que dans le cadre des motifs séquentiels, l'approche SPAM basée sur l'utilisation de vecteurs de bits était très efficace dans la mesure où les différentes opérations de gestion de candidats étaient réalisées par l'intermédiaire d'opérations logiques (AND) entre les vecteurs pour savoir si un candidat appartenait ou pas à la base de données. Une extension possible consiste donc à pousser plus en avant un tel type de représentation et notamment à intégrer la possibilité de représenter de manière complémentaire à la fois les niveaux d'imbrication mais également les différents sous arbres frères. Une analyse rapide de la prise en compte d'un joker comme nous l'avons décrit précédemment pourrait être réalisable avec l'utilisation de vecteurs de bits. Toutefois, cette perspective nécessite d'être étudiée plus avant.

Bibliographie

[AbQu97] S. Abiteboul, D. Quass, J. McHuch, J. Widom, and J. Wiener. “The Lorel Query Language for Semistructured Data”. In International Journal on Digital Libraries, 1(1), pp. 68-88, 1997.

[AgFa93] R. Agrawal, C. Faloutsos, and A. Swami. “Efficient Similarity Search in Sequence Databases”. In Proceedings of the 4th International Conference on Foundations of Data Organization and Algorithms (FODO’93), USA, 1993.

[AgIm93] R. Agrawal, T. Imielinski, and A. Swami. “Mining Association Rules between Sets of Items in Large Databases”. In Proceedings of the 1993 ACM SIGMOD Conference, pp. 207-216, Washington DC, USA, May 1993.

[AgPs95] R. Agrawal, G. Psaila, E. Wimmers and M. Zait. “Querying Shapes of histories”. In Proceedings of the 21st International Conference on Very Large Database (VLDB'95), pp. 502-514, Zurich, Switzerland, September 1995.

[AgPs95a] R. Agrawal and G. Psaila. “Active Data Mining”. In Proceedings of the 1st International Conference on Knowledge Discovery in Databases and Data Mining (KDD’95), pp. 3-8, Menlo Park, CA, USA, August 1995.

[AgSr94] R. Agrawal and R. Srikant. “Fast Algorithms for Mining Generalized Associations Rules”. In Proceedings of the 20th International Conference on Very Large Databases (VLDB’94), Santiago, Chile, September 1994.

[AgSr95] R. Agrawal and R. Srikant. “Mining Sequential Patterns”. In Proceedings of the 11th International Conference on Data Engineering (ICDE’95), pp. 3-14, Tapei, Taiwan, March 1995.

[AsAb02] T. Asai, K. Abe, S. Kawasoe, H. Arimura, H. Sakamoto and S. Arikawa. “Efficient Substructure Discovery from Large Semi-structured Data”. In Proceedings of the International Conference on Data Mining (ICDM’02), Washington DC, USA, April 2002.

[AyEl01] A. Ayad, N. El-Makky and Y. Taha. “Incremental Mining of Constrained Associations rules”. In Proceedings of the 1st SIAM International Conference on Data Mining, pp. 5-7, Chicago, IL, USA, April 2001.

[AyTa99] N. F. Ayan, A. U. Tanzel and E. Arkun. “An Efficient Algorithm to Update Large Itemsets with Early Pruning”. In Proceedings of the fifth ACM SIGKDD International Conference on Knowledge Discovery and Data Mining (KDD’99), pp. 287-291, San Diego, CA, USA, August 1999.

[BaBe01] X. Baril and Z. Bellahsène. “A Browser for Specifying XML Views”. In Proceedings of the 7th Internation Conference on Object Oriented Information Systems, pp. 165-174, 2001.

[BaGa96] R.A. Baeza-Yates and G.H. Gonnet, “Fast Text Searching for Regular Expressions or Automaton Searching on Tries”. Journal of ACM, 43(6), pp. 915-936, November 1996.

[Bari03] X. Baril. “Un modèle de vues pour intégrer des sources de données XML : VIMIX”. Thèse de Doctorat de l’Université Montpellier II, 2003.

[Baya98] R. J. Bayardo Jr. “Efficiently Mining Long Patterns from Databases”. In Proceedings of the International Conference on Management of Data (SIGMOD’98), pp. 85-93, Seattle, USA, June 1998

[Bour03] C. Bourrier. “Analyse Comportementale Temps Réel sur le Web”. Rapport de DEA de l’Université de Montpellier II, Juin 2003.

[BrFr84] L. Breiman, J.H. Friedman, R.A. Olshen, and C.J. Stone. “Classification and Regression Trees”. Belmont, 1984.

[BrMo97] S. Brin, R. Motwani, J.D. Ullman, and S. Tsur. “Dynamic Itemset Counting and Implication Rules for Market Basket Data”. In Proceedings of the International Conference on Management of Data (SIGMOD’97), pp. 255-264, Tucson, Arizona, May 1997.

[BrPa98] T. Bray, J. Paomi and C. M. Sperberg-McQueen. “Extensible Markup Language (XML)”. Version 1.0, W3C recommendation, Technical Report REC-xml-19980210, World Wide Web Consortium, February 1998.

[BuDa97] P. Buneman, S. Davidson, M. Fernandez and D. Suciu. “Adding Structure to Unstructured Data”. In Proceedings of the 6th International Conference on Database Theory (ICDT’97), pp. 336-350, Delphi, Greece, September 1997.

[ChAb98] S. Chawathe, S. Abiteboul, and J. Widom. “Representing and Querying Changes History in Semistructured Data”. In Proceedings of the International Conference on Data Engineering (ICDE’98), Orlando, USA, February 1998.

[ChGa94] S. Chawathe, H. Garcia-Molina, J. Hammer, K. Ireland, Y. Papakonstantinou, J. Ullman and J. Widom. “The TSIMMIS Project: Integration of Heterogeneous Information Sources”. In Proceedings of the 10th Meeting of the Information Processing Society of Japan, pp. 7-18, 1994.

[ChHa96] D. W. Cheung, J. Han, V.Ng and C. Y. Wong. “Maintenance of Discovered Association Rules in Large Databases: an Incremental Update Technique”. In Proceedings of the 12th International Conference on Data Engineering (ICDE’96), pp. 116-114, New Orleans, USA, February 1996.

[ChKa97] D. W. Cheung, B. Kao and J. Lee. *“A General Incremental Technique for Maintaining Discovered Association Rules”*. In Proceedings of the 5th International Conference on Database System for Advanced Applications (DASFA’97), pp. 185-194, Melbourne, Australia, April 1997.

[CoHo00] D. J. Cook and L. B. Holder. “Graph-based Data Mining”. Journal of IEEE Intelligent Systems, 15(2), pp. 32-41, 2000.

[CoMo97] R. Cooley, B. Mobasher and J. Srivastava. “Web Mining: Information and Pattern Discovery on the World Wide Web”. In Proceedings of the 9th IEEE International Conference on Tools with Artificial Intelligence (ICTAI’07), Newport Beach, CA, USA, November 1997.

[Cool00] R. Cooley. “Web Usage Mining: Discovery and Application of Interesting Patterns”. In Phd. Thesis, Graduate School of the University of Minnesota, MI, USA, 2000.

[DeTo98] L. Dehaspe, H. Toivonen and R. D. King. “Finding frequent substructures in chemical compounds”. In Proceedings of the 4th International Conference on Knowledge Discovery and Data Mining (KDD’98), pp. 30-36, New York, NY, USA, 1998.

[Dyre97] C. Dyreson. “Using an Incomplete Data Cube as a Summary Data Sieve”. In Bulletin of the IEEE Technical Committee on Data Engineering, pp. 19-26, March 1997.

[EsKr96] M. Ester, H.P. Kriegel, J. Sander, and X. Xu. “A Density-Based Algorithm for Discovering Clusters in Large Spatial Database with Noise“. In Proceedings of the 2nd International Conference on Knowledge Discovery and Data Mining (KDD’96), pp. 226-231, Portland, USA, 1996.

[FaPi96] U.M. Fayad, G. Piatetsky-Shapiro, P. Smyth, and P. Smyth. ”Advances in Knowledge Discovery and Data Mining”. AAAI Press, Menlo Park, CA, 1996.

[GaRa02] M. N. Garofalakis, R. Rastogi and K. Shim. « Mining Sequential patterns with Regular Expression constraints ». IEEE Transactions on Knowledge Data Engineering, Vol. 14, N°3, pp. 530-552, May/June, 2002.

[Gard02] G. Gardarin. “XML : des bases de données aux services Web”. Editions Dunod, 2002.

[GoMi90] M. Gondran et Michel Minoux. « Graphes et algorithmes ». Editions Eyrolles, 1990.

[GoWi97] R. Goldman and J. Widom. “DataGuides: Enabled Query Formulation and Optimization in Semi Structured Databases”. In Proceedings of the 23th International Conference on Very Large Databases (VLDB’97), pp. 436-445, 1997.

[GuRa98a] S. Guha, R. Rastori, and K. Shim. “CURE: An Efficient Clustering Algorithm for Large Databases”. In Proceedings of the International Conference on Management of Data (SIGMOD’98], pp. 73-84, Seattle, USA, 1998.

[GuRa98b] S. Guha, R. Rastori, and K. Shim. “ROCK: A Robust Clustering Algorithm for Categorical Attributes”. In Proceedings of the International Conference on Data Engineering (ICDE’98), pp. 345-366, Sidney, Australia, 1998.

[Hale03] H. Halevy. “Data Integration: A Status Report”. In Proceedings of the BTW’03 Conference, pp. 24-29, 2003.

[HaFu95] J. Han and Y. Fu. “Discovery of Multiple-Level Association Rules from Large Databases”. In Proceedings of the 21st International Conference on Very Large Databases (VLDB’95), pp. 420-431, Zurich, Switzerland, September 1995.

[HaKa01] J. Han and M. Kamber. “Data Mining – Concepts and Techniques”. Morgan Kauffmann Publishers, 2001.

[HaPe00] J. Han, J. Pei and Y. Yin. “Mining Frequent Patterns without Candidate Generation”. In Proceedings of the International Conference on Management of Data (SIGMOD’00), pp. 1-12, Dallas, Texas, 2000.

[HeMa00] I. Herman and M.S. Marshall. “Graph XML: An XML based graph interchange format”. Technical Report INS-R0009, Center for Mathematics and Computer Sciences (CWI), pp. 52-62, Amsterdam, 2000.

[HoOd00] A. Homer, M. Odhner and D. Sussman. “Application Center 2000 Pro”. In Wrox Edition, ISBN 1861004478, 2000.

[HoSw95] M. Houtsma and A. Swami. “Set Oriented Mining of Association Rules”. In Proceedings of the International Conference on Data Engineering (ICDE’95), pp. 25-33, Taiwan, 1995.

[HuKa98] Y. Huhtala, J. Karkkainen, P. Porkka, and H. Toivonen. “Efficient Discovery of Functional and Approximate Dependencies Using Partitions”. In Proceedings of 14th International Conference on Data Engineering (ICDE’98), pp. 392-401, USA, 1998.

[JaDu88] A.K. Jain and R.C. Dubes. “Algorithms for Clustering Data”. Prentice Hall, 1988.

[KaMa92] M. Kantola, H. Mannila, K.J. Raiha, and H. Sirtola. “Discovering Functional and Inclusion Dependencies in Relational Database”. In Journal of Intelligence Systems, pp. 591-607, 1992.

[KoBl00] R. Kosala and H. Blockeel. “Web Mining Research: A Survey”. In SIGKDD Explorations: Newsletter of the Special Interest Group (SIG) on Knowledge Discovery & Data Mining, ACM, Vol. 2, pp. 1-15, July 2000.

[LaMa00a] P.A. Laur, F. Masseglia, P. Poncelet and M. Teisseire. “A General Architecture for Finding Structural Regularities on the Web”. In Proceedings of the 9th International Conference on Artificial Intelligence (AIMSA’00), Varna, Bulgaria, LNAI, pp. 179-188, Vol. 2424, September 2000.

[LaMa00b] P.A. Laur, F. Masseglia and P. Poncelet. “Schema Mining: Finding Structural Regularity among Semi Structured Data”. In Proceedings of the 4th International Conference on Principles and Practice of Knowledge Discovery in Databases (PKDD’2000), short paper, Lyon, France, LNAI, pp. 498-503, Vol. 1910, September 2000.

[LaPo03] P.A. Laur et P. Poncelet. “AUSMS : un environnement pour l’extraction de sous structures fréquentes dans une collection d’objets semi structurés”. In Actes des 3ièmes Journées d’Extraction et Gestion des Connaissances (EGC’03), pp. 449-459, Lyon, France, Janvier 2003.

[LaTe03a] P.A. Laur, M. Teisseire and P. Poncelet. “AUSMS: An Environment for Frequent Sub Structures Extraction in a Semi Structured Object Collection”. In Proceedings of the 14th International Conference on Database and Expert Systems Applications (DEXA’03), pp. 38-35, LNCS, Vol. 2736, September 2003.

[LaTe03b] P.A. Laur, M. Teisseire and P. Poncelet. “Web Usage Mining: Extraction, Maintenance and Behaviour Trends”. In Proceedings of the 1st Indian International Conference on Artificial Intelligence (IICAI’03), pp. 14-22, Hyderabad, India, December 2003.

[LaTe03c] P.A. Laur, M. Teisseire et P. Poncelet. « Données semi structurées : extraction, maintenance et analyse de tendances ». In Revue Ingénierie des Systèmes d’Information (ISI), numéro spécial « Bases de Données Semi Structurées », pp. 49-78, Vol. 8, N°5/6, 2003.

[LaBa04] P.A. Laur et X. Baril. « Découverte de régularités pour l’intégration de données semi structurées ». In Actes des 4ièmes Journées d’Extraction et Gestion des Connaissances (EGC’04), pp. 537-542, Clermont Ferrand, France, Janvier 2004.

[LeAg97] B. Lent, R. Agrawal and R. Srikant. “Discovering Trends in Text Databases”. In Proceedings of the 3rd International Conference on Knowledge Discovery (KDD’97), pp. 227-230, Newport Beach, California, August 1997.

[LiLe98] M. Y. Lin and S. Y. Lee. “Incremental Update on Sequential Patterns in Large Databases”. Proceedings of the 10th International Conference on Tools for Artificial Intelligence (TAI’98), pp. 24-31, May 1998.

[LiLe03] M. Y. Lin and S. Y. Lee. “Improving the Efficiency of Interactive Sequential Pattern Mining by Incremental Pattern Discovery”. Proceedings of the 36th Hawaii International Conference on System Sciences (HICSS’03), pp. 24-31, Big Island, Hawaii, USA, January 2003.

[LoPe00] S. Lopez, J.-M. Petit, and L. Lakhal. “Discovery of Functional Dependencies and Armstrong Relations”. In Proceedings of the 7th International Conference on Extending Database Technology (EDBT’00), pp. 350-364, Germany, March 2000.

[MaHe00] M. Marshall, I. Herman and G. Melançon. “An Object-Oriented Design For Graph Visualisation”. Technical Report INS-R00001, Center for Mathematics and Computer Sciences (CWI), Amsterdam, 2000.

[MaHo99] T. Matsuda, T. Horiuchi, H. Motoda, T. Washio, K. Kumazawa and N. Arai. “Graph Based Induction for General Graph Structured Data”. In Proceedings of the DS’99 Conference, pp. 340-342, 1999.

[MaPo99a] F. Masseglia, P. Poncelet and R.Cicchetti. “An efficient Algorithm for Web Usage Mining”. In Networking and Information Systems Journal (NIS), Vol. 2, N° 5-6, pp. 571-603, December 1999.

[MaPo99b] F. Masseglia, P. Poncelet and R.Cicchetti. “Analyse du Comportement des Utilisateurs sur le Web”. Actes du 17ième Congrès Informatique des Organisations et Systèmes d’Information et de Décision (INFORSID’99), Toulon, France, Juin 1999.

[MaPo99c] F. Masseglia, P. Poncelet and R.Cicchetti. “WebTool : An Integrated Framework for Data Mining”. In Proceedings of the 10th International Conference on Database and Expert Systems Applications (DEXA’99), pp. 892-901, Florence, Italy, September 1999.

[Mass02] F. Masseglia. “Algorithmes et applications pour l’extraction de motifs séquentiels dans le domaine de la fouille de données : de l’incrémental au temps réel”. In Thèse de l’Université de Versailles Saint Quentin, Janvier 2002.

[MaTo94] H. Mannila, H. Toivonen, and A. I. Verkano. “Improved Methods for Finding Association Rules”. Technical Report, University of Helsinski, Finland, February 1994.

[MaTo97] H. Mannila, H. Toivonen and A.I. Verkamo. “Discovery of Frequent Episodes in Event Sequences”. In Data Mining and Knowledge Discovery, Vol. 1, N° 3, pp. 259-289, September 1997.

[MeMi96] A.O. Mendelzon, G.A. Mihaila and T. Milo. “Querying the World Wide Web”. In PIDS Conference, International Journal on Digital Libraries, Vol. 1, N°1, pp. 54-67, 1996.

[MeMo02] A. Méndez-Torreblanca, M. Montes-y-Gômez and A. Lopez-Lopez. “A Trend Discovery System for Dynamic Web Content Mining”. In Proceedings of the 11th International Conference on Computing (CIC’02), Mexico City, Mexico, November 2002.

[MiSh01a] T. Miyahara, T. Shoudai, T. Uchida, K. Takahashi and H. Ueda. “Discovery of Frequent Tree Structured Patterns in Semistructured Web Documents“. In Proceedings of the Pacific-Asia Conference on Knowledge Discovery and Data Mining (PAKDD’01), pp. 47-52, Hong Kong, China, April 2001.

[MiSh01b] T. Miyahara, T. Shoudai and T. Uchida. “Discovery of Maximally Frequent Tag Tree Patterns in Semistructured Data”. In Proceedings of the LA Winter Symposium, Kyoto, Japan, 2001.

[MoGe01] M. Montes y Gomez, A. Gelbukh and A. Lopez-Lopez. “Mining the news trends, associations and deviations”. In Computacion y Sistemas, pp. 14-24, Vol. 5, No 1, July-September 2001.

[MoJa96] B. Mobasher, N. Jain, E. Han and J. Srivastava. “Web Miner: Pattern Discovery from World Wide Transactions”. Technical Report TR96-050, Department of Computer Science, University of Minnesota, USA, February 1996.

[Mort99] B. Mortazavi-Asl. “Discovering and Mining User Web-Page Transversal Patterns”. In PDH Thesis, 1999.

[NeAb98] S. Nestorov, A. Abiteboul and R. Motwani. “Extracting Schema from Semistructured Data”. In Proceedings of the International Conference on Management of Data (SIGMOD’98), pp. 295-306, SIGMOD RECORD, Volume 27, Number 2, 1998.

[NeUl97] S. Nestorov, J. Ullman, J. Wiener and S. Chawathe. “Representative Objects: Concise Representations of Semistructured, Hierarchical Data”. In Proceedings of the 13th International Conference on Data Engineering (ICDE’97), pp. 79-90, Birmingham, UK, April 1997.

[PaBa99] N. Pasquier, T. Bastide, R. Taouil, and L. Lakhal. “Discovering Frequent Closed Itemsets for Association Rules Using Closed Itemset Lattices”. In Proceedings of the International Conference on Database Theory (ICDT’99), pp. 398-416, 1999.

[PaZa99] S. Parthasarathy and M. J. Zaki. “Incremental and Interactive Sequence Mining”. In Proceedings of the Conference on Information and Knowledge Management (CIKM’99), pp. 251-258, Kansas City, USA, November 1999.

[PeHa01] J. Pei, J. Han, B. Mortazavi-Asl, H. Pinto, Q. Chen, U. Dayal, and M.C. Hsu. “PrefixSpan: Mining Sequential Patterns Efficiently by Prefix Projected Pattern Growth”. In Proceedings of the 17th International Conference on Data Engineering (ICDE’01), pp. 215-226, Heidelberg, Germany, April 2001.

[PuKr01] J. R. Punin, M. S. Krishnamoorthy and M. J. Zaki. “Web Usage Mining – Languages and algorithms”. In Studies in Classification, Data Analysis, and Knowledge Organization, Springler Verlag, 2001.

[RaFa94] D.M. Ranganathan, C. Faloutsos, and Y. Manolopoulos. “Fast Sequence Matching in Time Series Database”. In Proceedings of the International Conference on Management of Data (SIGMOD’94), pp. 419-429, Mineapolis, MI, USA, June 1994.

[RaMo96] C. P. Rainsford, M. K. Mohania and J. F. Roddick. “Incremental Maintenance Techniques for Discovered Classification Rules”. In Proceedings of the International Symposium on Cooperative Database Systems for Advanced Applications (CODAS), pp. 302-305, Kyoto, Japan, 1996.

[RaMo97] C. P. Rainsford, M. K. Mohania and J. F. Roddick. “A Temporal Windowing Approach to the Incremental Maintenance of Association Rules”, In Proceedings of the 8th International Database Workshop, Data Mining, Data Warehousing and Client/Server Databases (IDW’97) (J. Fong., ed.), pp. 78-94, Springer Verlag, Hong Kong, 1997.

[RaSi01] C. Raynaud, J.P. Sirot and D. Vodislav. “Semantic Intefration of Heterogeneous XML Data Sources”. In Proceedings of the International Database Engineering and Applications Symposium, pp. 199-208, 2001.

[RiFl98] I. Rigoutos and A. Floratos. “Combinatorial Pattern Discovery In Biological Sequences: the teiresias algorithm”. In bioinformatics, 4(1), pp. 55-67, 1998.

[Riss89] J. Rissanen. “Stochastic Complexity in statistical inquiry”. World Scientific Journal, Singapore, 1989.

[SaOm95] A. Savasere, E. Omiecinski, and S. Navathe. “An Efficient Algorithm for Mining Association Rules in Large Databases”. In Proceedings of the 21st International Conference on Very Large Databases (VLDB’95), pp. 432-444, Zurich, Switzerland, September 1995.

[SaSr98] N. L. Sarda and N. V. Srinivas. “An Adaptive Algorithm for Incremental Mining of Association Rules”. In Proceedings of the 9th International Workshop on Database and Expert Systems Applications (DEXA’98), Vienna, Austria, August 1998.

[ShAg96] J.C. Shafer, R. Agrawal and M. Mehta. “SPRINT: A Scalable Parallel Classifier for Data Mining”. In Proceedings of the 22th International Conference on Very Large Databases (VLDB’96), Bombay, India, September 1996.

[SpFa98] M. Spiliopoulou and L. C. Faulstich. “WUM: A Web Utilization Miner”. In Proceedings of the 6th International Conference on Extending Database Technology (EDBT’98) Workshop, International Workshop on the Web and Databases (WebDB’98), pp. 184-203, Valencia, Spain, March 1998.

[SrAg95] R. Srikant and R. Agrawal. “Mining Generalized Association Rules”. In Proceedings of the 21st International Conference on Very Large Databases (VLDB’95), pp. 407-419, Zurich, Switzerland, September 1995.

[SrAg96] R. Srikant and R. Agrawal, “Mining Sequential Patterns: Generalizations and Performance Improvements”. In Proceedings of the 5th International Conference on Extending Database Technology, pp. 3-17, Avignon, France, 1996.

[SrCo00] J. Srivastava, R. Cooley, M. Deshpande and P. Tan. “Web Usage Mining: Discovery and Applications of Usage Patterns from Web Data”. In SIGKDD Explorations Journal, Vol. 1, N° 2, pp. 12-23, 2000.

[ThBo97] S. Thomas, S. Bodagala, K. Alsabti and S. Ranka. “An Efficient Algorithm for the Incremental Updating of Association Rules in Large Databases”. In Proceedings of the third International Conference on Knowledge Discovery and Data Mining (KDD’97), pp. 263-266, Newport Beach, CA, USA, August 1997.

[Toiv96] H. Toivonen. “Sampling Large Databases for Association Rules”. In Proceedings of the 22nd International Conference on Very Large Databases (VLDB’96), pp. 134-145, Bombay, India, September 1996.

[W3C03] W3C, Extensible Markup Language (XML), http://www.w3.org/XML, Juin 2003.

[WaCh94] J.T. Wang, G-W. Chirn, T. Marr, B. Shapiro, D. Shasha and K. Zhang. “Combinatorial Pattern Discovery for Scientific Data: Some Preliminary Results”. In Proceedings of the 1994 ACM SIGMOD Conference on Management of Data, pp. 115-125, Minneapolis, USA, May 1994.

[WaHa04] J. Wang and J. Han. “BIDE: Efficient Mining of Frequent closed Sequences”. In Proceedings of the International Conference on Data Engineering ICDE’04, Boston, MA, USA, Mars 2004.

[WaLi97a] K. Wang and H. Liu. “Mining Nested Association Patterns”. In Proceedings of SIGMOD’97 Workshop on Research Issues on Data Mining and Knowledge Discovery, May 1997.

[WaLi97b] K. Wang and H. Liu. ”Schema Discovery for Semi-structured Data“. In Proceedings of the International Conference on Knowledge Discovery and Data Mining (KDD’97), pp. 271-274., Newport Beach, USA, August 1997.

[WaLi98] K. Wang and H. Liu. “Discovering Typical Structures of Documents: A Road Map Approach”. In Proceeding of the Conference on Research and Development in Information Retrieval (SIGIR’98), pp. 146-154, Melbourne, Australia, August 1998.

[WaLi99] K. Wang and H. Liu. “Discovering Structural Association of Semistructured Data”. In IEEE Transactions on Knowledge and Data Engineering, pp. 353-371, January 1999.

[WaSh96] J.TL. Wang, B.A. Shapiro, D. Shasha, K. Zhang and C.Y Chang. “Automated Discovery of Active Motifs in Multiple RNA Secondary Structures”. In Proceedings of the International Conference on Knowledge Discovery and Data Mining (KDD’96), pp. 70-75, 1996.

[Webb00] Webbot and Libwww, W3C Sample Code Library, http://www.w3.org/Library/, 2000.

[WeKu91] S.M. Weiss and C.A. Kulikowski. “Computer System that Learn: Classification and Prediction Methods from Statistics, Neural Nets, Machine Learning, and Expert Systems”. Morgan Kaufman, 1991.

[Work97] The Workshop on Management of Semistructured Data, In www.research.att. com/suciu/workshop-papers.html, Tucson, AR, USA, May 1997.

[YaHa02] X. Yan and J. Han. “gSpan: Graph-Based Substructure Pattern Mining “. In Proceedings of the IEEE International Conference on Data Mining (ICDM’02), Maebashi City; Japan, December 2002.

[YeCh01] S. Yen and C. Cho. “An Efficient Approach for Updating Sequential Patterns Using Database Segmentation”. International Journal of Fuzzy Systems, Vol.3, No 2, June 2001.

[Zaki00] M. Zaki. “Scalable algorithms for association mining”. IEEE Transactions on Knowledge and Data Engineering, 12 (3), pp. 372-390, May-June 2000.

[Zaki01] M. J. Zaki. “SPADE: An Efficient Algorithm for Mining Frequent Sequences”. In Machine Learning Journal, Special Issue on Unsupervised Learning (Doug Fisher ed.), Vol. 42, N° 1-2, pp. 31-60, January 2001.

[Zaki02] M. Zaki. “Efficiently Mining Frequent Trees in a Forest “. In Proceedings of the International Conference on Knowledge Discovery and Data Mining (SIGKDD’02), pp. 71-80, Edmonton, Canada, July 2002.

[ZaXi98] O. R. Zaïne, M. Xin and J. Han. “Discovering Web Access Patterns and Trends by Applying OLAP and Data Mining Technology on Web Logs”. In Proceedings on Advances in Digital Libraries Conference (ADL’98), pp. 19-29, Santa Barbara, CA, USA, April 1998.

[ZhXu02a] Q. Zheng, K. Xu, S. Ma and W. Lv. “The Algorithms of Updating Sequential Patterns”. In Proceedings of the International Conference on Data Mining (ICDM’02), Washington DC, USA, April 2002.

[ZhXu02b] Q. Zheng, K. Xu, S. Ma and W. Lv. “Intelligent Search of Correlated Alarms from Database Containing Noise Data”. In Proceedings of the 8th International IFIP/IEEE Network Operations and Management Symposium (NOMS’02) , pp. 405-419, Florence, Italy, April 2002.

Résumé

Avec la popularité du web, les sources d'informations sont multiples et les formats de données hétérogènes n'obéissent plus à une structure rigide. La cohabitation est rendue possible par les données semi structurées incluant leur schéma et pour lesquelles les contraintes d'une représentation relationnelle sont relaxées. C'est dans ce contexte que se situe notre proposition : «identifier des structures typiques, par le biais de méthodes adaptées de fouilles de données, au sein d'une collection de données semi structurées».

Nous proposons une nouvelle approche appelée PSPtree, inspirée des motifs séquentiels, ainsi que sa généralisation PSPtree-GENERALISE. Ces algorithmes ont été intégrés au sein du système AUSMS (Automatic Update Schema Mining System) afin de collecter les données, de rechercher les sous-structures fréquentes et de maintenir les connaissances extraites suite aux évolutions des sources mais également d'analyser les tendances ainsi que visualiser les résultats.

Mots clés*:* données semi structurées, fouille de données, extraction de connaissances, évolution des sources de données, analyse de tendance.

Abstract

With the growing popularity of the World Wide Web, information sources are numerous and heterogeneous data type doesn’t follow a strict structure. The living together is allowed with semi structured data that include their schema and for which relational representation constraints have been released. Within this here is our proposal: “identify typical structures, by the way of fitted data mining methods, within a collection of semi structured data”.

We suggest a new approach named PSPtree, inspired by sequential patterns, and it generalization named PSPtree-GENERALISE. These algorithms have been fitted within the AUSMS system (Automatic Update Schema Mining System) in order to collect data, research frequent sub structures and maintain extracted know ledges following sources evolutions but also trends analysis and results visualization.

Keywords: semi structured data, data mining, knowledge extraction, data sources evolution, trend analysis.