Nř d'ordre : XXXX

Thèse

de

L'Université Paris–Sud

présentée en vue de l'obtention du

Doctorat de l'Université Paris–Sud

Spécialité : Informatique – Intelligence Artificielle

Par

Alexandre Termier

Extraction d'arbres fréquents dans un corpus hétérogène de données semi-structurées : application à la fouille de documents XML

Soutenue le 28 avril 2004 devant la commission d'examen :

M.	Serge Abiteboul	Directeur de Recherche	Président du jury
Mme	Marie-Christine Rousset	Professeur	Directrice de thèse
Mme	Michèle Sebag	Directeur de Recherche	Directrice de thèse
M.	Marc Sebban	Professeur	Rapporteur
M.	Hannu Toivonen	Professeur	Rapporteur
M.	Jean-Daniel Zucker	Professeur	Rapporteur

Laboratoire de Recherche en Informatique, U.M.R. CNRS 8623,
Université Paris-Sud, 91405 Orsay Cedex, France

Table des matières

Chapitre I

Introduction

1 La fouille de données

1.1 Des origines ...

Au XVIII$^{\text{ème}}$ siècle, une nouvelle école de pensée scientifique émergea. On peut en trouver les fondations dans "Le discours de la méthode" de René Descartes. Une des bases de cette nouvelle façon de faire de la science était de faire passer la raison avant l'expérience des sens. En somme, le triomphe de l'objectif sur le subjectif. Les scientifiques de l'époque entreprirent de jeter les bases d'une nouvelle compréhension du monde. Beaucoup des théories qui nous paraissent aquises de nos jours sont nées à ce moment-là. Les plantes furent pour la première fois classées de manière systématique. C'est la classification de Carl Von Linné, encore utilisée aujourd'hui. Grâce à la naissance de l'anatomie (Cuvier) et des méthodologies d'observation rigoureuses, les animaux purent également être classés en familles. L'appartenance d'un animal à une de ces familles étant déterminée par la présence ou l'absence de certains caractères. De coûteuses et périlleuses expéditions furent lancées aux quatre coins du globe pour découvrir de nouveaux spécimens d'êtres vivants et les ajouter aux classifications existantes. Les méthodes employées firent leurs preuves : si par exemple on trouvait un nouvel animal ayant les caractères : { tétrapode, poilu, squelette central }, on pouvait affirmer qu'il appartenait à la classe des Mammifères. Et donc, on pouvait en déduire qu'il devait être vivipare, comme tous les mammifères à l'exception de l'echidné et de l'ornythorinque.

1.2 ... à nos jours

Au XX$^{\text{ème}}$ siècle l'ordinateur permit l'apparition de nouveaux moyens de gérer l'information : les *Systèmes de Gestion de Bases de Données (SGBD)*. Ces SGBD permettent de stocker et d'interroger efficacement de grandes quantités de données, à condition que ces données se conforment à un schéma précis défini à l'avance.

Ainsi, d'immenses quantités de données de type *transactionnel* se mirent à peupler les SGBD d'organisations commerciales. Une donnée transactionnelle (ou *transaction*) est simplement la description d'une vente : un client a acheté les produits X, Y et Z. Un exemple concret est le ticket de caisse, qui, s'il finit sa vie de papier au fond d'une poubelle, reste par contre stocké très longtemps dans les ordinateurs du supermarché.

Les organisations commerciales se trouvèrent alors dans une situation analogue à celle des scientifiques du XVIII$^{\text{ème}}$ siècle : leurs expéditions (les centres de vente) leurs rapportaient constamment des nouveaux spécimens (des données transactionnelles). Toutes ces données décrivent un "monde" miniature : pour une entreprise, le comprendre, c'est augmenter son pouvoir économique, donc ses chances de survie. Ici les SGBD sont insuffisants : leurs capacités d'interrogation se limitent à retrouver les transactions vérifiant certaines contraintes prédéfinies, comme par exemple tous les tickets de caisse enregistrés à la même minute, ou encore tous ceux contenant un produit donné. Pour découvrir des informations plus intéressantes, les mêmes problèmes qu'au XVIII$^{\text{ème}}$ siècle se posèrent : quels sont les points communs entre les transactions ? Peut-on les classer ? Quelles lois cachées peut-on découvrir dans la multitude des données ? Le problème étant que les quantités de données ne sont plus de l'ordre de la centaine ou du millier, mais de l'ordre du million voire du milliard. Un traitement humain est de ce fait presque impossible : il faut trouver un moyen *automatique* d'analyser les données, grâce à des ordinateurs.

La *fouille de données* était née. Le but de ce domaine est de trouver automatiquement, dans de grandes collections de données informatiques, des lois cachées, des similarités, des motifs fréquents, des classes : autant d'informations permettant de mieux comprendre et exploiter les données.

Prenons un exemple : revenons aux tickets de caisses. Imaginons en des millions, représentant toutes les ventes des supermarchés français pour un mois. Une tâche qui peut s'avérer utile est la recherche d'*itemsets fréquents*, c'est-à-dire les ensembles de produits qui sont vendus au cours de la même transaction dans plus de ε % des transactions (ε est un *seuil de fréquence* déterminé à l'avance). Un itemset fréquent est caractérisé par son *support*, c'est-à-dire le pourcentage des transactions où il apparaît. Un itemset fréquent possible est : {lait, céréales}, car beaucoup de clients mangent des céréales avec du lait, et ont donc besoin simultanément de ces deux produits. À partir de cette information, le directeur d'une chaîne de supermarché peut prendre des décisions stratégiques : va-t-il rapprocher les rayons du lait et des céréales, pour faciliter la vie du client ? Ou alors va-t-il les éloigner, pour que le client puisse avoir sur sa route l'occasion d'acheter d'autres produits ?

D'un point de vue algorithmique, découvrir des itemsets fréquents dans une base de données revient à chercher des solutions (ici les itemsets *fréquents*) dans un *espace de recherche* (ici l'ensemble de tous les itemsets possibles). On peut illustrer la difficulté du problème en s'aidant de l'exemple des tickets de caisse. Les *items* sont les différents produits proposés par le supermarché. Les itemsets sont des ensembles de n produits, où n est compris entre 1 et N, le nombre de produits dans le catalogue du supermarché. Le nombre d'itemsets possible est 2^N. Ainsi même si le catalogue ne contient qu'une centaine de produits, le nombre d'ensembles de produits est 2^{100}. Et un catalogue réel contient en général des milliers de produits. L'espace de recherche considéré est donc très grand...

Une méthode classique d'exploration d'un espace de recherche en Intelligence Artificielle s'appelle *"Générer et Tester"*. Cette méthode repose sur deux modules : un *générateur* qui énumère toutes les solutions candidates possibles, et un *testeur* qui ne garde que les solutions satisfaisant des critères définis à l'avance. Pour la recherche d'itemsets fréquents, le générateur crée tous les itemsets possibles, et le testeur ne garde que les itemsets satisfaisant le critère de fréquence (voir algorithme 1).

Le défaut de cette méthode est qu'elle génère tous les itemsets existants, et est donc peu efficace. L'algorithme le plus connu pour découvrir des itemsets fréquents est APRIORI [Agrawal et Srikant 1994]. Cet algorithme est une amélioration de la méthode "Générer et Tester", qui réduit le nombre d'itemsets générés. Pour ce faire, la génération des itemsets est organisée par niveaux, c'est-à-dire que l'algorithme commence par générer les itemsets de taille 1, puis les itemsets de taille 2 et ainsi de suite. La réduction du nombre d'itemsets générés à chaque niveau vient d'une propriété d'*anti-monotonie* : tous les sous-ensembles d'un itemset fréquent sont fréquents, et tous les sur-ensembles d'un itemset infréquent sont infréquents. Donc pour générer les itemsets candidats de longueur $k+1$, il suffit de

---

Algorithme 1 Générer et Tester

$L \leftarrow \emptyset$ // *ensemble des itemset fréquents*
Tant que il reste des itemsets à générer **Faire**
  $I \leftarrow$ Générer un nouvel itemset
  Si $frequence(I) \geq \varepsilon$ **Alors**
    $L \leftarrow L \cup I$
  Fin Si
Fin Tant que
Retourner L

---

générer des sur-ensembles des itemsets fréquents de longueur k. Cette approche permet de réduire le nombre d'itemsets candidats générés, tout en restant une approche *complète* : à la fin de l'exécution de l'algorithme **tous** les itemsets fréquents sont trouvés. Le schéma simplifié de l'algorithme Apriori est donné dans l'algorithme 2.

Algorithme 2 Apriori. ($GenereCandidats$ et $TesteCandidats$ seront détaillés plus tard)

$L_1 \leftarrow$ tous les itemsets fréquents à 1 élément
$k \leftarrow 1$
Tant que $L_k \neq \emptyset$ **Faire**
  $C_{k+1} \leftarrow GenereCandidats(L_k)$
  $L_{k+1} \leftarrow TesteCandidats(C_{k+1})$
  $k \leftarrow k + 1$
Fin Tant que
Retourner $\bigcup_k L_k$

Si l'algorithme Apriori est un progrès par rapport à l'algorithme "Générer et Tester", il génère néanmoins beaucoup d'itemsets candidats, pour lesquels $TesteCandidats$ doit faire de coûteux accès à la base de données.

Limiter le nombre d'itemsets candidats infréquents générés est la meilleure méthode pour améliorer Apriori. De très nombreuses recherches ont été conduites dans ce sens, permettant la création d'algorithmes beaucoup plus efficaces que l'algorithme Apriori initial ([Toivonen 1996], [Brin et al. 1997], [Pasquier et al. 1999] entre autres).

2 Des données de plus en plus complexes

De nos jours, les données stockées sur support informatique sont non seulement de plus en plus nombreuses, mais aussi de plus en plus complexes. Les experts en génomique, chimie, télécommunications par exemple ont rassemblé de larges corpus de données spécifiques à leur domaine. Or ces données ne sont pas de simples données transactionnelles : ce sont des données structurées, ayant la forme de séquences (ADN), d'arbres (XML) ou de graphes (molécules chimiques).

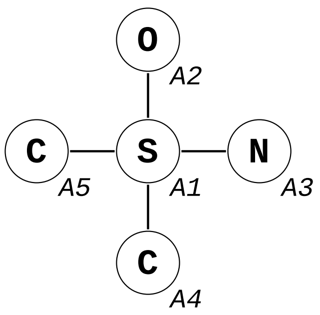
La découverte de motifs fréquents dans des données structurées est un enjeu capital. Il s'agit de découvrir des motifs fréquents qui sont des sous-structures de séquences, d'arbres ou de graphes. Les modèles de données et algorithmes conçus pour traiter des données transactionnelles ne sont plus adaptés.

Deux approches sont possibles :

– une approche générique, qui se fonde sur l'utilisation d'un langage suffisamment expressif pour qu'on puisse représenter dans ce langage toutes sortes de données structurées. Un algorithme général peut alors être conçu pour calculer les motifs fréquents dans un ensemble de formules de ce langage.
– une approche spécifique, qui, au contraire, est conçue pour un type de structure donné (séquence, arbre ou graphe). Une telle approche exploite les particularités des structures auxquelles elle s'applique, ainsi que de la structure des motifs recherchés, pour optimiser la découverte de motifs fréquents.

2.1 Approche générique

Cette approche est illustrée par le système Warmr ([Dehaspe 1998], [Dehaspe et Toivonen 1999]). Warmr est une généralisation de l'approche Apriori s'appliquant à des formules d'un fragment de la logique du premier ordre. Il est capable de trouver des motifs fréquents dans des données relationnelles. Or de nombreuses données structurées s'expriment dans ce formalisme de façon naturelle. Un exemple avec une molécule chimique (graphe) est donné dans la figure I.1. En effet, Warmr a fait ses preuves dans l'analyse de molécules chimiques. Il est également applicable à l'exploration de données structurées en arbres et en séquences.




```
atome(A1, soufre)
^ atome(A2, oxygene)
^ atome(A3, azote)
^ atome(A4, carbone)
^ atome(A5, carbone)
^ liaison(A1, A2)
^ liaison(A1, A3)
^ liaison(A1, A4)
^ liaison(A1, A5)
```

Fig. I.1 – Exemple de reformulation d'un graphe en formule logique

La figure I.2 montre un exemple de découverte de motif fréquent par Warmr. Pour aider le lecteur, les molécules ont été dessinées, mais Warmr ne travaille qu'à partir des formules logiques.

Pour arriver à ce résultat, Warmr a dû effectuer le processus "Générer et Tester" pour des formules logiques. Cela consiste, tout d'abord, à chercher toutes les formules logiques à un prédicat fréquentes sur les données. Ici les prédicats sont `atome` et `liaison`. Par exemple, la formule `atome(X, soufre)` est une formule à un prédicat. Cette formule est fréquente si elle généralise au moins ε formules données en entrée. Dans des données transactionnelles le test de généralisation est un simple test d'inclusion entre ensembles. Mais pour des données exprimées par des formules logiques, ce test est un *test de généralisation entre formules logiques.*

En toute rigueur, ce test de généralisation devrait être fondé sur l'implication logique. Cependant l'inconvénient de l'implication logique est d'être non décidable dans le cas général [Lloyd 1986]. Pour cette raison, Warmr utilise le test de θ-subsomption [Plotkin 1970], test correct mais qui n'est complet que dans certains fragments de la logique du premier ordre. Le test de θ-subsomption a l'avantage d'être décidable. Toutefois, la θ-subsomption d'un exemple par une formule est une opération de complexité exponentielle en la taille de la formule candidate.

Le test de θ-subsomption revient à trouver des substitutions des variables de la formule à tester avec les constantes et variables des données. Dans notre exemple il est possible de faire les substitutions `X/A1` dans la première molécule et `X/B1` dans la deuxième, donc la formule `atome(X,`

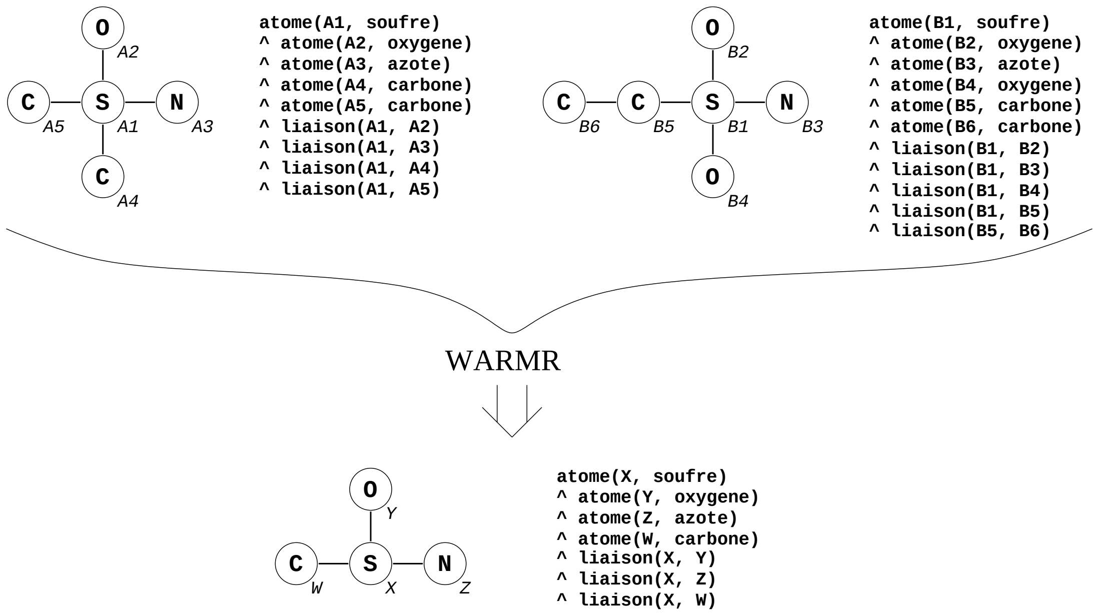


FIG. I.2 – Exemple de découverte de motif fréquent par WARMR, $\varepsilon = 2$

`soufre)` est fréquente. Ensuite, il faut générer toutes les formules logiques à deux prédicats possibles. Le nombre de possibilités est très élevé : montrons par exemple quelques manières de générer des formules à deux prédicats étendant le motif fréquent `atome(X, soufre)` :

- `atome(X, soufre)` $\wedge$ `atome(X, oxygene)`
- `atome(X, soufre)` $\wedge$ `atome(X, carbone)`
- `atome(X, soufre)` $\wedge$ `atome(X, azote)`
- `atome(X, soufre)` $\wedge$ `atome(Y, oxygene)`
- `atome(X, soufre)` $\wedge$ `atome(Y, carbone)`
- `atome(X, soufre)` $\wedge$ `atome(Y, azote)`
- `atome(X, soufre)` $\wedge$ `liaison(X, X)`
- `atome(X, soufre)` $\wedge$ `liaison(X, Y)`
- `atome(X, soufre)` $\wedge$ `liaison(Y, X)`
- `atome(X, soufre)` $\wedge$ `liaison(Y, Z)`

Comme on peut le voir, le nombre de manières de générer de nouvelles formules à partir d'une formule initiale est très élevé. Car pour chaque prédicat rajouté, il y a de nombreuses manières d'arranger les termes (ici, les variables et constantes) arguments d'un même prédicat. Ceci illustre le problème de la génération de motifs candidats ayant la forme de formules logiques : il existe de très nombreux candidats à chaque niveau, l'espace de recherche est donc gigantesque. Certaines optimisations de WARMR résolvent partiellement ce problème.

En résumé, WARM présente deux désavantages : l'espace de recherche est gigantesque à cause de la puissance du formalisme logique utilisé, ce qui conduit à générer beaucoup de candidats. Et le test de fréquence de chacun de ces candidats, reposant sur un test de θ-subsomption, est très coûteux.

Ces deux désavantages se payent par une perte d'efficacité, et limitent la taille des motifs fréquents qu'il est possible de découvrir.

Or des restrictions arbitraires sur la taille des motifs fréquents à découvrir ne sont pas toujours acceptables. C'est pourquoi les approches spécifiques ont vu le jour.

2.2 Approches spécifiques

Nous allons présenter brièvement des approches spécifiques s'appliquant à des graphes et à des arbres.

2.2.1 Approches s'appliquant à des graphes

Le besoin de méthodes permettant de découvrir des motifs fréquents qui soient des graphes a surtout été motivé par l'analyse de bases de données de molécules chimiques. C'est sur ce type de données qu'a été validée l'approche de [Inokuchi et al. 2000].

Dans leur algorithme *Apriori Graph Mining (AGM)*, [Inokuchi et al. 2000] considèrent la recherche de motifs fréquents dans des *graphes*. La recherche de motifs fréquents effectuée par AGM consiste à trouver tous les sous-graphes qui se répètent au moins ε fois dans l'ensemble des graphes donnés en entrée. Un exemple est donné par la figure I.3. L'intérêt de l'approche retenue dans AGM est de présenter un codage adéquat des graphes en matrices d'adjacence qui permet d'adapter l'approche Apriori à la recherche de motifs fréquents de type graphes.

AGM a été testé sur le même corpus de molécules chimiques que Warmr. Les motifs fréquents les plus gros trouvés par AGM avaient une taille de 13 atomes, alors que Warmr n'avait pas pu dépasser des motifs de 3 atomes.

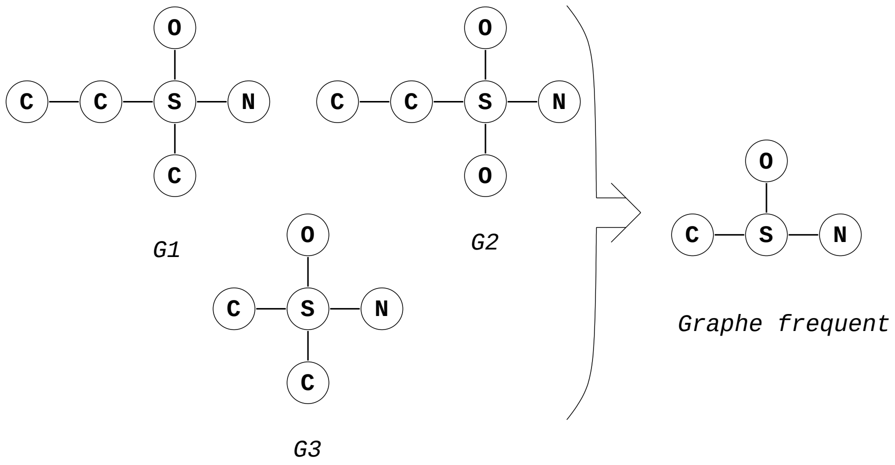


Fig. I.3 – Graphe fréquent le plus gros trouvé par AGM à partir de G_1, G_2 et G_3 ($\varepsilon = 3$)

2.2.2 Approches s'appliquant à des arbres

La motivation à l'origine de la recherche de motifs fréquents dans des arbres vient surtout de l'analyse de documents électroniques. Ainsi, les approches présentées dans [Asai et al. 2002] et [Zaki 2002] ont pour données d'entrée des traces de navigation dans des sites web, ou encore des documents informatiques mis sous forme d'arbre (pages HTML). Les documents informatiques structurés se prêtent particulièrement bien à la mise sous forme arborescente. C'est notamment le cas des documents XML, qui ont été conçus pour stocker l'information sous une forme structurée : le texte est encadré par des balises qui indiquent ce à quoi il correspond, et ces balises peuvent s'imbriquer sur plusieurs niveaux. La figure I.4 montre un exemple de document XML et l'arbre des balises associé.

Pour la suite, nous donnerons des exemples utilisant principalement ces documents, car ils sont au cœur de notre étude.

Notons que l'arbre des balises est un arbre *étiqueté ordonné*. En effet chaque nœud a comme étiquette une balise du document XML. De plus, l'ordre séquentiel des balises dans le document XML se traduit par l'ordre des nœuds-fils d'un nœud donné dans l'arbre correspondant. Il est donc possible de retrouver l'ordre des balises du document XML original à partir d'un parcours de l'arbre des balises associé.

```
<?xml version="1.0" encoding="ISO-8859-1">
<document>
   <livre>
      <titre> "Guerre et Paix" <\titre>
      <auteur> "Leon Tolstoi" <\auteur>
   <\livre>
<\document>
```

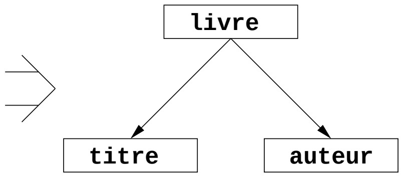


FIG. I.4 – Un document XML simple et l'arbre des balises associé

Les deux méthodes que nous allons présenter ont en commun de rechercher tous les arbres inclus fréquemment (c'est-à-dire plus de ε fois) dans un ensemble d'arbres donné en entrée. C'est sur la définition de l'inclusion d'un arbre dans un autre que [Asai et al. 2002] et [Zaki 2002] diffèrent. Pour [Asai et al. 2002], les motifs fréquents sont des arbres dont on retrouve la copie conforme dans les arbres des données, comme sur la figure I.5. Par contre, [Zaki 2002] est plus permissif : il considère qu'un arbre est inclus dans un des arbres des données même si l'imbrication des nœuds n'est pas exactement la même, comme c'est le cas pour l'arbre A_2 sur la figure I.6. Ceci permet de retrouver des motifs fréquents malgré une certaine hétérogénéité dans la structure des données. Toutefois, [Asai et al. 2002] et [Zaki 2002] se retrouvent sur un point : les fils/descendants d'un nœud donné doivent toujours se trouver dans le même ordre, pour faire partie d'un arbre fréquent.

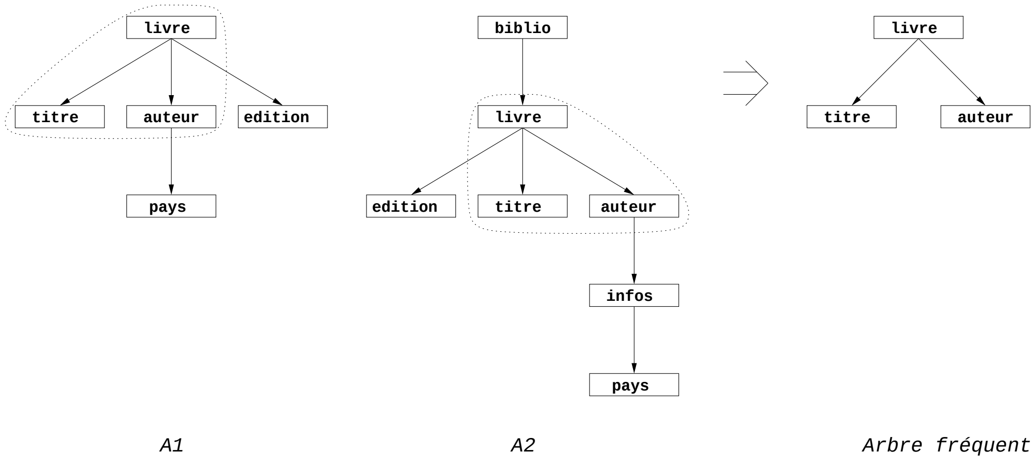


FIG. I.5 – Arbre fréquent le plus gros trouvé par Find-Freq-Trees [Asai et al. 2002] à partir de A_1 et A_2 ($\varepsilon = 2$)

Les algorithmes de [Asai et al. 2002] et [Zaki 2002] sont des adaptations de l'approche APRIORI pour un espace de recherche composé d'arbres. Conserver l'ordre des fils/descendants autorise l'utilisation de certaines représentations et optimisations. Ainsi, [Zaki 2002] représente les arbres par des chaînes de caractères, ce qui lui permet de bénéficier des traitements efficaces existant sur ces

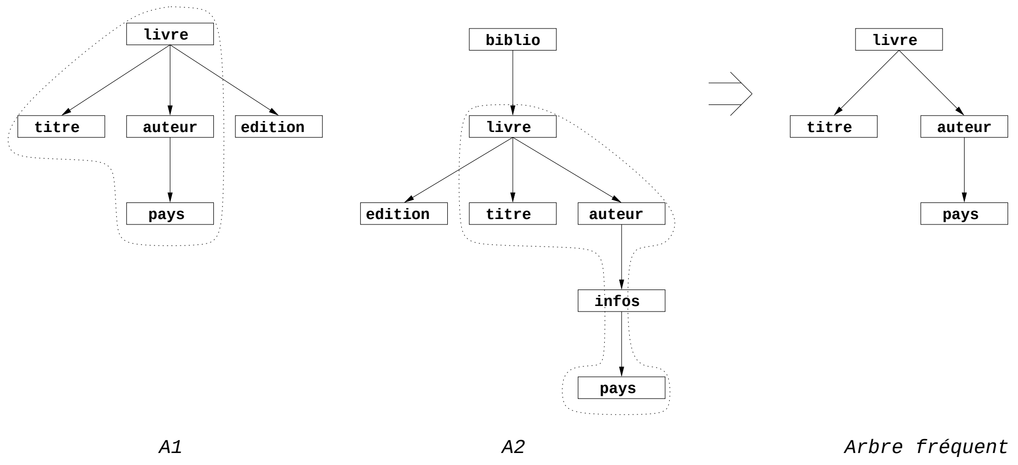


FIG. I.6 – Arbre fréquent le plus gros trouvé par TreeMiner [Zaki 2002] à partir de A_1 et A_2 ($\varepsilon = 2$)

structures de données.

En observant les figures I.5 et I.6, on remarque que le nœud étiqueté par *edition* est laissé de côté par les algorithmes de [Asai et al. 2002] et [Zaki 2002]. Cela vient justement du fait que ces algorithmes prennent l'ordre en compte : la position du nœud étiqueté par *edition* varie dans les documents, contrairement à celle du nœud étiqueté par *auteur* qui est toujours un frère droit du nœud étiqueté par *titre*.

Pourtant, les deux nœuds étiquetés par *livre* ont un fils étiqueté par *edition* : l'information contenue dans ce nœud apparaît fréquemment, et elle est peut-être intéressante.

La question est : doit-on s'inquiéter de perdre cette information ? Tout dépend des données étudiées. [Asai et al. 2002] a effectué des tests sur des données produites par des moteurs de recherche, où l'ordre était important. De même, l'évaluation de [Zaki 2002] porte sur des traces d'accès à un site web, et là encore l'ordre de l'accès aux pages par les utilisateurs est important. Pour ces deux approches, perdre une information parce qu'elle est mal ordonnée n'est pas un problème, car l'ordre des fils/descendants est une information plus importante.

Ce n'est pas le cas dans de nombreuses applications.

Considérons par exemple une base de données de documents XML hétérogènes, récupérés sur Internet. "Hétérogènes" signifie que les documents que nous considérons viennent de sources différentes, et ne suivent pas nécessairement les mêmes règles d'écriture. L'imbrication des balises pourra donc être différente, comme c'est le cas dans les exemples pour le nœud étiqueté par *pays*. L'ordre des fils/descendants pourra aussi être changé, comme c'est le cas dans le nœud étiqueté par *edition*. Dans ce cadre l'ordre n'est pas une information importante : ce qui importe est de savoir qu'un *livre* a un *titre*, un *auteur* et une *edition*, et que de plus l'auteur a un *pays*.

Il existe donc de nombreuses applications où trouver des motifs fréquents dans des arbres en étant flexible sur l'ordre est indispensable.

3 Contributions de cette thèse

Les travaux réalisés dans le cadre de cette thèse portent sur la recherche d'arbres fréquents dans des arbres ordonnés étiquetés. Ils généralisent les travaux précédents ([Asai et al. 2002], [Zaki 2002]) en permettant une plus grande flexibilité dans les motifs fréquents recherchés.

Les algorithmes que nous avons conçus trouvent des arbres fréquents dans des *arbres étiquetés* *ordonnés*, sans tenir compte de l'ordre des frères, et en autorisant au besoin des variations dans l'imbrication des balises, comme [Zaki 2002]. La figure I.7 illustre l'arbre fréquent trouvé par nos algorithmes sur les arbres A_1 et A_2 des exemples précédents. Cela montre que le type des motifs que nous recherchons est adaptée à la découverte de motifs fréquents dans des corpus de documents XML.

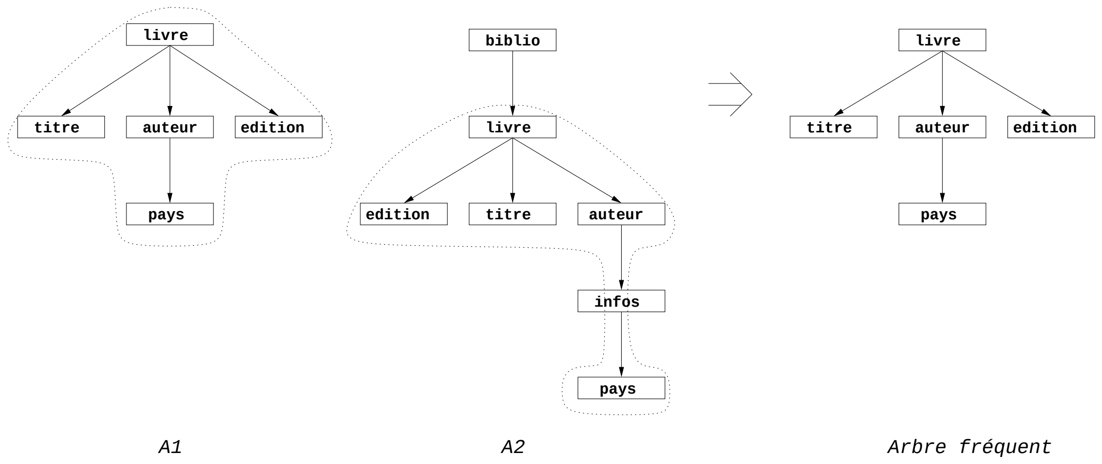


FIG. I.7 – Arbre fréquent le plus gros trouvé par nos algorithmes à partir de A_1 et A_2 ($\varepsilon = 2$)

Dans le cas de documents XML hétérogènes, plusieurs balises se référant à la même chose peuvent être employées. Par exemple, pour désigner une voiture, on peut trouver des balises telles que : *véhicule, voiture, automobile, auto* (synonymes). Pour que notre algorithme de recherche d'arbres fréquents soit le plus efficace possible sur des données hétérogènes, idéalement il faudrait que ces problèmes dit de *synonymie* et de *polysémie* soient résolus. Résoudre ces problèmes est un travail de recherche en soi, et nous avons décidé de ne pas nous focaliser sur cette tâche. Pour quelques pistes de solutions, le lecteur interessé peut se référer à nos travaux préliminaires sur le sujet : [Termier et al. 2001].

Pour la suite de ce mémoire, nous supposerons que nous disposons d'un préprocesseur qui unifie les balises se référant au même concept, et que nous n'avons donc pas de problème ni de synonymie, ni de polysémie. La chaîne de traitement est illustrée sur la figure I.8.

Les motifs fréquents arborescents que nous considérons sont plus généraux que ceux considérés dans les approches existant auparavant. Cela se paye par une plus grande difficulté algorithmique : ainsi Kilpeläinen a montré dans [Kilpeläinen 1992] que si le problème de tester l'inclusion d'un arbre dans un autre était polynomial dans le cas ordonné, il est NP-complet dans le cas non-ordonné qui nous intéresse.

Afin d'être efficace et flexible malgré cette plus grande difficulté, nous avons exploré deux approches différentes : la première a donné lieu au développement de l'algorithme TREEFINDER ([Termier et al. 2002]), la seconde a abouti à la création des algorithmes DRYAL et DRYADE.

TREEFINDER : approximation de l'ensemble des arbres fréquents

De part la généralité des motifs recherchés, nos algorithmes sont susceptibles d'êtres confrontés à des espaces de recherche gigantesques. Si les données sont trop nombreuses ou présentent des motifs fréquents trop complexes, cet espace risque de devenir si important que trouver tous les motifs fréquents en un temps raisonnable deviendra impossible. Dans ces cas, une approche *complète*, c'est-

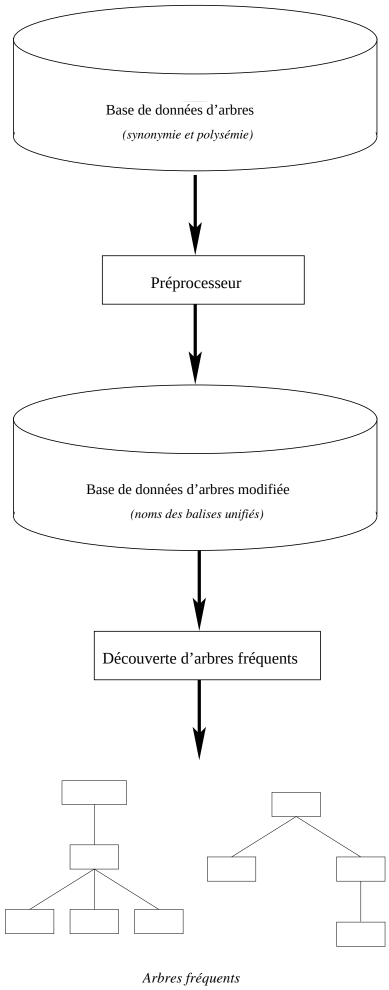


Fig. I.8 – La chaîne de traitement pour extraire des arbres fréquents à partir d'une collection de documents XML hétérogène à-dire qui trouve tous les motifs fréquents, peut être trop coûteuse. Nous avons donc développé un algorithme *approché*, TREEFINDER, qui ne trouve qu'une partie de tous les motifs fréquents présents dans les données. Pour cet algorithme, nous fournissons une étude qualitative et quantitative des motifs fréquents trouvés, mais aussi de ceux qui ne sont pas trouvés. Cela permet d'évaluer la qualité de l'approximation effectuée.

DRYAL et DRYADE : une nouvelle approche complète de recherche d'arbres fréquents

Les algorithmes DRYAL et DRYADE exploitent les spécificités des structures d'arbres pour être efficaces, et sont fondés sur les deux principes suivants.

Approche constructive :

Notre approche est une approche par niveaux, qui commence par créer les arbres fréquents de profondeur 1, puis les combine entre eux pour former les arbres fréquents de profondeur 2, et continue jusqu'à l'obtention de tous les arbres fréquents. Nous nous inspirons fortement des récentes techniques *verticales* (voir chapitre III), qui sont en général plus efficaces que les approches de type APRIORI classiques (dites *horizontales*). Ce sont des approches qui se révèlent plus adaptées à la découverte de motifs fréquents complexes.

Diviser pour reformuler :

Une technique couramment utilisée en Informatique pour résoudre un problème complexe s'appelle *"diviser pour règner"*. On découpe le problème en sous-problèmes plus simples que l'on sait résoudre, et à partir des solutions des sous-problèmes on reconstruit la solution du problème global. Nous nous inspirons de cette technique pour résoudre notre problème de telle sorte que les sous-problèmes à résoudre se ramènent, après une *reformulation*, à des recherches d'**itemsets fréquents dans des données transactionnelles**. Grâce à ces reformulations, notre approche bénéficie de la somme considérable de travaux effectués sur la recherche d'itemsets fréquents dans des données transactionnelles. La moindre amélioration sur ces méthodes est directement exploitable pour optimiser notre approche. Nous présentons aussi dans ce mémoire la notion d'*arbre fréquent fermé*, qui à notre connaissance n'a jamais été utilisée pour la recherche d'arbres fréquents, et qui permet un gain très important d'efficacité.

Restriction

Toujours pour pouvoir affronter un espace de recherche gigantesque, nous imposerons dans nos algorithmes DRYAL et DRYADE une restriction sur les arbres fréquents que nous pouvons découvrir : ce sont des arbres où un nœud ne peut pas avoir deux fils de même étiquette. Pour les distinguer, nous appelerons de tels arbres fréquents des *patterns fréquents*. Il a été montré dans [Delobel et al. 2003] que ces patterns correspondent à des patterns de requêtes XML présentant un bon compromis entre efficacité et expressivité.

La restriction utilisée nous permet d'éviter une trop grande explosion combinatoire. Nous verrons dans le chapitre V à quels moments des algorithmes la restriction intervient, et ce qu'elle permet de gagner en complexité.

La figure I.9 illustre le type d'arbres fréquents que notre restriction nous empêche de trouver.

L'arbre fréquent *a*) correspond à un journal avec à la fois un article ayant une photo et du texte et un article ayant un dessin et du texte. Avec la restriction que nous imposons, nous ne trouvons pas l'arbre *a*), mais nous trouvons les deux arbres fréquents de *b*). L'arbre fréquent *a*) est donc plus précis, et correspond à une analyse de documents XML avec un grain très fin : on peut différencier plusieurs types de journaux par exemple. Les deux arbres fréquents de *b*), correspondant à un grain

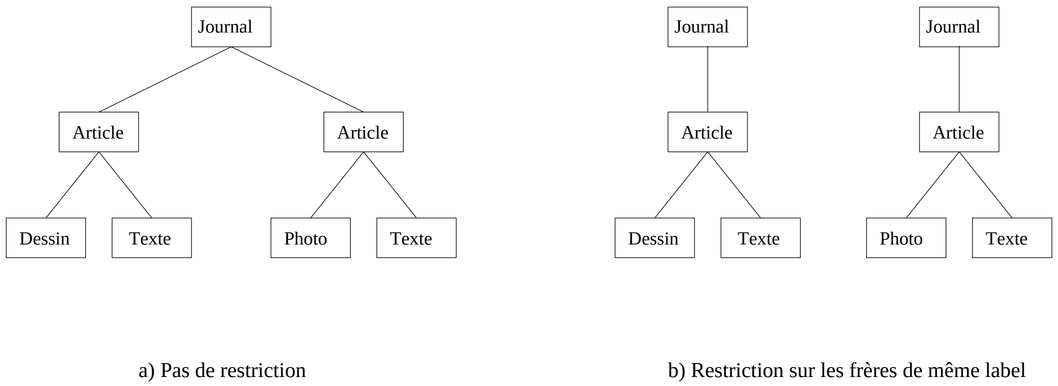


Fig. I.9 – Arbres fréquents découverts : a) sans restriction b) avec la restriction sur les étiquettes de frères que nous imposons

moins fin, représentant une petite perte d'information par rapport à *a*). Mais ils sont beaucoup moins coûteux à obtenir, ce qui peut s'avérer crucial pour certaines applications.

4 Organisation du mémoire

Pour conclure cette introduction, nous donnons le plan de ce mémoire :

- Le **Chapitre II** présente de manière formelle le problème auquel nous nous intéressons.
- Le **Chapitre III** contient un état de l'art sur la recherche de motifs fréquents. Les principales approches pour les données transactionnelles sont présentées en premier, puis plusieurs approches sur des données de type arbre sont ensuite exposées. Le lecteur y retrouvera les trois algorithmes brièvement abordés dans le présent chapitre.
- Le **Chapitre IV** est consacré à notre algorithme approché TreeFinder. L'algorithme est présenté en détail et une étude expérimentale précise est décrite. De plus, la perte de complétude est analysée, de manière qualitative et quantitative.
- Le **Chapitre V** concerne nos algorithme Dryal et Dryade, qui sont des algorithmes corrects et complets de recherche d'arbres fréquents. Les algorithmes sont présentés en détail, leur correction et complétude sont prouvées.
- Le **Chapitre VI** présente une étude expérimentale sur les performances de Dryade. L'étude a été réalisée aussi bien sur des données artificielles que sur des données réelles, et Dryade a été comparé à Warmr.
- Enfin, le **Chapitre VII** conclue le mémoire et donne quelques perspectives.

Chapitre II

Recherche d'arbres fréquents : Position du problème

Nous allons dans ce chapitre définir formellement les données sur lesquelles nous travaillons, ainsi que le problème que nous traitons dans cette thèse.

1 Arbres étiquetés ordonnés : définitions

Le type de donnée que nous allons utiliser quasi-exclusivement pour le reste de ce mémoire est l'*arbre*. Nous allons donc définir dans cette partie un cadre formel adapté aux arbres, et ensuite à la découverte de motifs fréquents dans des arbres. Ce cadre formel est classique, et s'inspire largement de celui que l'on peut trouver dans la thèse de Pekka Kilpeläinen [Kilpeläinen 1992].

Définition 1 (Relation binaire) *Soit D un ensemble quelconque. Une* relation binaire *R sur D est un sous-ensemble du produit cartésien $D \times D$.*

Définition 2 (Fermeture transitive) *La* fermeture transitive *de R, notée R^+, est définie comme suit :*

$$
\begin{array}{rcl}
R^1 & = & \{(x, y) \mid (x, y) \in D^2\} \\
R^{n+1} & = & \{(x, y) \mid \exists z \in D \ tq \ (x, z) \in R \ et \ (z, y) \in R^n\}, \ \ n > 0 \\
R^+ & = & \bigcup_{n>0} R^n
\end{array}
$$

Définition 3 (Arbre) *Un arbre est défini par un triplet $T = (N_T, A_T, racine(T))$ où N_T est un ensemble de nœuds, $A_T \subseteq N_T^2$ est une relation binaire sur N_T, et $racine(T) \in N_T$ est un nœud particulier appelé la **racine**. Pour un couple $(u, v) \in A_T$, on dira que (u, v) est un* arc *de l'arbre, et que u est le* parent *de v, ce qui sera noté $u = parent(v)$. A_T doit vérifier les conditions suivantes :*

- *$racine(T)$ n'a pas de parent.*
- *Tout nœud de l'arbre (sauf la racine) a exactement un parent.*
- *À partir de la racine, on peut atteindre n'importe quel nœud de l'arbre en suivant des arcs de A_T, i.e. pour tout nœud $v \in N_T$ différent de la racine, $(racine(T), v) \in A_T^+$.*

Notations : Soit T un arbre. Soit $u \in N_T$ un nœud de T. Tous les nœuds de T dont u est le parent seront appelés les *fils* de u :

$$fils(u) = \{v \in N_T \mid (u, v) \in A_T\}$$

On définit les *descendants* de u par :

$$descendants(u) = \{v \in N_T \mid (u, v) \in A_T^+\}$$

et les *ancêtres* de u par :

$$ancetres(u) = \{v \in N_T \mid (v, u) \in A_T^+\}$$

Une séquence d'arcs de A_T $(u_1, u_2), ..., (u_{n-1}, u_n)$ entre u_1 et u_n est appelée un *chemin* entre u_1 et u_n. La longueur de ce chemin est $n - 1$.

Les nœuds de N_T qui n'ont pas de fils sont appelés des *feuilles*. Les autres nœuds de N_T sont appelés *nœuds internes*.

On appelera *profondeur* de l'arbre T, notée $profondeur(T)$, la longeur du plus grand chemin de la racine de T à une des feuilles T, augmentée de 1.

Définition 4 (Sous-arbre) *Soit T un arbre. Soit $u \in N_T$ un nœud de T. Le sous-arbre de T de racine u, noté $T[u]$, est l'arbre $T[u] = (N_{T[u]}, A_{T[u]}, u)$ tel que :*
- $N_{T[u]} = \{u\} \cup descendants(u)$
- $A_{T[u]} = A_T \cap (N_{T[u]} \times N_{T[u]})$.

Définition 5 (Arbre étiqueté) *Soit E un ensemble d'étiquettes. Un arbre T étiqueté par E est défini par $(N, A, racine(T), \varphi)$ tel que :*
- *$(N, A, racine(T))$ est un arbre.*
- *$\varphi : N \mapsto E$ est une application qui associe à chaque nœud de N une étiquette de E.*

Définition 6 (Arbre ordonné) *Un arbre ordonné T est défini par $(N_T, A_T, racine(T), \preceq)$ tel que :*
- *$(N_T, A_T, racine(T))$ est un arbre.*
- *$\prec \subseteq N_T^2$ est la relation* frère, *c'est une relation binaire définissant un ordre partiel sur N_T^2 et un ordre total sur les fils d'un même nœud de N_T.*
  Ainsi pour un nœud interne u, ses fils $u_1, ..., u_n$ $(n \geq 0)$ sont ordonnés de gauche à droite tels que $u_1 \prec u_2 \prec ... \prec u_n$.
  En fonction de cet ordre, u_i est appelé le i-ème fils de u.

Notations : Soit T un arbre ordonné, et soit u un nœud interne de T ayant n fils $u_1, ..., u_n$. Les fils sont ordonnés comme suit : $u_1 \prec u_2 \prec ... \prec u_n$. Alors pour tout $i \in [1, n]$ on notera
- $\forall j < i$ u_j est un *frère gauche* de u_i $(i > 1)$.
- u_{i-1} est le *frère gauche immédiat* de u_i $(i > 1)$.
- $\forall j > i$ u_j est un *frère droit* de u_i $(i < n)$.
- u_{i+1} est le *frère droit immédiat* de u_i $(i < n)$.

Pour la suite de ce mémoire, tous les arbres considérés seront, sauf mention du contraire, des arbres étiquetés ordonnés. Par abus de langage et dans un souci de simplicité, on utilisera alors le terme **arbre** pour désigner ces arbres étiquetés ordonnés.

Identification des nœuds : Les nœuds d'un arbre peuvent être identifiés en leur affectant des numéros, chaque nœud ayant un numéro unique déterminé par sa position dans l'arbre.

Les deux principales techniques pour numéroter les nœuds d'un arbre sont *l'ordre préfixe* et *l'ordre postfixe.*

Définition 7 (Ordre préfixe) *Soit T un arbre. À un nœud $u \in N_T$ on peut assigner un numéro dans l'ordre **préfixe** $pre(u)$ à partir des règles suivantes :*
- $pre(racine(T)) = 1$
- *Le premier fils de u a le numéro préfixe de u plus 1 : $pre(u_1) = pre(u) + 1$*
- *Soit v le frère gauche immédiat de u. Soit p le plus grand numéro préfixe assigné dans $T[v]$. Alors $pre(u) = p + 1$.*

Définition 8 (Ordre postfixe) *Soit T un arbre. À un nœud $u \in N_T$ on peut assigner un numéro dans l'ordre **postfixe** $post(u)$ à partir des règles suivantes :*
- *Soit u la feuille la plus à gauche de T. Alors $post(u) = 1$.*
- *Soit u un nœud interne de T, et soit p le plus grand numéro postfixe assigné à un descendant de u. Alors $post(u) = p + 1$.*
- *Soit v le frère droit immédiat de u. Alors le numéro postfixe de la feuille la plus à gauche de $T[v]$ est $post(u) + 1$.*

On pourra remarquer que l'ordre préfixe (resp. postfixe) correspond à numéroter les nœuds de l'arbre au cours d'un parcours préfixe (resp. postfixe).

2 Inclusion d'arbres

Il existe de très nombreuses manières de définir l'inclusion entre deux arbres T_1 et T_2. Ainsi, dans sa thèse [Kilpeläinen 1992], Kilpeläinen définit 10 relations d'inclusion d'arbre différentes.

Toutes ces définitions se basent sur l'existence d'une application que nous appelerons *mapping* des nœuds de T_1 vers les nœuds de T_2. Ce mapping préserve de manière forte ou faible les étiquettes des nœuds et la structure de l'arbre.

Les différences proviennent des propriétés de préservation qui sont considérées, ainsi que de l'injectivité ou non du mapping.

Nous englobons dans la définition suivante toutes les définitions d'inclusion dont nous nous servirons dans cette thèse. Nous illustrerons ensuite par des exemples les principales différences entres les différentes variantes de cette définition.

Définition 9 (Inclusion d'arbre (stricte)(exacte)(ordonnée)(faible)) *Soient T_1 et T_2 deux arbres étiquetés ordonnés, T_1 est inclus dans T_2, noté $T_1 \sqsubseteq T_2$, s'il existe un mapping $\mu : N_{T_1} \mapsto N_{T_2}$ tel que :*

1. *μ préserve les étiquettes, de façon stricte (1.a) ou selon un ordre $\leq_E$ sur les étiquettes (1.b)*

   1.a) préservation stricte : $\forall u \in N_{T_1}\ \ \varphi_{T_1}(u) = \varphi_{T_2}(\mu(u))$

   1.b) préservation selon un ordre : $\forall u \in N_{T_1}\ \ \varphi_{T_1}(u) \leq_E \varphi_{T_2}(\mu(u))$

   *Si la préservation est stricte (1.a)), on dira que l'inclusion est **stricte**.*

2. *μ préserve la relation de parenté (2.a) ou d'ancestralité (2.b).*
   2.a) préservation de la parenté : $\forall u, v \in N_{T_1}$ *si* $(u,v) \in A_{T_1}$ *alors* $(\mu(u), \mu(v)) \in A_{T_2}$
   2.b) préservation de l'ancestralité : $\forall u, v \in N_{T_1}$ *si* $(u,v) \in A_{T_1}$ *alors* $(\mu(u), \mu(v)) \in A^+_{T_2}$
   Si la relation de parenté est préservée (2.a)), on dira que l'inclusion est ***exacte****.*

Et de façon optionnelle :

3. *μ préserve l'ordre des fils :* $\forall u, v \in N_{T_1}$ *si* $u \prec_{T_1} v$ *alors* $\mu(u) \prec_{T_2} \mu(v)$*. Si l'ordre des fils est préservé on dira que l'inclusion est* ***ordonnée****.*
4. *μ est injective :* $\forall u, v \in N_{T_1}$ *si* $u \neq v$ *alors* $\mu(u) \neq \mu(v)$*. Si l'injectivité n'est pas vérifiée on dira que l'inclusion est* ***faible****.*

Cette définition générale nous permet de construire de nombreuses variantes. La définition d'inclusion la plus contrainte est l'**inclusion stricte exacte ordonnée**.

La figure II.1 donne l'exemple d'un arbre T_1 inclus dans un arbre T_2 par cette définition. Mais les arbres T_3 à T_7 des figures II.2 à II.6 ne sont pas inclus dans T_2 par cette définition.

Cependant, il existe une relation d'**inclusion stricte exacte** entre T_3 et T_2 (fig. II.2), et une relation d'**inclusion stricte ordonnée** entre T_4 et T_2 (fig. II.3).

Entre T_5 et T_2, il existe une relation d'**inclusion stricte** (fig. II.4).

Et entre T_6 et T_2, on peut trouver une relation d'**inclusion stricte faible** (fig. II.5).

Enfin, la figure II.6 illustre une définition d'**inclusion exacte ordonnée** utilisant un ordre des étiquettes $\leq_E$ tel que *in-folio* $\leq_E$ *livre*.

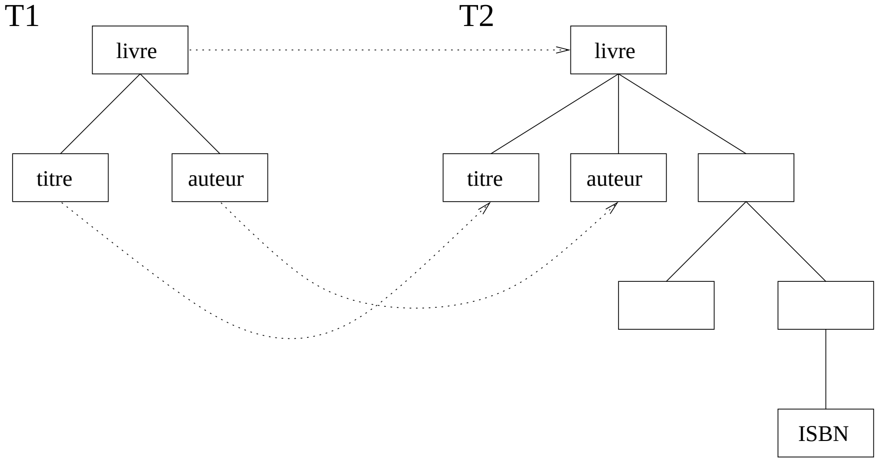


FIG. II.1 – Inclusion stricte exacte ordonnée

Notons que pour des raisons de lisibilité, nous n'avons pas fait apparaître les identifiants des noeuds dans les figures II.1 à II.6. Nous ne ferons apparaître ces identifiants dans les figures que quand ils sont vraiment nécéssaires.

Le problème du test de l'inclusion d'arbre que nous venons d'évoquer est un problème difficile en lui-même. Suivant la définition de l'inclusion choisie, le problème de déterminer l'inclusion d'un arbre dans un autre est un problème polynômial (inclusions ordonnées) ou NP-complet (inclusions non ordonnées). Une étude de complexité détaillée, ainsi que des algorithmes de test d'inclusion efficaces, se trouvent dans [Kilpeläinen 1992].

Notation : Pour une définition d'inclusion donnée $\sqsubseteq$, l'ensemble des mappings entre T_1 et T_2 sera noté $\mathcal{EM}_{\sqsubseteq}(T_1, T_2)$.

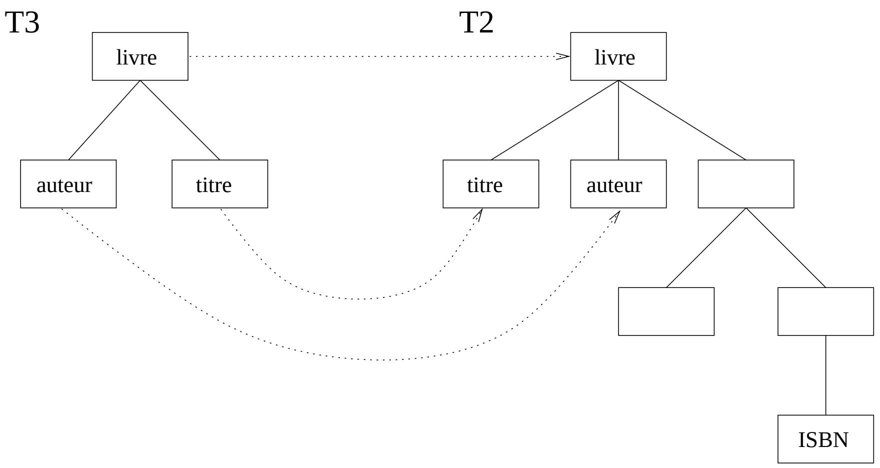


FIG. II.2 – Inclusion stricte exacte (non ordonnée)

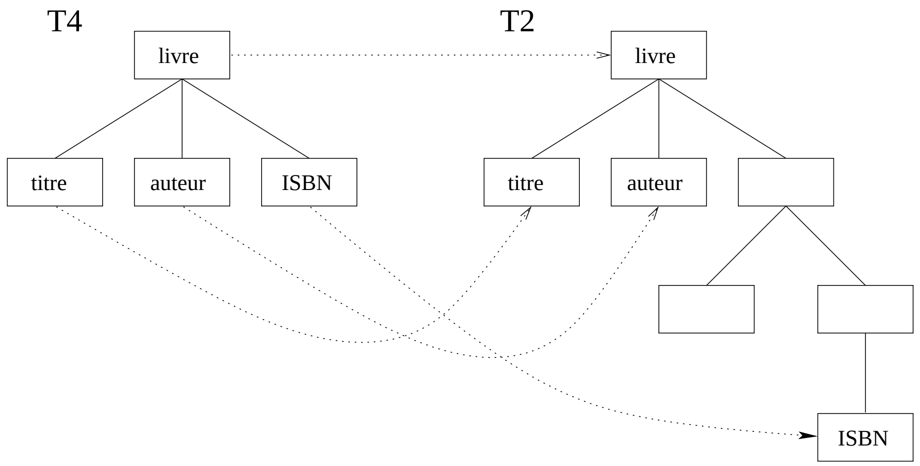


FIG. II.3 – Inclusion stricte ordonnée (non exacte)

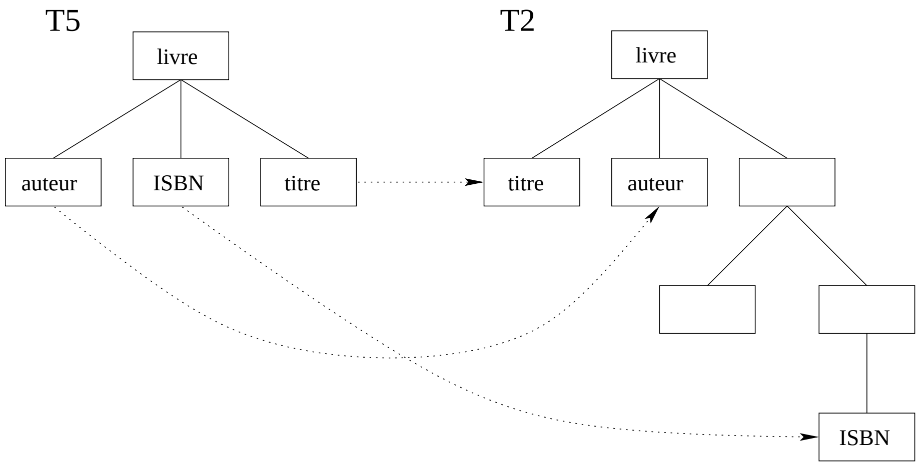


FIG. II.4 – Inclusion stricte (non ordonnée, non exacte)

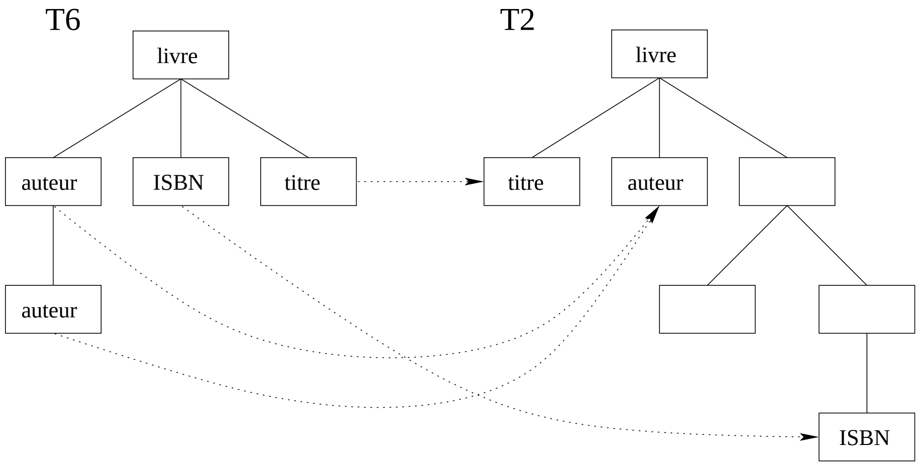


Fig. II.5 – Inclusion stricte faible (non ordonnée, non exacte)

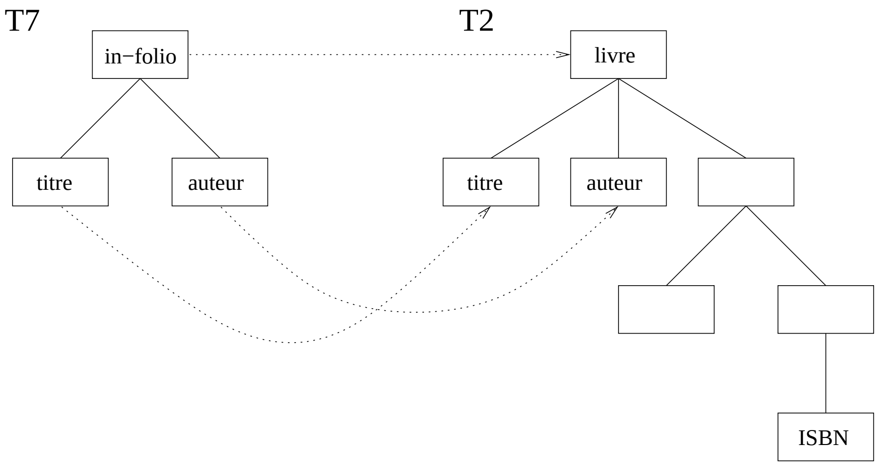


Fig. II.6 – Inclusion non stricte mais exacte et ordonnée

Le problème général de l'inclusion d'arbre est de trouver, étant donnés deux arbres T_1 et T_2, tous les sous-arbres de T_2 qui sont des occurrences de T_1. On dira alors que la racine d'un sous-arbre de T_2 qui est une occurrence de T_1 est une *occurrence de racine* de T_1.

Le problème général de l'inclusion d'arbre est plus difficile que le simple test d'inclusion entre deux arbres. En effet il faut cette fois trouver toutes les occurrences de l'arbre T_1 dans l'arbre T_2, et pas se contenter de prouver qu'il en existe une.

Nous nous intéressons à une restriction de ce problème, à savoir trouver tous les sous-arbres de T_2 qui sont des occurrences de T_1 **de racines différentes**.

Cela revient à trouver toutes les occurrences de racine de T_1 dans T_2.

Définition 10 (Occurrence de racine) *Soit $\sqsubseteq$ une définition d'inclusion entre arbres, et soient T_1 et T_2 deux arbres tels que $T_1 \sqsubseteq T_2$. Soit μ un mapping de T_1 vers T_2. On appelera* ***occurrence de la racine*** *de T_1 pour μ le nœud u de T_2 tel que $u = \mu(racine(T_1))$.*

L'ensemble des occurrences de racine de T_1 dans T_2 sera appelé $Locc_{\sqsubseteq}(T_1, T_2)$ et défini comme suit :

$$Locc_{\sqsubseteq}(T_1, T_2) = \{u \in N_{T_2} \mid u = \mu(racine(T_1)) \text{ avec } \mu \in \mathcal{EM}_{\sqsubseteq}(T_1, T_2)\}$$

Notons que pour une occurrence de racine u il peut y avoir plusieurs mappings μ satisfaisant les conditions précédemment énoncées. L'ensemble de ces mappings sera noté $\mathcal{EM}_u(T_1, T_2)$.

Propriété 1 *On a :*

$$\mathcal{EM}_{\sqsubseteq}(T_1, T_2) = \bigcup_{u \in Locc_{\sqsubseteq}(T_1, T_2)} \mathcal{EM}_u(T_1, T_2)$$

Preuve : Trivial car $Locc_{\sqsubseteq}(T_1, T_2)$ nous donne toutes les racines des mappings, pour chaque racine $u \in Locc_{\sqsubseteq}(T_1, T_2)$, $\mathcal{EM}_u(T_1, T_2)$ représente tous les mappings associés à cette racine. Donc l'union des $\mathcal{EM}_u(T_1, T_2)$ donne bien $\mathcal{EM}_{\sqsubseteq}(T_1, T_2)$.

Remarque 1 *Pour la suite de ce mémoire, quand aucune confusion ne sera possible on utilisera le mot* occurrence *d'un arbre T pour faire référence à une occurrence de la racine de T.*

3 Arbres fréquents

Nous avons vu dans l'introduction de ce mémoire que des documents semi-structurés peuvent être représentés par des arbres. Une collection de tels documents est donc représentable par un ensemble d'arbres, que l'on va regrouper dans une structure appelée *base de données d'arbres*.

Définition 11 (Base de données d'arbres) *Soit $F = \{T_1, ..., T_m\}$ un ensemble d'arbres étiquetés à étiquettes dans E, et soit $\Delta = \{id_1, ..., id_m\}$ un ensemble d'identifiants tels que pour tout $i \in [1, m]$ on associe l'arbre T_i à l'identifiant id_i.*

Une base de données d'arbres $\mathcal{BA}$ est définie par le triplet (D, Δ, δ), avec :

- *D est un arbre étiqueté ordonné à étiquettes dans E, appelé* **data tree***. La racine r de D est un nœud particulier, dont l'étiquette e_0 ne fait pas partie de E. Les fils de D sont les arbres $T_1, ..., T_m$: $\forall i \in [1, m]$ $D[r_i] = T_i$ (où r_i est la racine de T_i).*
- *$\delta : N_T \backslash r \mapsto \Delta$ est une fonction d'identification des nœuds des arbres, définie comme suit. Soit $D[r_i]$ un sous-arbre dont la racine est le i-ème fils de r. Alors δ est la fonction telle que pour tout $v \in N_{D[r_i]}$ on a $\delta(v) = id_i$.*

La figure II.7 présente un ensemble d'arbres et la base de données d'arbre associée.

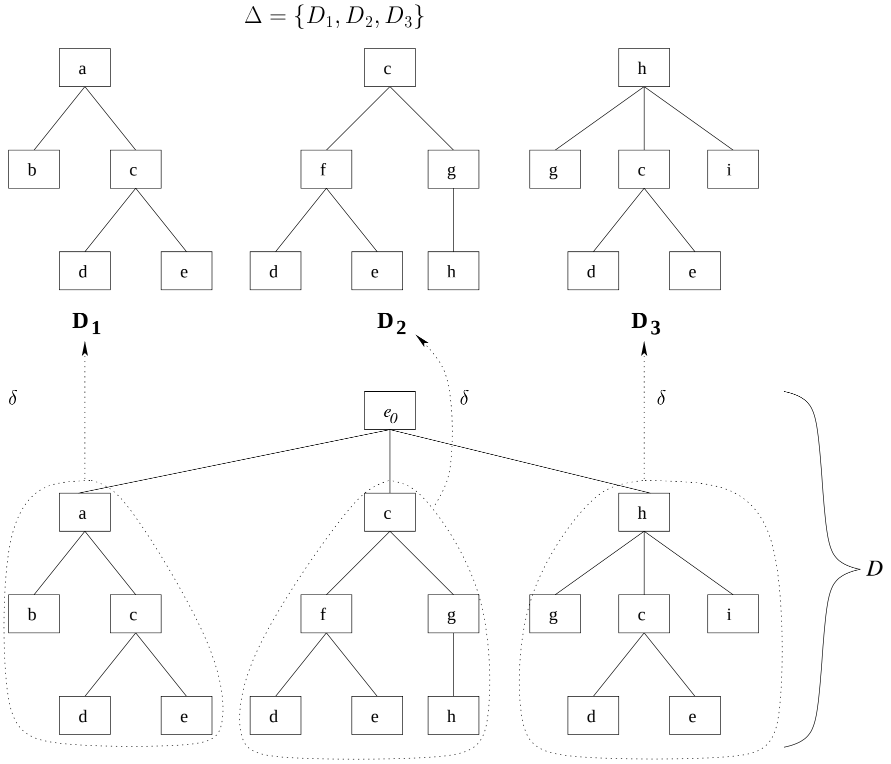


FIG. II.7 – Un exemple de base de données d'arbre

Définition 12 (projeté d'un arbre dans un datatree) *Soit $\mathcal{BA} = (D, \Delta, \delta)$ une base de données d'arbres, $\sqsubseteq$ une définition d'inclusion et P un arbre tel que $P \sqsubseteq D$.*

*Le **projeté** de P dans D, noté $projete(P, D)$, est le datatree dont la racine est la racine de D, et dont chaque nœud n est un nœud de D qui est image d'un noeud de P par un mapping de $\mathcal{EM}_{\sqsubseteq}(P, D)$. Dans $projete(P, D)$, on a $n_1 = parent(n_2)$ si dans D, $n_1 = ancetre(n_2)$ et il n'existe aucun nœud de $projete(P, D)$ dans le chemin qui va de n_1 à n_2 dans D.*

Propriété 2 *Par construction, ce projeté est unique.*

Le projeté d'un arbre P dans le datatree de la figure II.7 est illustré dans la figure II.8.

Pour deux arbres P et P' tels que $P' \sqsubseteq P$, comme on le verra au chapitre V, on peut être amené à calculer les occurrences de P' dans le projeté de P, pour les comparer aux occurrences de P' dans D. Dans le cas particulier où tous les mappings entre P' et P envoient la racine de P' sur la racine de P (i.e. $Locc_{\sqsubseteq}(P', P) = \{racine(P)\}$), l'ensemble des occurrences de P' dans le projeté de P dans D se confond avec l'ensemble des occurrences de P dans D. Cette propriété est formalisée dans la propriété 3 et sera exploitée dans les algorithmes du chapitre V.

Propriété 3 *Soit $\mathcal{BA} = (D, \Delta, \delta)$ une base de données d'arbres, $\sqsubseteq$ une définition d'inclusion, P et P' deux arbres tel que $P' \sqsubseteq P \sqsubseteq D$.*

Si $Locc_{\sqsubseteq}(P', P) = \{racine(P)\}$ alors $Locc_{\sqsubseteq}(P', projete(P, D)) = Locc_{\sqsubseteq}(P, D)$.

Le problème qui nous intéresse est de trouver des motifs fréquents de type arbre dans une base de données d'arbres. Un motif fréquent de type arbre, que nous appelerons *arbre fréquent* dans la

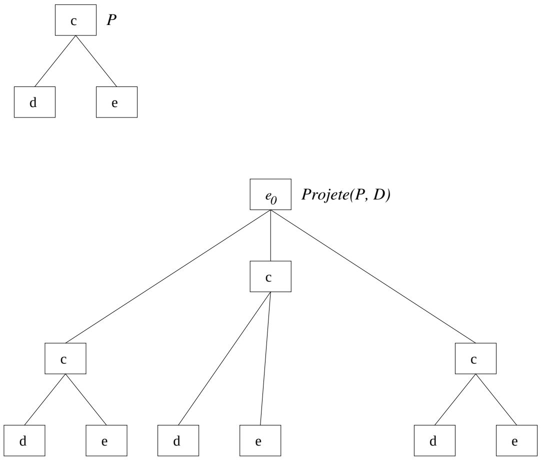


FIG. II.8 – Projeté d'un arbre P dans le datatree D de la figure II.7

suite, est un arbre qui pour une inclusion donnée a un nombre d'occurrences qui dépasse un seuil fixé.

Pour une définition d'inclusion donnée, il y a plusieurs manières de définir la fréquence d'un arbre. On peut en effet définir :

- la fréquence par occurrences, basée sur le nombre d'occurrences de l'arbre dans le datatree
- la fréquence par identifiants, qui compte le nombre d'arbres distincts (d'identifiants différents) du datatree dans lesquels l'arbre est inclus.

La différence entre ces deux fréquences est que l'une est basée uniquement sur le nombre d'occurrences de l'arbre dans le datatree, tandis que l'autre impose que ces occurrences apparaissent dans des arbres distincts.

Définition 13 (Arbre fréquent par occurrences) *Soit $\mathcal{BA} = (D, \Delta, \delta)$ une base de données d'arbres, $\sqsubseteq$ une définition d'inclusion, et ε un seuil de fréquence. Alors P est un arbre fréquent par occurrences de $\mathcal{BA}$ si P a au moins ε occurrences dans D, c'est-à-dire $\mid Locc_{\sqsubseteq}(P, D) \mid \geq \varepsilon$*

*Le **support d'occurrences** de P, noté $S_o(P)$ sera donc défini comme suit :*

$$S_o(P) = \mid Locc_{\sqsubseteq}(P, D) \mid$$

Définition 14 (Arbre fréquent par identifiants) *Soit $\mathcal{BA} = (D, \Delta, \delta)$ une base de données d'arbres, $\sqsubseteq$ une définition d'inclusion, et ε un seuil de fréquence. Soit P un arbre inclus dans D, nous appelerons* **lid** *la liste des identifiants des arbres de D dans lesquels P est inclus. Cette liste est l'image par δ des éléments de $Locc_{\sqsubseteq}(P, D)$, on a donc $lid(P, D) = \{d_i \mid d_i = \delta(u) \text{ et } u \in Locc_{\sqsubseteq}(P, D)\}$.*

Alors P est un arbre fréquent par identifiants de $\mathcal{BA}$ si P est inclus dans au moins ε_a arbres distincts de $\mathcal{BA}$: $\mid lid(P, D) \geq \varepsilon \mid$

*Le **support d'identifiants** de P, noté $S_D(P)$ sera donc défini comme suit :*

$$S_D(P) = \mid lid(P, D) \mid$$

Remarque 2 *Si un arbre AF est fréquent par identifiants, alors il est aussi fréquent par occurrences.*

La figure II.9 montre qu'avec un seuil absolu $\varepsilon = 2$, l'arbre de racine A ayant les fils B et C est fréquent par occurrences (l'arbre a 2 occurrences), mais pas par identifiants (l'arbre n'est inclus que dans un seul des arbres du datatree). Par contre, l'arbre de racine D ayant pour seul fils F est inclus dans les deux arbres identifiés par T_1 et T_2 du datatree : il est donc fréquent par identifiants.

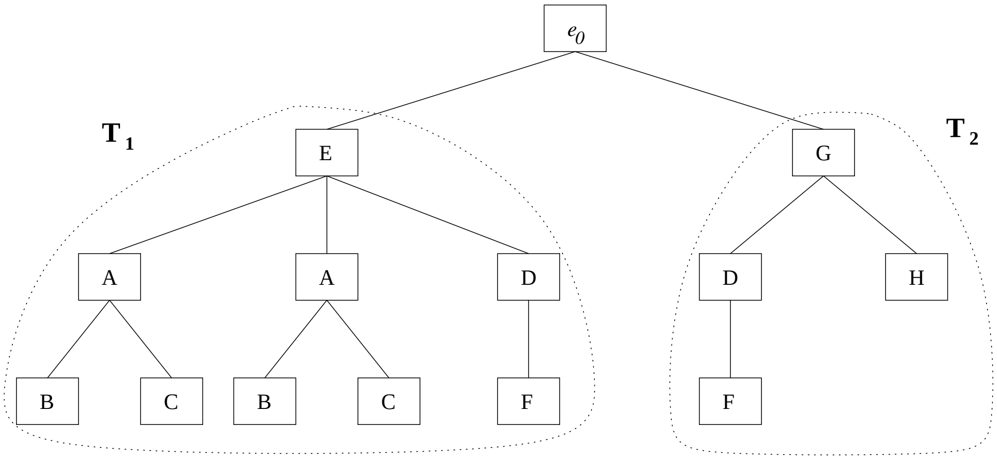


Fig. II.9 – Un exemple des différences entre les définitions de fréquence

Il est important de remarquer qu'une occurrence ou qu'un identifiant peut être vu comme un regroupement de plusieurs mappings de P dans D ayant une propriété en commun. Pour une occurrence, cette propriété est de donner la même image de la racine ; pour un identifiant, c'est de donner le même identifiant d'arbres pour les différentes images de la racine.

4 Exemple récapitulatif

Considérons la base de données d'arbres $\mathcal{BA}$ dont l'arbre D_X est représenté sur la figure II.10.

Avec un seuil de fréquence de 2 et une inclusion avec préservation stricte des étiquettes, le plus gros arbre fréquent que l'on puisse trouver est l'arbre AF représenté sur la figure II.11.

On a : $Locc_{\sqsubseteq}(AF, D) = \{n_1, n_6\}$.

On en déduit $\mathcal{EM}_{\sqsubseteq}(AF, D) = \mathcal{EM}_{n_1}(AF, D) \cup \mathcal{EM}_{n_6}(AF, D)$. Ces ensembles sont représentés dans le tableau II.1.

	$\mathcal{EM}_{n_1}(AF, D)$		$\mathcal{EM}_{n_6}(AF, D)$
μ :	$A \to n_1$	μ' :	$A \to n_6$
	$B \to n_2$		$B \to n_{10}$
	$C \to n_4$		$C \to n_8$
	$D \to n_3$		$D \to n_{12}$
		μ'' :	$A \to n_6$
			$B \to n_{10}$
			$C \to n_8$
			$D \to n_{13}$

Tab. II.1 – $\mathcal{EM}_{\sqsubseteq}(AF, D)$

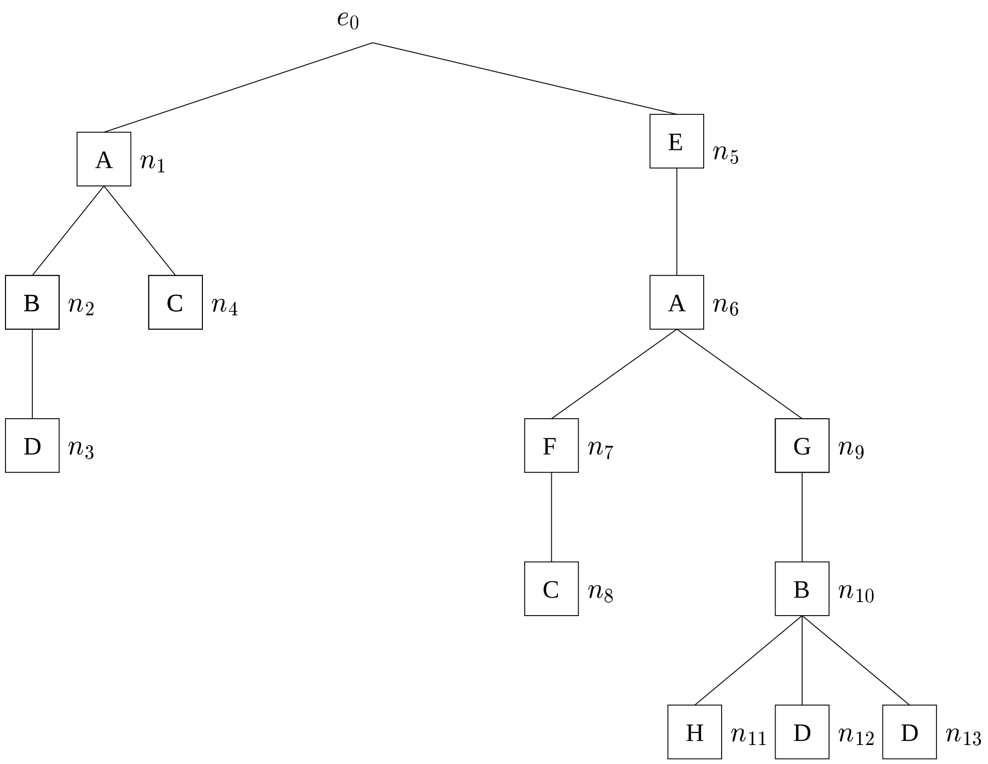


FIG. II.10 – Base de données d'arbre D_X

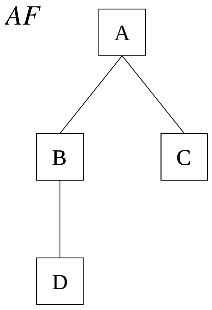


FIG. II.11 – Le plus gros arbre fréquent de la figure II.10, seuil $\varepsilon = 2$

Chapitre III

État de l'art

Le but de ce chapitre est de faire une synthèse des travaux effectués sur la découverte de motifs fréquents, en relation avec le sujet de notre étude : la découverte de motifs fréquents dans des arbres.

Dans une première partie, nous traiterons des principaux algorithmes créés pour la découverte de motifs fréquents dans des données transactionnelles. Ces algorithmes sont à la base des algorithmes de découverte de motifs fréquents dans des données plus complexes. Ils nous permettront d'introduire un formalisme et une méthode d'exploration de l'espace de recherche auxquels il sera souvent fait référence par la suite.

Dans une seconde partie, nous exposerons plusieurs méthodes découvrant des motifs dans des données structurées de type arbre, dont les méthodes mentionnées dans l'introduction.

1 Données transactionnelles

Avant d'introduire les algorithmes de recherche de motifs fréquents, nous allons poser le cadre formel permettant de caractériser les données sur lesquelles ces algorithmes opèrent, ainsi que les résultats qu'ils trouvent.

1.1 Définitions

Soit $\mathcal{I} = \{i_1, ..., i_n\}$ un ensemble de n littéraux, appelés *items*. À partir de $\mathcal{I}$ on peut donner les définitions suivantes :

Définition 15 (Itemset) *Tout sous-ensemble $I \subseteq \mathcal{I}$ de $\mathcal{I}$ sera appelé itemset. Un itemset à k littéraux sera appelé k-itemset (avec $k \leq n$).*

La taille de l'ensemble des itemsets de $\mathcal{I}$ est $2^{|\mathcal{I}|}$.

Si l'on ordonne l'ensemble des itemsets par la relation d'inclusion ensembliste, on obtient un treillis complet.

Exemple : Soit $\mathcal{I} = \{A, B, C, D, E\}$. Le diagramme de Hasse [Wille 1982] associé au treillis des itemsets de $\mathcal{I}$ est représenté par la figure III.1.

Définition 16 (Transaction) *Une transaction $t = (id_t, I_t)$ est un couple composé d'un identifiant unique id_t et d'un itemset I_t.*

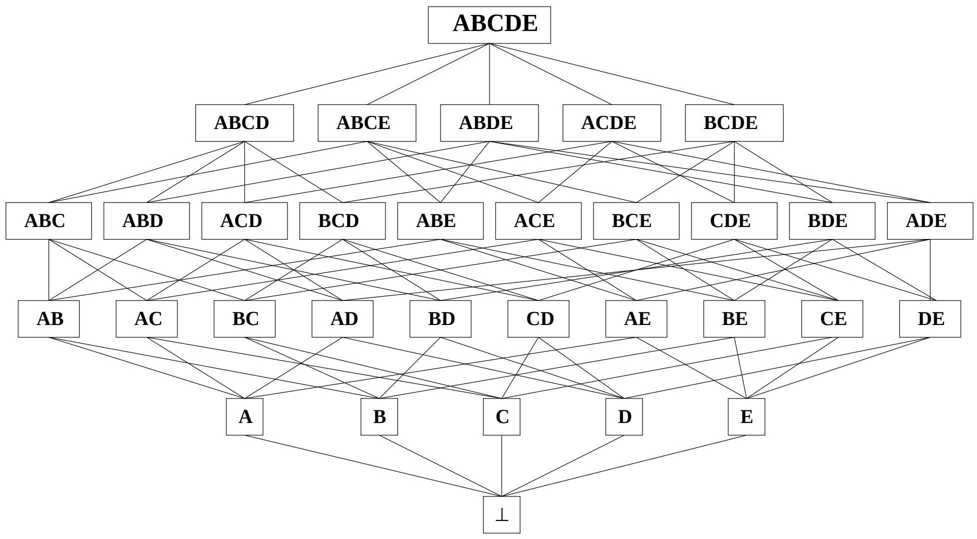


FIG. III.1 – Treillis des itemsets de $\mathcal{I} = \{A, B, C, D, E\}$

Définition 17 (Base de données transactionnelle) *Une base de données transactionnelle* $\mathcal{B} = \{t_1, ..., t_m\}$ *est un ensemble de* m *transactions.*

Exemple : La base de données transactionnelle $\mathcal{B}_x$ que nous allons utiliser pour le reste de ce chapitre est donnée dans la table III.1.

id_t	I_t
1	A B C D E
2	A B C E
3	C
4	B C
5	A B C D
6	A B C

TAB. III.1 – La base de données transactionnelle $\mathcal{B}_x$

Cette dernière définition nous permet d'introduire la notion de *fréquence.*

Définition 18 (Itemset fréquent) *Soit une base de données transactionnelle* $\mathcal{B}$ *et un seuil de fréquence absolu* ε_a*. Un itemset* I *est fréquent s'il est inclus dans au moins* ε_a *transactions de* $\mathcal{B}$*.*

On appelera ***ensemble-support*** *de* I*, noté* $\mathcal{L}(I)$*, l'ensemble des identifiants des transactions dans lesquelles* I *est inclus.*

$$\mathcal{L}(I) = \{id_t \mid t : (id_t, I_t) \in \mathcal{B} \wedge I \subseteq I_t\}$$

On appelera ***support*** *de* I*, noté* $S(I)$*, le nombre de transactions dans lesquelles* I *est inclus.*

$$S(I) = |\mathcal{L}(I)|$$

L'ensemble des itemsets fréquents est :

$$IF = \{I \subset \mathcal{I} \mid S(I) \geq \varepsilon_a\}$$

Remarque 3 *Dans la définition précédente, on peut remplacer la donnée d'un seuil de fréquence absolu par celle d'un seuil de fréquence relatif $\varepsilon_r \in [0,1]$. Le seuil de fréquence absolu correspondant est donné par l'équation :*

$$\varepsilon_a = \varepsilon_r \times |\mathcal{B}|$$

Exemple : Considérons la base de données transactionnelle $\mathcal{B}_x$ de l'exemple précédent, et un seuil de fréquence relatif $\varepsilon_r = 2/6$ (soit $\varepsilon_a = 2$). Les itemsets fréquents sont les itemsets entourés d'un carré solide dans la figure III.2, tandis que les itemsets infréquents sont entourés d'un carré en pointillés.

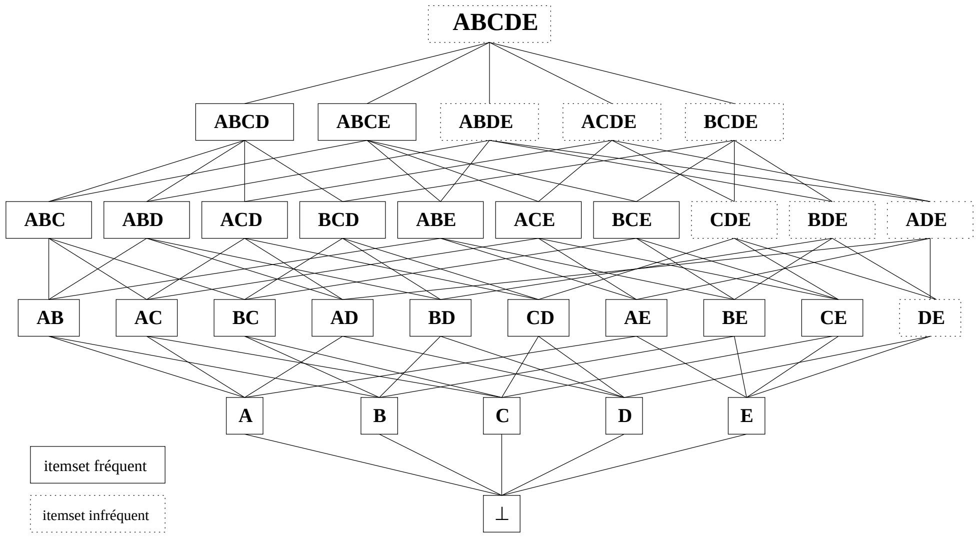


FIG. III.2 – Treillis des itemsets fréquents pour la base $\mathcal{B}_x$ et le seuil $\varepsilon_a = 2$

Les itemsets fréquents et leurs ensembles-support sont de plus récapitulés dans le tableau III.2.

1.2 Apriori

Les premiers algorithmes de calcul des itemsets fréquents sont AIS [Agrawal et al. 1993] et SETM [Houtsma et Swami 1995]. Ces algorithmes sont très coûteux à exécuter, et ne sont pas utilisables en pratique sur de grandes bases de données. Nous allons présenter l'algorithme le plus connu de découverte des itemsets fréquents, Apriori [Agrawal et Srikant 1994], qui permet de beaucoup réduire le temps de découverte des itemsets fréquents par rapport à ces deux algorithmes.

Apriori est un algorithme qui recherche, étant donné un seuil de fréquence, tous les itemsets fréquents d'une base de données transactionnelle. C'est un algorithme de type "générer et tester" par niveaux : à la $k^{\text{ème}}$ itération l'algorithme génère des k-itemsets candidats en se basant sur les $k-1$-itemsets fréquents trouvés à l'itération précédente. Ensuite, la fréquence des k-itemsets candidats est évaluée par un balayage de la base de données transactionnelle.

Pour éviter les redondances dans les itemsets candidats produits, l'ensemble des itemsets est muni de l'ordre lexicographique.

De plus, afin de réduire le nombre d'itemsets candidats générés, l'algorithme Apriori s'appuie sur deux propriétés d'anti-monotonie :

Propriété 4 *Tous les sous-ensembles d'un itemset fréquent sont fréquents.*

I	$S(I)$
{A}	{1, 2, 5, 6}
{B}	{1, 2, 4, 5, 6}
{C}	{1, 2, 3, 4, 5, 6}
{D}	{1, 5}
{E}	{1, 2}
{A, B}	{1, 2, 5, 6}
{A, C}	{1, 2, 5, 6}
{A, D}	{1, 5}
{A, E}	{1, 2}
{B, C}	{1, 2, 4, 5, 6}
{B, D}	{1, 5}
{B, E}	{1, 2}
{C, D}	{1, 5}
{C, E}	{1, 2}
{A, B, C}	{1, 2, 5, 6}
{A, B, D}	{1, 5}
{A, B, E}	{1, 2}
{A, C, D}	{1, 5}
{A, C, E}	{1, 2}
{B, C, D}	{1, 5}
{B, C, E}	{1, 2}
{A, B, C, D}	{1, 5}
{A, B, C, E}	{1, 2}

TAB. III.2 – Les itemsets fréquents de $\mathcal{B}_x$ et leurs ensembles-support, pour $\varepsilon_a = 2$

Donc, pour générer un k-itemset candidat, il faut que ce soit un sur-ensemble de $k-1$-itemsets fréquents.

Propriété 5 *Tous les sur-ensembles d'un itemset infréquent sont infréquents.*

Donc si un k-itemset candidat est un sur-ensemble d'un $k-1$-itemset infréquent, alors on peut l'éliminer de la liste des candidats car il ne peut pas être fréquent.

Le pseudo-code de l'algorithme Apriori est donné dans l'algorithme 3. Les notations utilisées sont :
- C_k : ensemble des k-itemsets candidats, chaque candidat possède deux champs : *itemset* et *support*. Le champ *support* est initialisé à 0.
- L_k : ensemble des k-itemsets fréquents

Algorithme 3 Apriori

Entrée : $\mathcal{B}$ une base de données transactionnelle, ε_a un seuil de fréquence absolu
Sortie : L'ensemble de tous les itemsets fréquents de $\mathcal{B}$

1: $L_1 \leftarrow \{$1-itemsets fréquents$\}$
2: **Pour** $(k = 2\ ;\ L_{k-1} \neq \emptyset\ ;\ k++)$ **Faire**
3:     $C_k \leftarrow Apriori\_Gen(L_{k-1})$ // *Voir algorithme 4*
4:     **Pour tout** $t \in \mathcal{B}$ **Faire**
5:         $C_t \leftarrow Subset(C_k, t)$
6:         **Pour tout** $c \in C_t$ **Faire**
7:             $c.support++$
8:         **Fin Pour**
9:     **Fin Pour**
10:     $L_k \leftarrow \{c \in C_k \mid c.support \geq \varepsilon_a\}$
11: **Fin Pour**
12: **Retourner** $\cup_k L_k$

La première étape de l'algorithme (ligne 1) consiste en un balayage sur la base de données, elle permet de déterminer tous les 1-itemsets fréquents. Le reste de l'algorithme (lignes 2 à 10) est une itération de génération de candidats, et de test de ces candidats. À la ligne 3, les itemsets candidats de longueur k sont produits par la fonction *Apriori_Gen*. Les lignes 4 à 9 permettent de calculer le support de ces itemsets candidats. Pour cela, la boucle commencée à la ligne 4 parcourt toute la base de données. Pour chaque transaction rencontrée, la fonction *Subset* (ligne 5) détermine les itemsets de C_k contenus dans cette transaction. Cette fonction est basée sur des tests d'inclusion d'itemsets, elle est optimisée par l'utilisation de tables de hachage (non décrites ici). Le support de ces itemsets candidats est mis à jour ligne 7.

La sélection des itemsets fréquents est effectuée ligne 10. L'algorithme s'arrête quand il ne trouve plus de nouveaux itemsets fréquents : $L_{k-1} = \emptyset$ dans ce cas.

Fonction Apriori_Gen : La fonction *Apriori_Gen* assure la génération des itemsets candidats pour l'algorithme Apriori. Son pseudo-code est donné dans l'algorithme 4.

Cette fonction a deux phases : une phase de *jointure*, où les k-itemsets candidats sont créés à partir des $k-1$-itemsets fréquents (lignes 2 à 5). Un itemset candidat est créé à partir de deux $k-1$-itemsets fréquents p et q si les $k-2$ premiers items de p et q sont identiques, et que le dernier item de q est inférieur à celui de p. Nous rappelons ici que l'ordre utilisé est l'ordre lexicographique. On utilise pour cette phase la première propriété d'antimonotonie, car tous les k-itemsets candidats créés de cette manière sont des sur-ensembles d'itemsets fréquents.

Algorithme 4 *Apriori_Gen*

Entrée : L_{k-1} l'ensemble des $k-1$-itemsets fréquents
Sortie : C_k l'ensemble des k-itemsets candidats

1: // *Étape de jointure*
2: **insérer dans** C_k
3: **select** $p.itemset[1], p.itemset[2], ..., p.itemset[k-1], q.itemset[k-1]$
4: **from** L_{k-1} p, L_{k-1} q
5: **where** $p.itemset[1] = q.itemset[1], ..., p.itemset[k-2] = q.itemset[k-2], p.itemset[k-1] = q.itemset[k-1]$
6: // *Étape d'élagage*
7: **Pour tout** itemset $c \in C_k$ **Faire**
8:     **Pour tout** sous-ensemble s de c tel que $|s| = k-1$ **Faire**
9:         **Si** $s \notin L_{k-1}$ **Alors**
10:             **supprimer** c de C_k
11:         **Fin Si**
12:     **Fin Pour**
13: **Fin Pour**
14: **Retourner** C_k

Mais ces k-itemsets candidats peuvent de plus être des sur-ensembles d'itemsets infréquents. On utilise la deuxième propriété d'antimonotonie dans la phase d'élagage (lignes 7 à 13) : on supprime tous les itemsets candidats dont au moins l'un des sous-ensembles de longueur $k-1$ est infréquent.

Exemple : Une exécution de l'algorithme Apriori sur notre base de données $\mathcal{B}_x$ des exemples précédents avec le seuil de fréquence relatif $\varepsilon_r = 2/6$ est donnée dans la table III.3. Chaque itération de l'algorithme est représentée par une ligne du tableau. La première colonne correspond au numéro k de l'itération en cours, la deuxième colonne donne le C_k produit par *Apriori_Gen*, la troisième colonne donne le C_k avec leurs supports une fois que la base de données a été examinée, et la dernière colonne donne le L_k une fois que les itemsets infréquents ont été éliminés.

Complexité d'Apriori **:** Soit $\mathcal{B}$ une base de données de m transactions, et $\mathcal{I}$ un ensemble de n items. A l'étape k la complexité de la génération de candidats par *Apriori_Gen* est dominée par la complexité de la phase d'*élagage*, qui effectue $O(k.|C_k|)$ tests : il y a $|C_k|$ itérations, et pour chaque itération k tests. Le calcul du support des itemsets candidats sur la base a une complexité, dans le pire des cas, de $O(m.|C_k|)$. Le test de fréquence permettant la construction de L_k a une complexité en $O(|C_k|)$.

On en déduit que si m (la taille de la base de données) est grand, la complexité de l'algorithme est dominée par la phase de calcul du support des itemsets candidats, c'est à dire par $O(m.|C_k|)$. La complexité totale de l'algorithme peut donc être approximée par $O(m.\sum_k C_k)$. Dans le pire des cas, l'ensemble des itemsets candidats est l'ensemble de tous les itemsets possibles, ce qui donne une complexité au pire en $O(m.2^n)$.

La complexité dans le cas le pire de l'algorithme Apriori est donc linéaire en fonction de la taille de la base de données, mais exponentielle en fonction du nombre d'items.

En pratique, l'algorithme Apriori passe bien à l'échelle à condition :

- que le seuil de fréquence ne soit pas trop petit
- que la base de données soit peu dense, c'est à dire que les transactions contiennent peu d'items, et que ces items soient faiblement corrélés entre eux.

Si par contre la base de données est dense, et que le seuil de fréquence est petit, alors il y aura

Itération k	C_k	C_k + support	L_k + support
1	{A}	{A} 4/6	{A} 4/6
	{B}	{B} 5/6	{B} 5/6
	{C}	{C} 6/6	{C} 6/6
	{D}	{D} 2/6	{D} 2/6
	{E}	{E} 2/6	{E} 2/6
2	{A B}	{A B} 4/6	{A B} 4/6
	{A C}	{A C} 4/6	{A C} 4/6
	{A D}	{A D} 2/6	{A D} 2/6
	{A E}	{A E} 2/6	{A E} 2/6
	{B C}	{B C} 5/6	{B C} 5/6
	{B D}	{B D} 2/6	{B D} 2/6
	{B E}	{B E} 2/6	{B E} 2/6
	{C D}	{C D} 2/6	{C D} 2/6
	{C E}	{C E} 2/6	{C E} 2/6
	{D E}	*{D E} 1/6*	
3	{A B C}	{A B C} 4/6	{A B C} 4/6
	{A B D}	{A B D} 2/6	{A B D} 2/6
	{A B E}	{A B E} 2/6	{A B E} 2/6
	{A C D}	{A C D} 2/6	{A C D} 2/6
	{A C E}	{A C E} 2/6	{A C E} 2/6
	{B C D}	{B C D} 2/6	{B C D} 2/6
	{B C E}	{B C E} 2/6	{B C E} 2/6
	{B D E}	*{B D E} 1/6*	
	{C D E}	*{C D E} 1/6*	
4	{A B C D}	{A B C D} 2/6	{A B C D} 2/6
	{A B C E}	{A B C E} 2/6	{A B C E} 2/6
	{A C D E}	*{A C D E} 1/6*	
	{B C D E}	*{B C D E} 1/6*	

TAB. III.3 – Exécution de l'algorithme APRIORI sur $\mathcal{B}_x$, avec $\varepsilon_r = 2/6$

beaucoup d'itemsets fréquents de grande taille à découvrir. Or l'algorithme Apriori fait autant d'itérations qu'il y a d'items dans le plus grand itemset fréquent à découvrir. Chacune de ces itérations comprend un coûteux balayage de la base de données. Réduire le nombre de ces balayages est critique pour l'efficacité de l'algorithme.

De plus, plus il y a d'itemsets candidats dans C_k, plus le balayage de la base de données sera coûteux. Réduire au maximum le nombre d'itemsets candidats est donc également une priorité en vue d'améliorer Apriori.

Ces deux directions de recherche ont engendré de nombreux travaux d'optimisation de l'algorithme Apriori. Nous verrons en 1.3 les améliorations que reprennent la structure générale de l'algorithme, en perfectionnant les méthodes de génération et de test. En 1.4, nous introduirons d'autres approches, dites *verticales*, qui découvrent les itemsets fréquent avec encore plus d'efficacité.

1.3 Améliorations d'Apriori

1.3.1 Optimiser la gestion de la mémoire

Le principal problème avec les balayages de la base de données est que cette base est en général beaucoup trop volumineuse pour tenir en mémoire vive. Elle est donc stockée sur le disque, d'où le coût élevé des accès à cette base. La plupart des algorithmes parus immédiatement après Apriori, comme *AprioriTid* [Agrawal et Srikant 1994], *Partition* [Savasere et al. 1995], *Sampling* [Toivonen 1996] et *DIC* [Brin et al. 1997] prennent en compte cette limitation, et proposent des modifications dans la manière de manipuler la mémoire. *Partition*, par exemple, découpe la base de données en plusieurs parties dont chacune peut tenir en mémoire vive. Il effectue un traitement Apriori sur chacune de ces partitions, puis une jointure adéquate sur les résultats. L'algorithme *Sampling* va encore plus loin : pour pouvoir gérer de très grosses bases de données, il calcule les itemsets fréquents sur un échantillon de la base de données, avant de raffiner ses résultats grâce à un balayage supplémentaire sur la base de données entière. Mais cet algorithme ne garantit pas de trouver tous les itemsets fréquents.

1.3.2 Rechercher des itemsets fréquents *maximaux*

Les algorithmes précédemment mentionnés améliorent tous les performances de l'algorithme Apriori initial, et permettent de rechercher des itemsets fréquents dans des bases de données de plus en plus grosses.

Toutefois, tout comme l'algorithme Apriori, ces algorithmes ne passent pas à l'échelle sur des bases de données où la taille des itemsets fréquents les plus longs est grande.

Pour ce type de bases de données, de nouveaux algorithmes ont été créés, avec un but différent : le but de ces nouveaux algorithmes est de découvrir des itemsets fréquents *maximaux* sur la base de données.

Définition 19 (Itemset fréquent maximal) *Soit un ensemble d'items $\mathcal{I}$, une base de données transactionnelle $\mathcal{B}$, et un seuil de fréquence ε. Soit I un itemset fréquent de $\mathcal{B}$ par rapport au seuil ε. I est* ***maximal*** *ssi il n'existe pas d'itemset fréquent $I' \subseteq \mathcal{I}$ tel que $I \subset I'$.*

Formellement, l'ensemble des itemsets fréquents maximaux est :

$$IFM = \{I \in IF \mid \forall I' \supset I \;\; I' \notin IF\}$$

Rechercher les itemsets fréquents maximaux au lieu de tous les itemsets fréquents permet de réduire de plusieurs ordres de grandeur la taille de l'espace de recherche, donc permet d'avoir des ensembles d'itemsets fréquents maximaux candidats moins volumineux que dans des algorithmes de type Apriori. Pour réduire le nombre de balayages effectués sur la base de données, les algorithmes de recherche d'itemsets fréquents maximaux font un double parcours de l'espace de recherche. Ils utilisent un parcours du bas vers le haut, comme les algorithmes Apriori classiques (commencer par les plus petits itemsets et les faire grossir), mais aussi, simultanément, un parcours du haut vers le bas, comme le montre la figure III.3. Ce parcours du haut vers le bas recherche en priorité les itemsets fréquents maximaux les plus gros. Dans le temps où le parcours du bas vers le haut avance d'un niveau, le parcours du haut vers le bas est lui autorisé à descendre de plusieurs niveaux. Cela lui permet de trouver plus vite les itemsets fréquents maximaux de grande taille. L'avantage le plus remarquable de ce deuxième parcours est que lorsqu'un itemset fréquent maximal de grande taille est identifié, ses nombreux sous-ensembles n'ont plus besoin d'être considérés, ce qui réduit le travail du parcours classique du bas vers le haut.

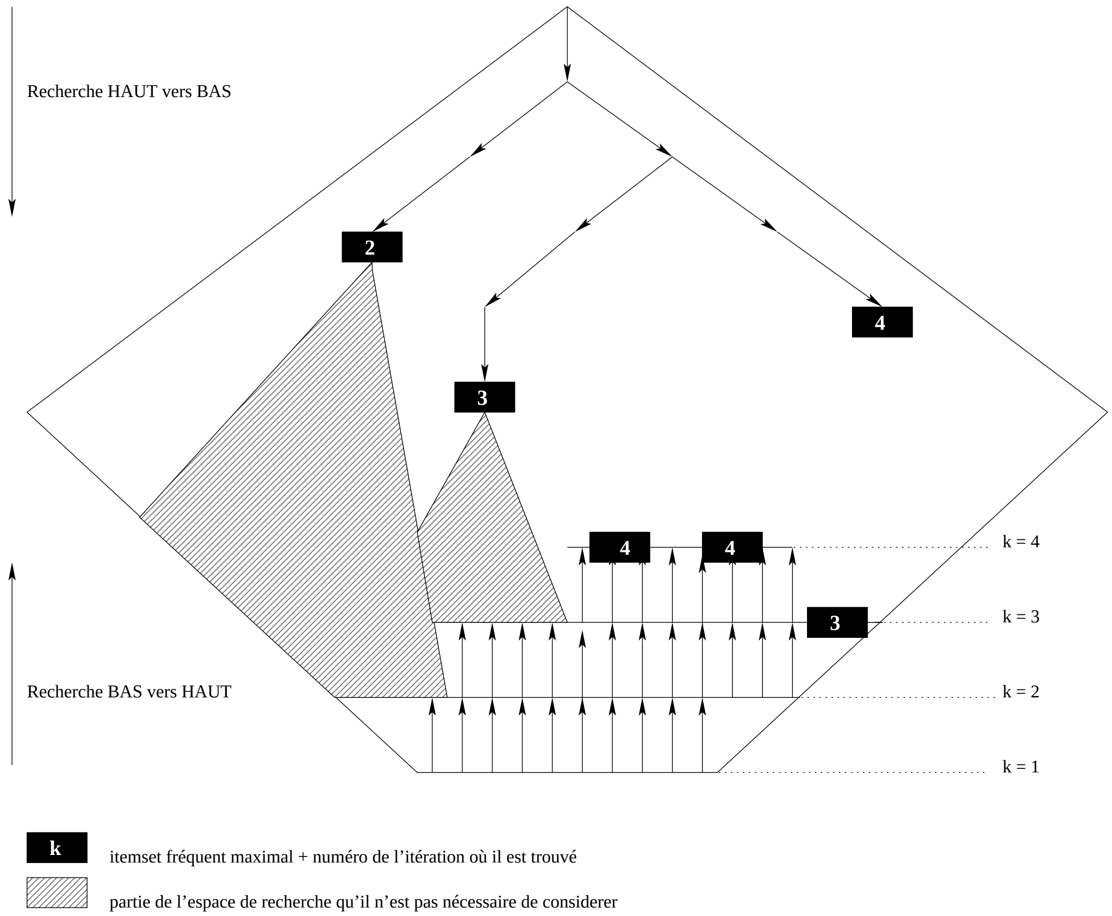


Fig. III.3 – Schéma de la double recherche effectuée par un algorithme recherchant des itemsets fréquents maximaux

Plusieurs algorithmes implémentent cette approche. Citons en particulier Pincer-Search [Lin et Kedem 1998] et Max-Miner [Jr. 1998]. L'algorithme Max-Miner, proposé par Bayardo en 1998, est l'un des plus rapides pour la découverte d'itemsets fréquents maximaux.

1.3.3 Rechercher des itemsets fréquents *fermés*

Le défaut des algorithmes de recherche d'itemsets fréquents maximaux est que la réduction de l'espace de recherche qu'ils effectuent s'accompagne d'une perte d'information. À partir du résultat de ces algorithmes, à savoir les itemsets fréquents maximaux, s'il est possible de retrouver tous les itemsets fréquents, il n'est en revanche pas possible de retrouver leur support exact.

Dans certains cas, en particulier pour calculer des règles d'association entre itemsets fréquents [Agrawal et Srikant 1994], cette perte n'est pas acceptable. Pasquier a proposé en 1999 [Pasquier et al. 1999] une méthode permettant de réduire la taille de l'espace de recherche tout en permettant de découvrir tous les itemsets fréquents.

Cette méthode consiste à rechercher les itemsets fréquents *fermés* de la base de données transactionnelle.

Définition 20 (Itemset fréquent fermé) *Soit $\mathcal{I}$ un ensemble d'items, $\mathcal{B}$ une base de données transactionnelle sur $\mathcal{I}$, et ε un seuil de fréquence. Soit $I \subset \mathcal{I}$ un itemset fréquent de $\mathcal{B}$ par rapport au seuil ε. I est* ***fermé*** *s'il n'existe pas d'itemset fréquent $I' \supset I$ tel que $S(I') = S(I)$.*

Formellement, l'ensemble des itemsets fréquents fermés est caractérisé comme suit :

$$IFF = \{I \in IF \mid \nexists I' \supset I \ tq \ S(I') = S(I)\}$$

Les trois ensembles d'itemsets fréquents que nous avons définis jusqu'ici s'organisent comme suit : $IFM \subseteq IFF \subseteq IF$. En général, IFM est un ordre de grandeur plus petit que IFF, qui lui même est un ordre de grandeur plus petit que IF.

Contrairement à l'ensemble des itemsets fréquents maximaux, l'ensemble d'itemsets fréquents fermés est sans perte d'informations par rapport à l'ensemble des itemsets fréquents. Ainsi, pour savoir si un itemset I est fréquent, il suffit de trouver le plus petit itemset fréquent fermé qui soit un sur-ensemble de I. Si ce sur-ensemble n'existe pas, alors I est infréquent. Sinon, I est fréquent, et il a le même support que son plus petit sur-ensemble fermé fréquent.

La figure III.4 montre une comparaison entre le treillis des itemsets fréquents et le treillis des itemsets fréquents fermés, ainsi que les itemsets maximaux.

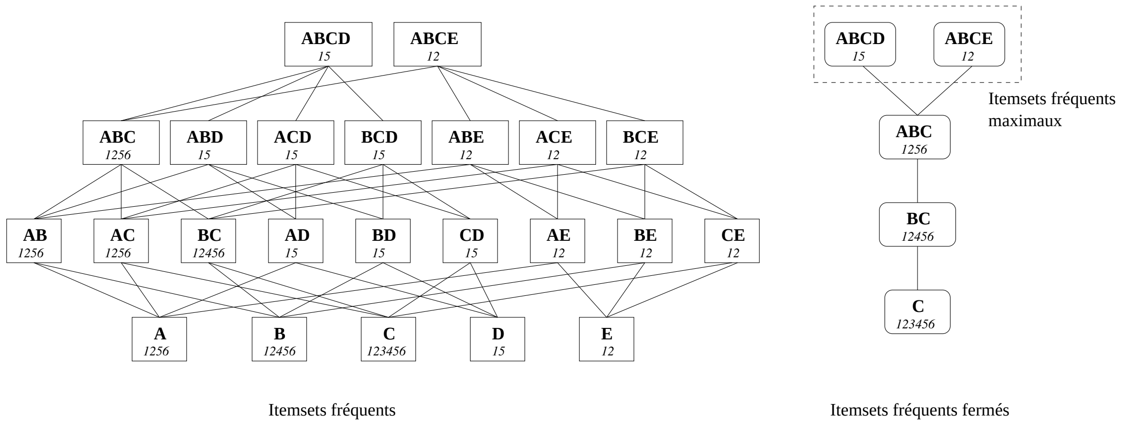


FIG. III.4 – Les treillis des itemsets fréquents, des itemsets fréquents fermés, et des itemsets fréquents maximaux pour la base exemple $\mathcal{B}_x$

Pasquier a implémenté la recherche d'itemsets fréquents fermés dans l'algorithme *Close*. Cet algorithme est très proche par sa structure de l'algorithme APRIORI : c'est un algorithme qui explore l'espace de recherche en largeur d'abord, du bas vers le haut. Seules changent les procédures de génération de candidats et de comptage du support par rapport à Apriori, pour s'adapter à la découverte d'itemsets fréquents fermés.

Cet algorithme a par la suite été amélioré par Zaki avec Charm [Zaki et Hsiao 2002], puis par Han avec Closet [Pei et al. 2000]. Les algorithmes à base d'itemsets fermés fréquents sont actuellement parmi les algorithmes les plus efficaces pour l'analyse de bases de données transactionnelles.

1.4 Une approche verticale : l'algorithme Eclat

La recherche d'itemsets fréquents fermés a permis d'améliorer d'une manière significative les algorithmes de recherche d'itemsets fréquents. Toutefois, les algorithmes proposés utilisent tous le principe de la recherche *horizontale*, en largeur d'abord que l'on trouve dans Apriori. Cette recherche a le défaut de nécessiter de nombreux et coûteux balayages de la base de données. De plus, pour calculer le support de chaque candidat, il faut faire des tests d'inclusion sur toutes les transactions de la base de données, ce qui là encore est coûteux, surtout si les itemsets fréquents sont de grande taille.

Pour pallier tous ces problèmes, Zaki a proposé de représenter les données sous un autre format, dit *vertical* [Zaki et al. 1997]. Ce format permet de calculer beaucoup plus rapidement le support d'un itemset, et réduit au fur et à mesure la portion de la base qui doit être considérée par l'algorithme. De plus, il s'adapte à de nombreuses stratégies de recherche, que ce soit en largeur d'abord, en profondeur d'abord, ou un hybride des deux.

Le format vertical consiste à adjoindre à chaque itemset I son *ensemble-support* $\mathcal{L}(I)$. La table III.4 montre notre base de données $\mathcal{B}_x$ en format vertical (on considère les 1-itemsets et leurs ensembles-support).

A	B	C	D	E
1	1	1	1	1
2	2	2	5	2
5	4	3		
6	5	4		
	6	5		
		6		

Tab. III.4 – Représentation verticale de $\mathcal{B}_x$

La puissance de la représentation verticale vient de la facilité avec laquelle on calcule l'ensemble-support et donc le support d'un k-itemset à partir des $k-1$-itemsets qui le composent.

Propriété 6 *Soit I un k-itemset, soit I_1 et I_2 deux de ses sous-ensembles à $k-1$ éléments. On a :* $\mathcal{L}(I) = \mathcal{L}(I_1) \cap \mathcal{L}(I_2)$

Le calcul des ensembles-support est donc une simple intersection de listes d'identifiants de transactions.

Une solution simple pour trouver tous les itemsets fréquents consiste donc à parcourir le treillis des itemsets en partant du bas (les 1-itemsets), en calculant au fur et à mesure le support des itemsets de plus grande taille grâce à l'intersection des ensembles-support de leurs sous-ensembles.

Toutefois, si la base de données est grande, les ensembles-support seront très volumineux, et ne pourront pas tenir en mémoire vive. Zaki a résolu ce problème en découpant le treillis en plusieurs *classes d'équivalence*, qui sont des sous-treillis du treillis des itemsets pouvant tenir en mémoire.

La relation d'équivalence considérée par Zaki dans l'algorithme ECLAT [Zaki et al. 1997] est une relation d'équivalence par préfixe. Pour un entier k donné, deux itemsets sont dans la même classe d'équivalence s'ils partagent le même préfixe de longueur k.

Les classes d'équivalences de notre exemple, avec un préfixe k valant 0, sont illustrées sur la figure III.5.

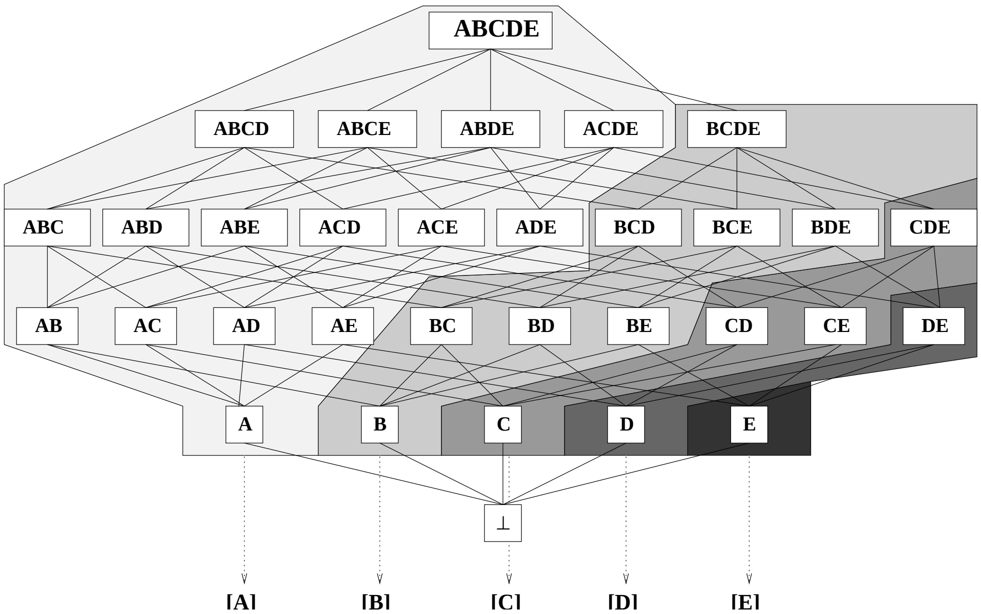


FIG. III.5 – Classes d'équivalences de $\mathcal{B}_x$, pour un préfixe $k = 0$

Cette méthode des classes d'équivalence est récursive : chaque classe d'équivalence trouvée pour un niveau est elle même découpée en sous-classes d'équivalence, avec une longueur de préfixe augmentée de 1.

Par exemple, la classe d'équivalence de $[A]$ (trouvée pour $k = 0$), est elle-même découpée pour $k = 1$ en les sous-classes suivantes : $\{[AB], [AC], [AD], [AE]\}$.

Ce découpage permet de chercher les itemsets fréquents en se restreignant à une classe d'équivalence. En parcourant toutes les classes d'équivalence, on découvre tous les itemsets fréquents.

L'algorithme ECLAT proposé par Zaki est donné dans l'algorithme 5.

La première phase de l'algorithme (ligne 2) calcule L_1 et L_2 de manière horizontale, comme APRIORI. En effet, à ce niveau les itemsets sont petits mais les ensembles-support sont très grands : il n'est pas encore avantageux d'utiliser la représentation verticale. Mais pour les étapes suivantes, la base de données subit une transformation pour être sous forme verticale (ligne 4). Les classes d'équivalence pour un préfixe de longueur 0 sont calculées à la ligne 5, puis des lignes 7 à 9 on découvre les itemsets fréquents de toutes les classes d'équivalences grâce à la procédure récursive $Calcule\_Frequent$.

Le pseudo-code de cette procédure est donné dans l'algorithme 6. La boucle principale des lignes 1 à 5 tente de faire une jointure entre les $k-1$-itemsets de la classe d'équivalence courante. Si la taille de l'intersection des ensembles-support de deux $k-1$-itemsets I_1 et I_2 est supérieure au seuil de fréquence, alors l'itemset $I_1 \cup I_2$ est fréquent, et est donc ajouté à L_k. Par la suite, L_k est lui même partitionné en classes d'équivalence (ligne 6), et la procédure est appelée récursivement sur chacune de ces classes d'équivalence.

L'algorithme ECLAT a montré d'excellentes performances expérimentales. De plus, l'approche

Algorithme 5 ECLAT

Entrée : $\mathcal{B}_x$ une base de données transactionnelle, ε_a un seuil de fréquence relatif
Sortie : L'ensemble de tous les itemsets fréquents

// *Initialisation*
Calculer L_1 et L_2
// *Transformation*
Transformer la base de données en représentation verticale
CE_2 = Décomposition de L_2 en classes d'équivalences
// *Recherche*
Pour tout $E_2 \in CE_2$ **Faire**
    $Calcule\_Frequents(E_2)$ // *Voir algorithme 6*
Fin Pour
// *Fin*
retourner $\cup_k L_k$

Algorithme 6 $Calcule\_Frequent(E_{k-1})$

Entrée : E_{k-1} une classe d'équivalence
Sortie : Tous les itemsets fréquents de cette classe d'équivalence

Pour tout itemsets I_1 et I_2 dans E_{k-1} **Faire**
    Si $|I_1.ensSupport \cap I_2.ensSupport| \geq \varepsilon_a$ **Alors**
        $L_k \leftarrow L_k \cup \{I_1 \cup I_2\}$
    Fin Si
Fin Pour
CE_k = Décomposition de L_k en classes d'équivalences
Pour tout $E_k \in CE_k$ **Faire**
    $Calcule\_Frequents(E_k)$
Fin Pour

verticale peut être utilisée pour rechercher des itemsets fréquents fermés, ce qui augmente encore les performances. C'est la méthode employée dans le système Charm [Zaki et Hsiao 2002].

Un des problèmes de l'approche verticale est que si la base de données est très grande et très dense les ensembles-support peuvent occuper beaucoup de place en mémoire, et dans certains cas les classes d'équivalence ne sont plus une optimisation suffisante. Zaki a proposé dans [Zaki et Gouda 2003] de représenter les ensembles-support par des *diffsets*, c'est-à-dire stocker l'ensemble-support initial correspondant aux 1-itemsets, puis pour un $k+1$ itemset stocker uniquement les différences entre son ensemble-support et ceux des k-itemsets l'engendrant. Cela permet de réduire considérablement la taille mémoire nécessaire à leur stockage. Il a montré que cette approche permet d'améliorer significativement les algorithmes verticaux. A l'heure actuelle c'est une des méthodes de recherche d'itemsets fréquents les plus rapides.

2 Données structurées de type arbre

Dans le cadre de la découverte de motifs fréquents dans des données structurées, comme par exemple les documents XML, de nouveaux algorithmes ont vu le jour.

Nous allons présenter en détail plusieurs d'entre eux, qui utilisent chacun des définitions de l'inclusion d'arbre différentes (cf. chapitre II).

La figure III.6 est un exemple d'une base de données d'arbre $\mathcal{BA}$. Pour chaque algorithme adapté à ce type de données d'entrée, nous montrerons quels sont les plus gros arbres fréquents qu'il trouve sur cette base, avec une fréquence par occurences (déf. 13 du chapitre II) et un seuil absolu $\varepsilon_a = 2$.

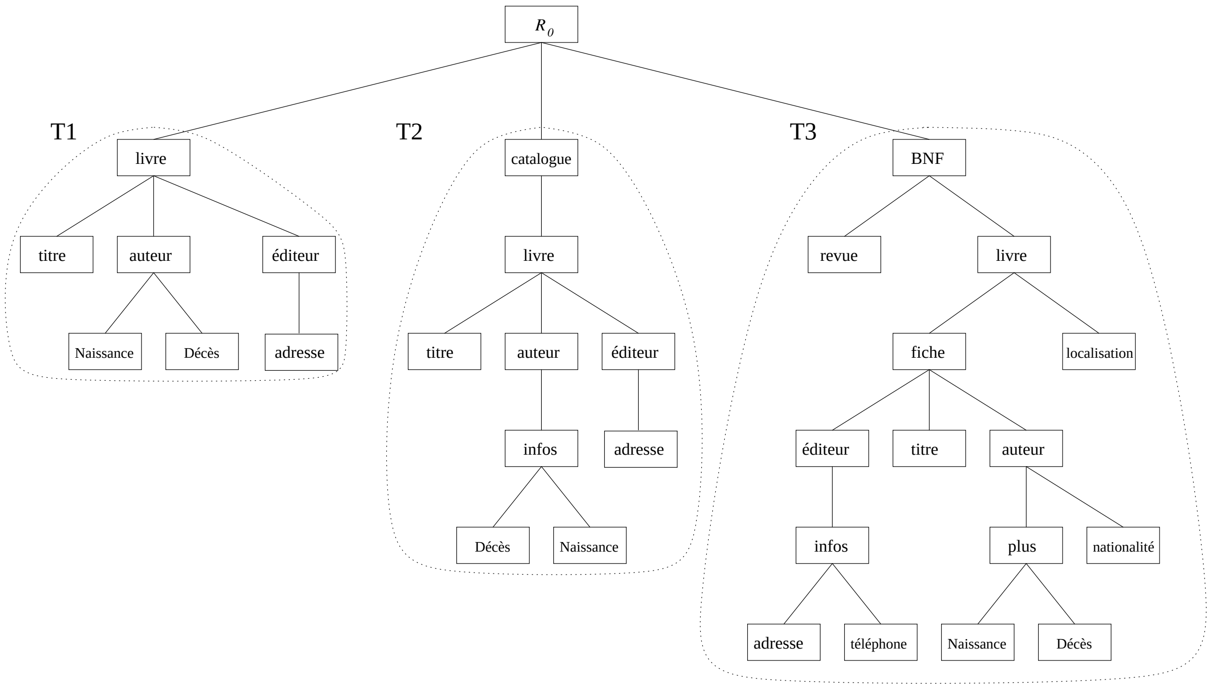


FIG. III.6 – Base de données d'arbres d'exemple

2.1 Recherche d'arbres fréquents ordonnés avec inclusion exacte

Dans [Asai et al. 2002],une méthode de recherche d'arbres fréquents adaptée à des documents semi-structurés simples est présentée. Leurs expérimentations portent sur des pages HTML mises sous forme d'arbres, produites par un moteur de recherche en réponse à une requête.

Ces documents sont stockés dans une base de données d'arbres. Les noeuds du datatree de cette base de données sont numérotés suivant l'ordre préfixe.

La définition d'inclusion d'arbre utilisée par Asai est l'inclusion stricte exacte ordonnée. Pour toute cette partie, nous noterons simplement cette inclusion par $\sqsubseteq$.

C'est la définition d'inclusion d'arbre la plus contrainte que nous ayons proposée dans le chapitre précédent. Le plus gros arbre fréquent que l'on trouve dans $\mathcal{BA}$, avec $\varepsilon_a = 2$, est l'arbre de la figure III.7.

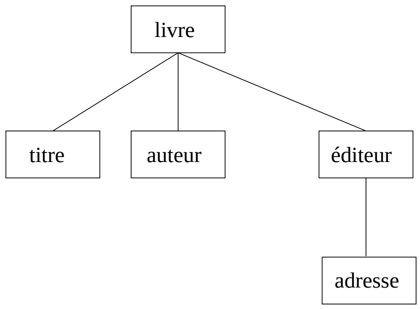


FIG. III.7 – Le plus gros arbre fréquent trouvé par l'approche d'Asai sur $\mathcal{BA}$

Les contraintes de la définition d'inclusion permettent d'optimiser l'algorithme pour limiter le nombre d'arbres fréquents candidats produits lors de la phase de génération. Le chemin le plus à droite des arbres va être distingué pour jouer un rôle important au cours de cette procédure de génération.

Le calcul des occurences d'un arbre candidat en particulier ne va pas se baser sur les occurences du noeud racine, mais sur les occurences de la feuille la plus à droite de cette arbre. Pour un arbre T, nous appelerons cette feuille $rml(T)$, par souci de cohérence avec les notations utilisées par Asai (rml = *Right Most Leaf* en anglais).

Pour un arbre T on définira donc $Roc(T)$ l'ensemble des occurences de $rml(T)$.

Définition 21 (Occurence de rml) *Soient T_1 et T_2 deux arbres tels que $T_1 \sqsubseteq T_2$. Soit μ un mapping de T_1 vers T_2. On appelera* ***occurence de la rml*** *de T_1 le noeud u de T_2 tel que $u = \mu(rml(T_1))$. L'ensemble des occurences de la rml de T_1 dans T_2 sera appelé $Roc(T_1, T_2)$ et défini comme suit :*

$$Roc(T_1, T_2) = \{u \in N_{T_2} \mid u = \mu(rml(T_1)) \wedge \mu \in \mathcal{EM}_{\sqsubseteq}(T_1, T_2)\}$$

L'algorithme de recherche d'arbres fréquents FIND-FREQ-TREES est défini dans l'algorithme 7. Cet algorithme a une structure très similaire à celle d'APRIORI : à la ligne 1, il effectue un balayage du datatree pour déterminer les occurences des différentes étiquettes. Les résultats sont stockés dans $\mathcal{F}_1$ et Roc_1. Les itérations suivantes sont en deux phases : premièrement, l'algorithme génère à la ligne 4 l'ensemble $\mathcal{C}_k$ des candidats de taille k et l'ensemble Roc_k de leurs listes d'occurences de rml grâce à la procédure EXPAND-TREES. Puis dans la boucle des lignes 5 à 10, l'algorithme calcule le support de chaque candidat à la ligne 6, et ajoute ce candidat à la liste des arbre fréquents (ligne 8) si son support dépasse le seuil ε_a.

Algorithme 7 FIND-FREQ-TREES

Entrée : Une base de données d'arbre $\mathcal{BA}$, un ensemble d'étiquettes E et un seuil de fréquence absolu ε_a

Sortie : L'ensemble $\mathcal{F}$ de tous les arbres ε_a-fréquents de $\mathcal{BA}$.

$\mathcal{F}_1 \leftarrow$ l'ensemble de tous arbres à 1 noeud ε_a-fréquents, et l'ensemble Roc_1 de leurs rml-occurences.
$k \leftarrow 2$
Tant que $\mathcal{F}_{k-1} \neq \emptyset$ **Faire**
  $(C_k, Roc_k) \leftarrow ExpandTrees(\mathcal{F}_{k-1}, Roc_{k-1})$
  Pour tout arbre candidat $T \in C_k$ **Faire**
    Calculer $S_o(T)$ à partir de $Roc_k(T)$
    Si $S_o(T) \geq \varepsilon_a$ **Alors**
      $\mathcal{F}_k \leftarrow \mathcal{F}_k \cup \{T\}$
    Fin Si
  Fin Pour
  $k \leftarrow k+1$
Fin Tant que
retourner $\mathcal{F} = \mathcal{F}_1 \cup ... \cup \mathcal{F}_k$

La phase la plus délicate de l'algorithme est la génération d'arbres candidats. Pour étendre un arbre de k noeuds, on peut lui ajouter un nouveau noeud, fils de l'un de ces k noeuds. Si l'ensemble des étiquettes est de taille n, cela nous donne donc $k \times n$ nouveaux arbres possibles. Si n est grand et si les arbres recherchés ont beaucoup de noeuds, cette technique va générer beaucoup de candidats et être très inefficace. Pire, au bout de quelques itérations elle va générer beaucoup de doublons, car en partant d'un noeud initial et en lui ajoutant des noeuds un par un, il y a plusieurs manières d'arriver à un arbre de taille k, comme illutré sur la figure III.8.

La solution proposée par Asai est d'utiliser une forme restreinte d'expansion des arbres appelée *expansion par la droite*. Soit *la branche la plus à droite* d'un arbre T l'unique chemin entre $racine(T)$ et $rml(T)$. L'expansion par la droite ne peut ajouter des noeuds à T qu'en ajoutant des fils aux noeuds de la branche la plus à droite de T.

Définition 22 (**(p,l)-extension**) *Soit T un arbre étiqueté ordonné, $p \in [0, profondeur(T)[$ un entier, et $l \in E$ une étiquette. Soit $x = rml(T)$, et $y = parent^p(x)$ le p-ième parent de x. Alors la (p,l)-extension S de T est l'arbre étiqueté ordonné obtenu en attachant un nouveau fils z d'étiquette l au noeud y tel que z soit le fils le plus à droite de y.*

Cette définition est illustrée sur la figure III.9. Une extension par la droite d'un arbre T est une (p,l)-extension S de T pour un entier $p \geq 0$ et une étiquette $l \in E$. On dit alors que S est le *successeur* de T ou que T est le *prédecesseur* de S. Il existe un arbre spécial, appelé l'*arbre vide*, noté $\perp$, qui ne contient aucun noeud et tel que tout arbre à un seul noeud est un successeur de $\perp$.

Asai a montré dans son article qu'en générant des arbres candidats par extension par la droite, on ne générait jamais deux fois le même arbre.

La procédure de génération d'arbres candidats EXPAND-TREES est donnée dans l'algorithme 8. Pour créer des arbres candidats de taille k, elle consiste simplement à générer toutes les (p,l)-extensions des arbres fréquents de taille $k-1$.

Pour chacun des candidats générés, il faut calculer le *Roc* correspondant. Le *Roc* stocke les mappings entre un arbre T et le datatree D, en ne gardant pour représenter chaque mapping μ

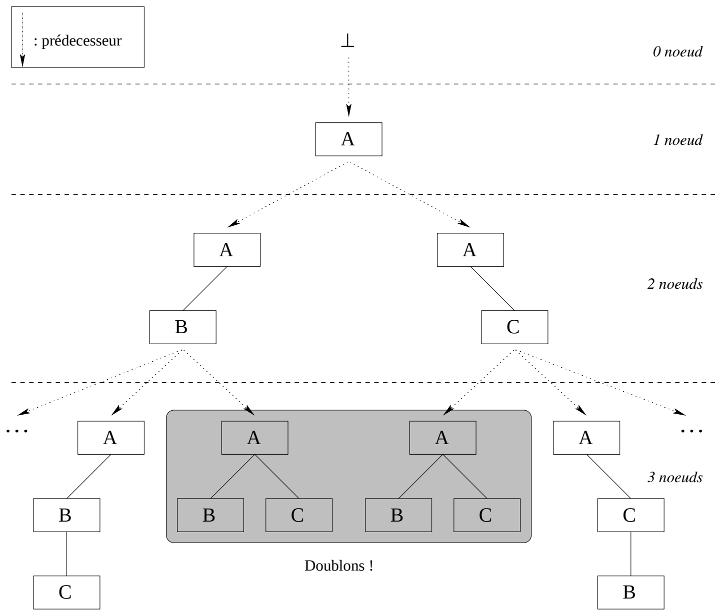


Fig. III.8 – Un exemple de génération incrémentale d'arbres

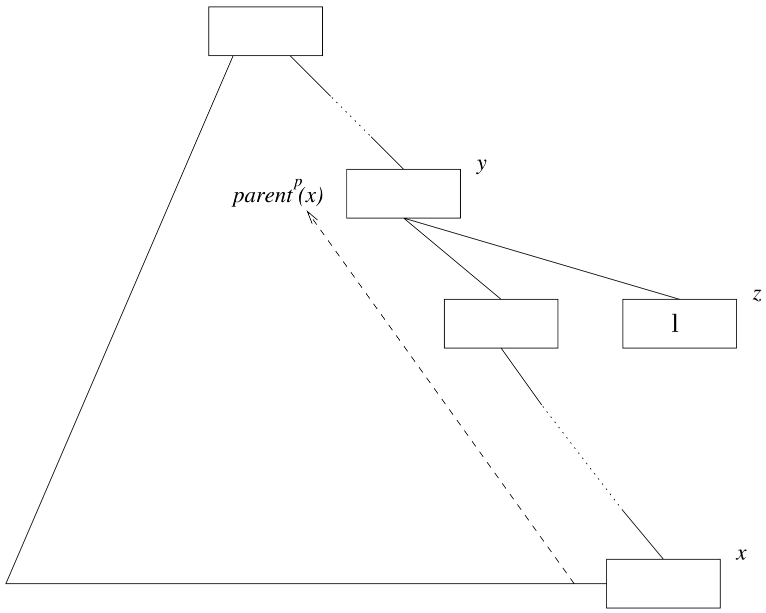


Fig. III.9 – (p, l)-extension

Algorithme 8 EXPAND-TREES($\mathcal{F}_{k-1}, Roc_{k-1}$)

Entrée : Un ensemble $\mathcal{F}_{k-1}$ d'arbres fréquents à $k-1$ noeuds, et l'ensemble Roc_{k-1} de leurs rml-occurrences.

Sortie : L'ensemble $\mathcal{C}_k$ des extensions par la droite des arbres de $\mathcal{F}_{k-1}$, et l'ensemble Roc_k de leurs rml-occurrences.

```
C_k ← ∅
Roc_k ← ∅
Pour tout arbre T ∈ F_{k-1} Faire
   Pour tout position 0 ≤ p < profondeur(rml(T)) Faire
      Pour tout étiquette l ∈ E Faire
         Calculer la (p, l)-extension S de T
         Roc_k(S) ← UPDATE-ROC(Roc_{k-1}, p, l)
         F_k ← F_k ∪ {S}
      Fin Pour
   Fin Pour
Fin Pour
retourner (F_k, Roc_k)
```

que l'image de $rml(T)$ par μ. Le lemme suivant permet de calculer incrémentalement le *Roc* d'une (p, l)-extension d'un arbre T à partir du *Roc* de T.

Lemme 1 *Soit T un arbre fréquent à $k-1$ noeuds et D un datatree tels que $T \sqsubseteq D$. Soit μ un mapping entre T et D. Soit T' une (p, l)-extension de T, alors la fonction ν satisfaisant les trois conditions suivantes est un mapping entre T' et D.*

1. *ν est une extension de μ en tant que mapping, c'est à dire que $\forall i \in [1, k-1]$ $\nu(i) = \mu(i)$*
2. *$\nu(k)$ est un fils de $parent_D(w)$, et si $p \neq 0$ alors $w \prec \nu(k)$, avec $w = parent_D^{p-1}(\mu(k-1))$*
3. *$\varphi_D(\nu(k)) = l$*

L'algorithme de mise à jour des *Roc*, UPDATE-ROC, se déduit de ce lemme, en particulier de la 2ème propriété que doit vérifier ν. Son pseudo-code est présenté dans l'algorithme 9.

2.2 Recherche d'arbres fréquents ordonnés avec préservation des ancêtres

L'approche précédente a le défaut d'utiliser une définition de l'inclusion très contrainte. Cela signifie que cette approche est restreinte à l'analyse de documents structurés ayant des structures très proches. La moindre variation dans l'imbrication des noeuds ne permettra plus de découvrir des arbres fréquents.

Pour pouvoir trouver des arbres fréquents plus généraux, Zaki a proposé dans [Zaki 2002] une technique utilisant l'inclusion stricte ordonnée. La contrainte de préservation des parents étant relâchée, on peut découvrir des arbres fréquents dont les noeuds n'ont pas le même niveau d'imbrication dans leurs différentes occurences.

On peut alors trouver des arbres plus gros que ceux trouvés par l'approche précédente sur une même base de données d'arbres, comme l'illustre la figure III.10. Notons que la préservation de l'ordre empêche de découvrir que le noeud *auteur* a simultanément les fils *Naissance* et *Décès*.

Zaki propose, comme Asai, une approche basée sur le même schéma "générer et tester" que les approches de type APRIORI. La représentation et la génération des candidats est donc là encore la principale originalité de l'approche.

Algorithme 9 UPDATE-ROC(Roc_{k-1}, p, l)

Entrée : Une liste de noeuds Roc_{k-1}, un entier $p \geq 0$ et une étiquette $l \in E$.
Sortie : Une liste de noeuds Roc_k

$Roc_k \leftarrow \emptyset$
Pour tout $u \in Roc_{k-1}$ **Faire**
  Si $p = 0$ **Alors**
    $v \leftarrow premier\_fils(u)$
  Sinon
    $courant \leftarrow parent_D^{p-1}(u)$
    $v \leftarrow frere\_droit\_immediat(courant)$
  Fin Si
  Tant que $v \neq NIL$ **Faire**
    Si $\varphi_D(v) = l$ **Alors**
      $Roc_k \leftarrow Roc_k \cup \{v\}$
    Fin Si
    $v \leftarrow frere\_droit\_immediat(v)$
  Fin Tant que
Fin Pour
retourner Roc_k

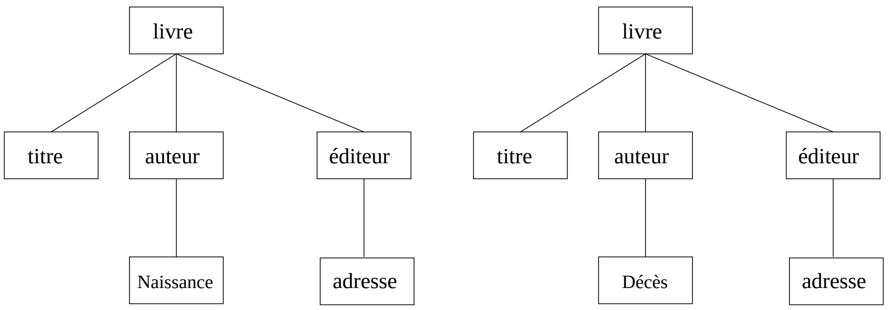


FIG. III.10 – Les plus gros arbres fréquents trouvés par l'approche de Zaki sur $\mathcal{BA}$

Afin d'avoir la représentation la plus efficace possible, Zaki a choisi de représenter ses arbres par des chaînes de caractères. En effet, ces structures de données sont bien connues, et il existe de nombreux algorithmes optimisés permettant de les traiter.

Représentation d'un arbre par une chaîne de caractères : La représentation $\mathcal{T}$ d'un arbre T en chaîne de caractères est obtenue comme suit : effectuer un parcours en profondeur de l'arbre T. Pour chaque noeud rencontré, ajouter son étiquette x à $\mathcal{T}$. À chaque fois que l'on remonte d'un enfant à son parent, ajouter -1 à $\mathcal{T}$ (on supposera que $-1 \notin E$).

Exemple : La figure III.11 montre un arbre et sa représentation en chaîne de caractères.

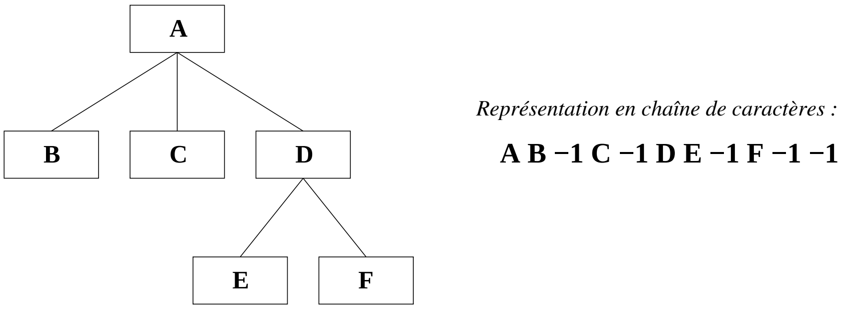


Fig. III.11 – Représentation d'un arbre par une chaîne de caractères

Cette représentation permet à Zaki de créer des classes d'équivalences d'arbres candidats, une méthode qu'il avait déjà employée avec succès dans l'algorithme Eclat.

Les classes d'équivalences considérées sont les classes des arbres à k noeuds dont la représentation en chaîne de caractères est la même jusqu'au $(k-1)$-ème noeud. Tous les arbres d'une classe d'équivalence partagent le même préfixe $\mathcal{P}$ sur les $k-1$ premiers noeuds de leur représentation en chaîne de caractère, et donc possèdent tous le même sous-arbre préfixe P de représentation $\mathcal{P}$. Une classe correspondant aux arbres de préfixe P est notée $[P]$. Un exemple de classe d'équivalence est donné sur la figure III.12.

Cette figure montre également le format de stockage des classes d'équivalence utilisé par Zaki. Chaque classe est représentée par son préfixe et par la liste des éléments qui peuvent être ajoutés à ce préfixe. Chaque élément est représenté par une paire (e, pos) où e est une étiquette et où pos est le numéro en ordre préfixe du noeud où attacher un nouveau noeud d'étiquette e.

Remarque 4 *Les seules positions valides pour ajouter un noeud et rester dans la même classe d'équivalence sont les noeuds de la branche droite. Sinon le préfixe change et donc la classe aussi. On peut voir ici un parallèle avec la méthode d'Asai : les extensions se feront toujours par la branche droite.*

La génération de candidats proprement dite se fait par une jointure deux à deux entre tous les éléments de chaque classe d'équivalence.

Définition 23 (Jointure) *Soit $[P]$ une classe d'équivalence dont la représentation du préfixe est $\mathcal{P}$. Soient (x, i) et (y, j) deux éléments quelconques de cette classe, $\mathcal{PX}$ et $\mathcal{PY}$ les représentations en chaînes de caractères de ces éléments. L'opération de jointure $\otimes$ de ces deux éléments, notée $(x, i) \otimes (y, j)$ est définie comme suit :*

1. *$i < j$: impossible de créer de nouveaux candidats dans ce cas, on perdrait le préfixe $\mathcal{PX}$*
2. *$i \geq j$: Créer une nouvelle classe $[Px]$, de préfixe $\mathcal{PX}$*

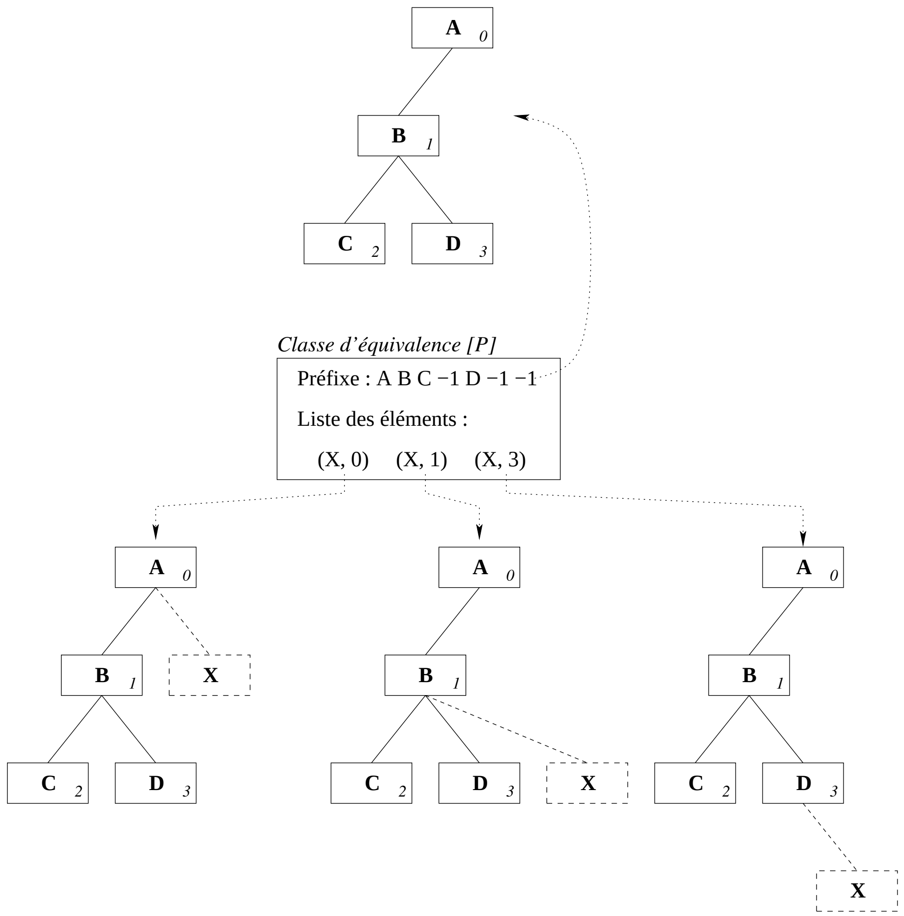


FIG. III.12 – Une classe d'équivalence pour un préfixe P de représentation A B C -1 D -1 -1

(a) $i = j$ *: si* $\mathcal{P} \neq \emptyset$*, ajouter à la classe* $[Px]$ *les éléments* (y, j) *et* $(y, j+1)$
Sinon, ajouter $(y, j+1)$ *à la classe* $[Px]$

(b) $i > j$ *: ajouter l'élément* (y, j) *à la classe* $[Px]$

Zaki a montré que par cette jointure sur tous les éléments d'une classe $[P]$ d'arbres à k noeuds on énumérait correctement et sans doublons tous les arbres à $k+1$ noeuds ayant le préfixe P.

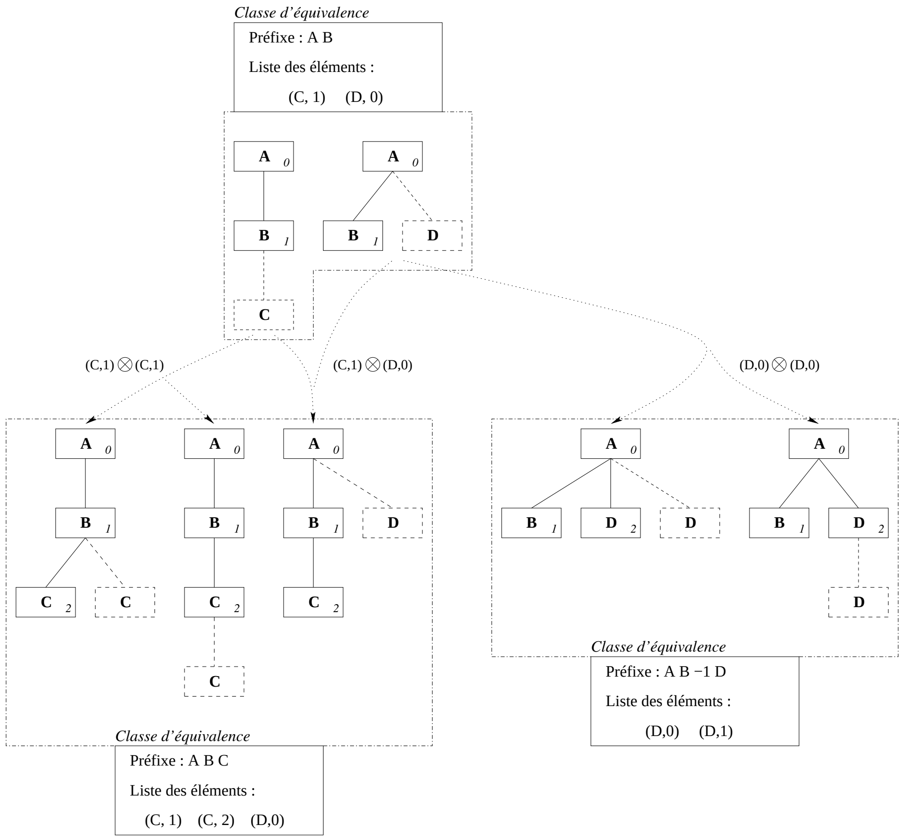


FIG. III.13 – Extension d'une classe d'équivalence simple

A partir de cette méthode de génération, on peut présenter l'algorithme de recherche d'arbres fréquents TREEMINERH, qui reprend la même structure qu'APRIORI. Son pseudo-code est donné dans l'algorithme 10.

Les lignes 1 et 2 sont des calculs optimisés des ensembles $\mathcal{F}_1$ et $\mathcal{F}_2$. La phase itérative de l'algorithme a lieu des lignes 3 à 10. À la ligne 4, les classe d'équivalence candidates sont générées à partir des classes d'équivalence précédentes, en utilisant la méthode de jointure définie plus haut. Les lignes 5 à 8 servent à évaluer le support par document de chaque candidat. Dans le datatree, on considère chaque document D_i (c'est un sous-arbre immédiat de la racine). Pour ce document, on ne considère que les classes d'équivalence $[P]_{k-1}$ dont le préfixe est inclus dans D_i. Pour ces classes, on détermine s'il y a inclusion de chaque élément de la classe dans D_i.

Déterminer l'inclusion entre un préfixe P et un arbre T est la partie la plus complexe du comptage du support. Soit $P[i]$ le i-ème noeud du sous-arbre P. Le test est effectué de manière récursive. Supposons qu'à l'étape r on aie déterminé que les noeuds $P[0], ..., P[r]$ correspondent aux noeuds $T[i_0], ..., T[i_r]$. Notons que si les noeuds $P[0], ..., P[r]$ sont traversés consécutivement par une traversée en profondeur d'abord de P, les noeuds $T[i_0], ..., T[i_r]$ peuvent eux être assez éloignés (cela est

Algorithme 10 TreeMinerH

Entrée : Un datatree D et un seuil de support relatif ε_r
Sortie : $\mathcal{F}$ l'ensemble de tous les arbres fréquents

```
F_1 ← { arbres fréquents à 1 noeud (1-arbres) }
F_2 ← { classes d'équivalences des 2-arbres fréquents (préfixes de longueur 1) }
Pour tout (k = 3 ; F_{k-1} ≠ ∅ ; k++) Faire
    C_k ← { classes d'équivalence [P]_{k-1} de- k-arbres candidats }
    Pour tout sous-arbre immédiat D_i de D Faire
        // pour chaque document d_i
        Augmenter le support par document de tout S ⊑ D_i, S ∈ [P]_{k-1}
    Fin Pour
    F_k ← { classes d'équivalences des k-arbres fréquents }
Fin Pour
retourner F = ⋃_k F_k
```

dû au choix de l'ancestralité plutôt que de la parenté dans la définition d'inclusion retenue). Si le noeud suivant recherché $P[r+1]$ est un fils de $P[r]$, alors on cherche son correspondant $T[i_{r+1}]$ dans le sous arbre de racine $T[i_r]$. Par contre, si $P[r+1]$ n'est pas un descendant de $P[r]$, mais est au contraire attaché à une position l de P, avec $0 \leq l < r$, alors on cherche son correspondant $T[i_{r+1}]$ dans le sous-arbre de racine $T[i_l]$. Cette opération est répétée jusqu'à ce que tous les noeuds de P aient trouvé leur correspondant dans T.

Si l'opération réussi et donc que $P \sqsubseteq T$, alors pour chaque élément $(x,k) \in [P]$ on cherche s'il a un correspondant dans T. Ce correspondant est recherché dans $T[i_k]$. Si un correspondant est trouvé le support de (x,k) est incrémenté, de 1 à chaque fois si l'on recherche un support par occurences, et sinon une fois seulement pour chaque arbre T représentant un document.

Une fois le support calculé, il est facile de déterminer les candidats qui sont des arbres fréquents.

De part sa proximité avec Apriori, TreeMinerH hérite des défauts de cette approche, en particulier la difficulté de trouver des arbres fréquents de grande taille. C'est pourquoi, tout comme Zaki a amélioré Apriori par une approche verticale appelée Eclat, il a aussi amélioré TreeMinerH en proposant une version verticale, TreeMinerV.

La représentation verticale est particulièrement adaptée au calcul des ensembles support des documents arbres. Pour la suite, on considèrera donc que le datatree D est divisé en n documents $D_1, ...D_n$, qui sont les sous-arbres immédiats de la racine de D. L'identifiant d'un arbre D_i sera donc i.

Pour présenter la représentation verticale d'un arbre T, nous avons besoin de la définition de la *portée* d'un noeud.

Définition 24 (Portée d'un noeud) *Soit T un arbre, soit x un noeud de T, et soit i le numéro préfixe de x. Alors le couple $[i,j]$ est appelée* ***portée*** *de x, où j est le numéro préfixe de la feuille la plus à droite du sous-arbre de T de racine x.*

Représentation verticale : Soit T un arbre, k un entier et X un k-sous-arbre de T. Soit x_k la feuille la plus à droite de X. On note par $\mathcal{L}(X)$ la *liste de portée* de X, c'est-à-dire sa représentation verticale. Chaque élément de cette liste de portée est un triplet (t,s,μ), où t est l'identifiant de l'arbre T, s est la portée de x_k, et μ est un mapping entre X et T.

La figure III.14 montre la représentation verticale pour un arbre simple et des sous-arbres réduits à un seul noeud.

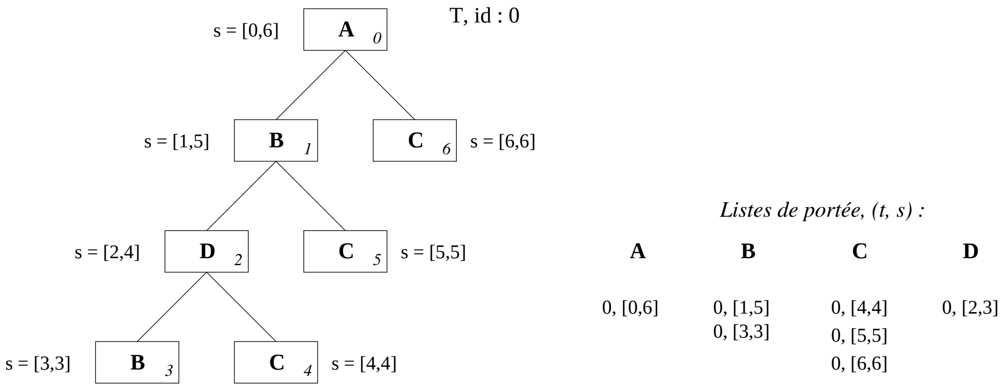


FIG. III.14 – Représentation verticale pour un arbre simple

L'algorithme 11 montre le schéma général de TREEMINERV. Le calcul des classes d'équivalences des arbres fréquents de taille 1 et 2 est fait de la même manière que pour TREEMINERH. À la ligne 3, l'algorithme calcule la représentation verticale des classes d'équivalences de $\mathcal{F}_2$, pour pouvoir à la ligne 4 effectuer le calcul des arbres fréquents de taille supérieure ou égale à 3.

Algorithme 11 TREEMINERV

Entrée : Un datatree D et un seuil de support relatif ε_r
Sortie : $\mathcal{F}$ l'ensemble de tous les arbres fréquents
1: $\mathcal{F}_1 \leftarrow$ { arbres fréquents à 1 noeud (1-arbres) }
2: $\mathcal{F}_2 \leftarrow$ { classes d'équivalences des 2-arbres fréquents (préfixes de longueur 1) }
3: **Pour tout** $[P]_1 \in \mathcal{F}_2$ **Faire**
4: $\quad Enumerate\_Frequent\_Subtrees([P]_1)$
5: **Fin Pour**

Algorithme 12 $Enumerate\_Frequent\_Subtrees([P])$

Pour tout élément $(x,i) \in [P]$ **Faire**
$\quad [P_x] \leftarrow \emptyset$
$\quad$ **Pour tout** élément $(y,j) \in [P]$ **Faire**
$\quad\quad R \leftarrow (x,i) \otimes (y,j)$
$\quad\quad \mathcal{L}(R) \leftarrow \mathcal{L}(x) \cap_{\otimes} \mathcal{L}(y)$
$\quad\quad$ **Pour tout** $r \in R$ tq r est fréquent **Faire**
$\quad\quad\quad [P_x] \leftarrow [P_x] \cup \{r\}$
$\quad\quad$ **Fin Pour**
$\quad$ **Fin Pour**
$\quad Enumerate\_Frequent\_Subtrees([P_x])$
Fin Pour

La procédure *Enumerate_Frequent_Subtrees* permet de calculer les candidats de taille $k+1$, puis d'effectuer l'intersection des listes de portées de ces candidats afin de déterminer s'ils sont fréquents.

L'intersection de deux listes de portées $\mathcal{L}(x)$ et $\mathcal{L}(y)$ se déduit de la jointure que nous avons vue précédemment. Deux cas se distinguaient lorque l'on effectuait la jointure $(x,i) \otimes (y,j)$: on ajoutait à $[Px]$ l'élément $(y, j+1)$ ou l'élément (y,j).

Premier cas : L'élément $(y, j+1)$ n'est ajouté à $[Px]$ que si $i = j$, donc quand le noeud y sera un fil de x dans la nouvelle classe $[Px]$. Le sous-arbre de racine y doit donc être inclus dans le sous-arbre de racine x. Donc il suffit de trouver $(t_x, s_x, \mu_x) \in \mathcal{L}(x)$ et $(t_y, s_y, \mu_y) \in \mathcal{L}(y)$ tels que :

1. $t_x = t_y$
2. $s_y \subseteq s_x$, c-à-d. si $s_y = [a_y, b_y]$ et $s_x = [a_x, b_x]$, on a $a_x \leq a_y$ et $b_x \geq b_y$
3. $\mu_x = \mu_y$

Si ces trois conditions sont satisfaites, on ajoute le triplet $(t_y, s_y, \{\mu_y \cup (x \to a_x)\})$ à la liste de portée de $(y, j+1)$ dans $[Px]$.

Deuxième cas : l'élément (y, j) est ajouté à $[Px]$ si x et y sont descendants d'un noeud à une position j dans P, et le y n'est pas dans le sous-arbre de x mais plus à droite. Donc il suffit de trouver $(t_x, s_x, \mu_x) \in \mathcal{L}(x)$ et $(t_y, s_y, \mu_y) \in \mathcal{L}(y)$ tels que :

1. $t_x = t_y$
2. si $s_y = [a_y, b_y]$ et $s_x = [a_x, b_x]$, alors $b_x < a_y$
3. $\mu_x = \mu_y$

Si ces trois conditions sont satisfaites, alors on ajoute le triplet $(t_y, s_y, \{\mu_y \cup (x \to a_x)\})$ à la liste de portée de l'élément (y, j) dans $[Px]$.

Les résultats expérimentaux ont montré que TreeMinerV était beaucoup plus rapide que TreeMinerH, surtout sur si les données contiennent des arbres fréquents de grande taille. La différence de performances est du même ordre que la différence de performances entre Apriori et Eclat.

2.3 Recherche d'arbres fréquents sans préserver l'ordre

Les approches précédentes ont des applications où l'ordre est important. Mais pour certaines données de type arbre, la préservation de l'ordre empêche de découvrir des arbres fréquents intéressants.

Yang et al. ont présenté très récemment [Yang et al. 2003b, Yang et al. 2003a] un algorithme de recherche d'arbres fréquents adapté aux "query patterns". Leur démarche est plus proche du domaine des bases de données que de la fouille de données. Avec la quantité croissante de documents semi-structurés, des entrepôts adéquats ont été créés pour stocker ces documents. Pour interroger le contenu de ces entrepôts, des langages de requêtes spécialisés dans des données de type arbre ont été créés, comme XQuery [XQY]. A cause de la structure, ces requêtes sont plus difficiles à traiter que des requêtes SQL traditionnelles. Afin d'accélérer la vitesse moyenne de réponse d'un entrepôt aux requêtes, Yang et al. ont eu l'idée de rechercher les requêtes fréquentes, afin de stocker les réponses à ces requêtes dans un cache.

Le problème qu'ils ont à traiter est donc le suivant : étant donné un ensemble de requêtes assimilables à un ensemble d'arbres Q et un seuil de fréquence relatif ε_f, déterminer les sous-arbres fréquents de Q.

Les arbres considérés par cette approche sont des *arbres de requête*, définis comme suit :

Définition 25 (Arbre de requête) *Soit E un ensemble d'étiquettes. Un arbre de requête T est défini par $(N_T, A_T, racine(T), \varphi, \prec)$, où T est un arbre étiqueté ordonné, et où φ est une fonction d'étiquetage définie comme suit : $\varphi : N_T \mapsto E \cup \{*, //\}$.*

– L'étiquette "$*$" signifie que le noeud qui porte cette étiquette peut être "mappé" avec un noeud ayant une étiquette quelconque .

– L'étiquette "//" signifie que le noeud qui porte cette étiquette peut être remplacé par un chemin de noeuds à étiquettes dans E (éventuellement de longueur nulle).

L'utilisation de ces étiquettes dans les requêtes est montré dans les figures III.15 et III.16.

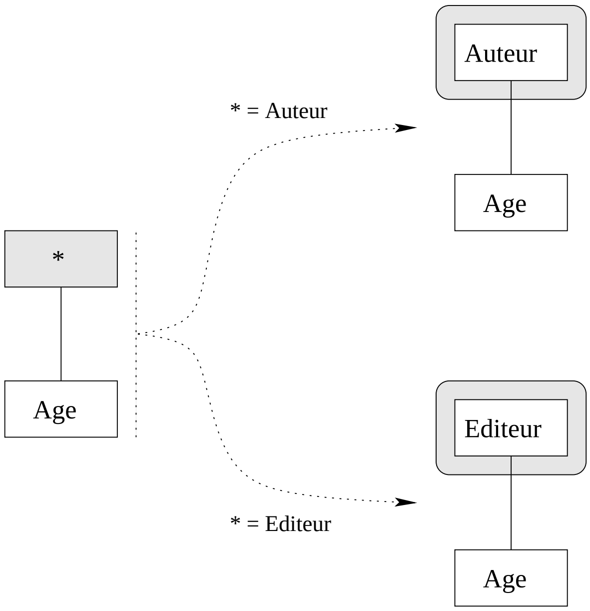


Fig. III.15 – Utilisation de $*$

La figure III.15 montre qu'une requête contenant une $*$ va correspondre à plusieurs documents ayant la même forme, au noeud étoilé près.

La figure III.16 montre qu'une requête contenant un // permet de découvrir des documents similaires à l'exception d'un chemin.

Les arbres fréquents recherchés doivent être des sous-arbres d'arbres représentant des requêtes. Pour gérer le cas des étiquettes spéciales, Yang et al. ont introduit un ordre entre les étiquettes. Soit $e \in E$ une étiquette quelconque de l'ensemble des étiquettes. On a :

– $e \leq e$
– $* \leq *$
– $// \leq //$
– $e \leq * \leq //$

La définition d'inclusion d'arbre utilisée est une inclusion exacte, utilisant l'ordre des étiquettes $\leq$ que nous venons juste de définir.

Cela signifie que lorsque l'on cherche l'inclusion entre un arbre de requête T_1 et un arbre de requête T_2, un noeud de T_1 à étiquettes dans E peut correspondre à un noeud de T_2 ayant soit une étiquette de E, soit l'étiquette $*$, soit l'étiquette //. Par contre, un noeud de T_1 ayant l'étiquette $*$ ne peut correspondre qu'à un noeud de T_2 ayant soit l'étiquette $*$, soit l'étiquette //. Enfin, un noeud de T_1 d'étiquette // ne peut correspondre qu'à un noeud de T_2 d'étiquette //.

On peut remarquer que les changements de la définition d'inclusion n'impliquent pas de gros changements dans la génération de candidats par rapport au problème traité par Asai. Effectivement, l'algorithme proposé par Yang et al., XQPMiner, est très proche de l'algorithme Find-Freq-Trees proposé par Asai. La génération de candidats par expansion de la branche la plus à droite est très proche dans les deux algorithmes. Toutefois, l'algorithme de Yang et al. utilise le fait que toutes les requêtes proviennent du même schéma, qui est connu : l'expansion par la droite est guidée par

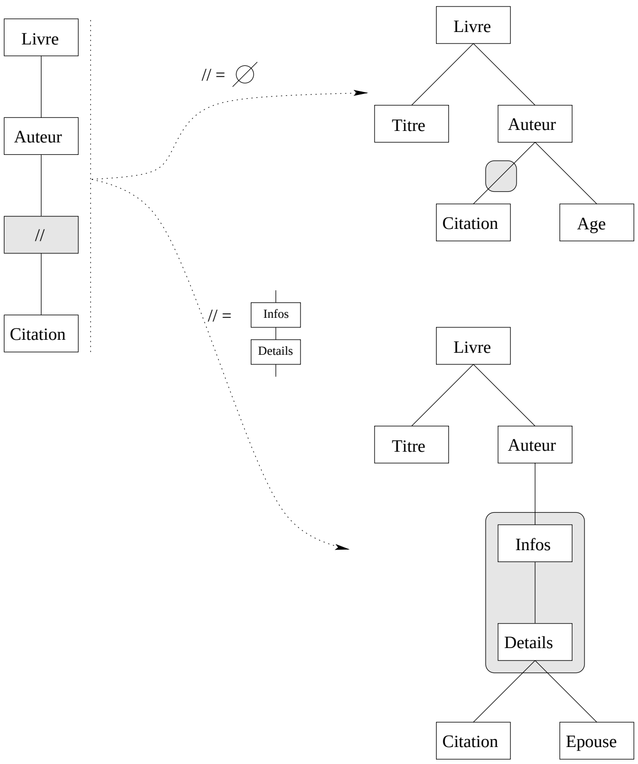


FIG. III.16 – Utilisation de //

ce schéma, ce qui permet de réduire le nombre de candidats générés. La version présentée dans [Yang et al. 2003a] utilise de plus des classes d'équivalences fondées sur le modèle de celles de Zaki pour accélérer encore la génération de candidats. Le test d'inclusion d'arbres prend en compte les spécificités de l'inclusion d'arbre considérée : mêmes racines, étiquettes spéciales et pas d'ordre. C'est par ce test que les algorithmes de Yang et al. se différencient le plus de ceux de Asai et Zaki.

Une autre approche est celle choisie par Cong et al. [Cong et al. 2002] . Leur but est de trouver des arbres fréquents dans une collection de documents XML. Leurs documents ne contiennent que des étiquettes de E, mais pour plus de généralité, les arbres fréquents découverts peuvent contenir les étiquettes spéciales $*$ et //, ces étiquettes spéciales ayant la même signification que précédemment.

La technique retenue par Cong et al. pour avoir un algorithme efficace est de décomposer les arbres en chemins. Leur algorithme cherche ensuite des ensembles de chemins fréquents, avec un algorithme très proche de l'algorithme APRIORI de base. Ces ensembles de chemins fréquents forment des sous-arbres fréquents. Les étiquettes spéciales $*$ et // peuvent être placées au début d'un chemin, ce qui permet de retrouver le même sous-arbre à des profondeurs différentes selon les documents, et avec des ancêtres différents. La définition d'inclusion utilisée est une inclusion stricte exacte, avec une gestion particulière des étiquettes $*$ et // : les sorties de l'algorithme sont des arbres inclus par une inclusion stricte exacte dans le datatree, et les arbres de cette sorte qui apparaissent fréquemment ensemble sont rattachés sous une racine commune, si nécessaire par l'intermédiaire d'un noeud ayant une étiquette spéciale $*$ ou //.

La figure III.17 montre le plus gros arbre trouvé par cette approche. Le noeud d'étiquette // indique que le sous-arbre qui suit est trouvé à plusieurs profondeurs différentes.

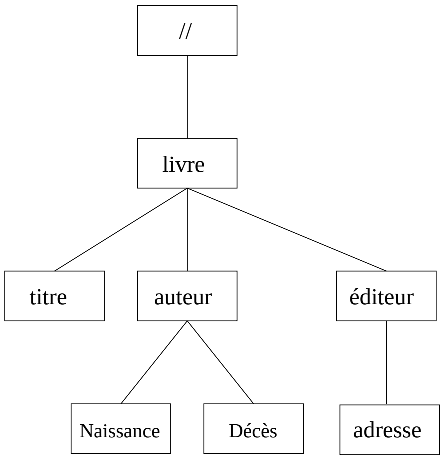


Fig. III.17 – Le plus gros arbre fréquent trouvé par l'approche de Cong et al. sur $\mathcal{BA}$

2.4 Lien avec les travaux sur l'inférence d'arbres

La recherche d'arbres fréquents est un problème proche mais différent du problème de l'inférence de structures d'arbres sur lequel un certain nombre de travaux ont vu le jour dans la lignée des travaux sur l'inférence grammaticale dont l'article fondateur est [Gold 1967]. Ces travaux relèvent de l'apprentissage automatique supervisé et visent à construire, à partir d'un ensemble d'arbres fournis comme exemples positifs, un automate d'arbres permettant de reconnaitre ces exemples. Même si les travaux les plus récents [Kosala et al.] considèrent des arbres d'arité quelconque, la plupart des travaux sont des extensions directes de l'inférence grammaticale sur des mots au cas des arbres d'arité bornée [Carme et al.]. Dans la communauté bases de données, des travaux récents (e.g., [Papakonstantinou et Vianu]) se sont intéressés à l'inférence de DTDs à partir d'un ensemble de documents XML. Ils montrent les limites des DTDs pour représenter des grammaires d'arbres suffisamment expressives et proposent des extensions du langage des DTDs pour lesquelles ils fournissent des algorithmes d'inférence de DTDs étendus. Les DTDs étendus, comme les automates d'arbres, sont des représentations de structure d'arbres. Cependant, ces représentations sont à la fois plus abstraites et moins flexibles que les arbres fréquents qu'il nous intéresse de découvrir dans cette thèse. En effet, le but de l'inférence d'automates d'arbres (ou de DTDs) est de capturer dans un automate (ou un DTD) l'ensemble de toutes les structures d'arbres fournies en entrée. S'il y a une trop grande hétérogénéité de structure dans ces arbres, l'automate (ou le DTD) obtenu sera très général et peu informant. Par construction, la découverte d'arbres fréquents est au contraire bien adaptée à des données hétérogènes. De fait, la découverte d'arbres fréquents permet d'expliciter et de caractériser des clusters homogènes d'arbres. Par ailleurs, le seuil de fréquence permet de traiter des données bruitées et d'en éliminer le bruit.

Chapitre IV

L'algorithme TreeFinder

Les méthodes de découverte d'arbres fréquents que nous avons vues dans le chapitre précédent exigent toujours que les arbres soient ordonnés. Notre objectif tout au long de ce mémoire va être de présenter des méthodes permettant de dépasser cette limitation.

Le premier algorithme que nous avons proposé dans ce sens est l'algorithme TreeFinder [Termier et al. 2002][1].

Cet algorithme prend en entrée une base de données d'arbres étiquetés $\mathcal{BA} = (D, \delta, \sigma)$ dont les sous-arbres fils de D sont les arbres de $\mathcal{T} = \{T_1, ..., T_n\}$ et un seuil de fréquence ε. Il renvoie un ensemble d'arbres ε-fréquents dans $\mathcal{BA}$, en utilisant la fréquence par identifiants.

L'algorithme TreeFinder utilise la définition d'inclusion stricte faible (définition 9, chapitre II) pour découvrir des arbres fréquents, qui ne préserve pas l'ordre des fils, et qui préserve la relation d'ancestralité plutôt que la relation de parenté.

Lorsque nous l'avons conçu, une de nos motivations était d'avoir un algorithme se fondant sur des outils bien connus et potentiellement efficaces, ceci afin que les performances de notre algorithme puissent évoluer avec les améliorations de ces outils.

Dans TreeFinder, la recherche d'arbres fréquents est guidée par une première étape de clustering : les arbres fréquents sont en fait les **arbres communs** inclus dans tous les arbres d'un cluster.

Pour l'étape de clustering, TreeFinder est guidé par la recherche d'occurrences fréquentes selon un certain seuil de paires d'étiquettes pour des nœuds en relation d'ancestralité. Les clusters sont les ensembles d'arbres dans lesquels on retrouve les mêmes occurrences de paires d'étiquettes. Cette étape de clustering est effectuée en appliquant un algorithme de recherche d'itemsets fréquents maximaux au codage transactionnel des arbres, où chaque arbre est représenté par l'ensemble des paires d'étiquettes correspondant aux nœuds en relation d'ancestralité.

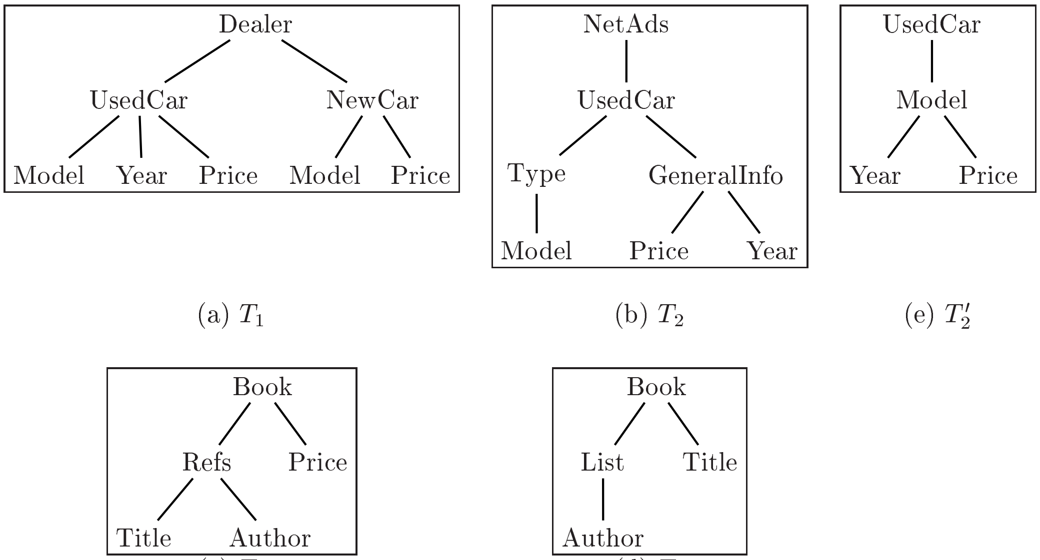
Pour l'extraction d'arbres fréquents, l'algorithme calcule les arbres communs à tous les arbres regroupés dans chaque cluster. Pour cela, TreeFinder utilise une méthode empruntée à la programmation logique inductive, pour calculer *Least General Generalization (LGG)*, sur une représentation adaptée des arbres de chaque cluster.

[1] par souci de compatibilé avec cet article, nous conserverons tels quels la plupart des exemples et notations, bien que ceux-ci soient en anglais.

Le plan de ce chapitre est le suivant :

– Nous allons d'abord détailler les deux étapes que nous venons d'introduire
– Puis nous caractériserons les résultats de TreeFinder. Nous montrerons en particulier que TreeFinder est correct dans tous les cas, et nous caractériserons les cas où il est complet.
– Enfin, nous étudierons les performances de TreeFinder et la qualité des résultats fournis dans une étude expérimentale.

Tout au long de ce chapitre, nous allons illustrer notre méthode sur la base de données d'arbres de la figure IV.1, dans laquelle nous n'avons pas fait apparaître la racine commune.



(a) T_1 (b) T_2 (e) T_2'

(c) T_3 (d) T_4

Fig. IV.1 – $\mathcal{BA}$

(a) P_1 (b) P_2

Fig. IV.2 – Arbres 2-fréquents

Avec un seuil de fréquence absolu $\varepsilon = 2$, les arbres fréquents par identifiants de cette base sont les arbres P_1 et P_2 représentés sur la figure IV.2.

1 Clustering guidé par les occurrences de paires d'étiquettes fréquentes

Entrée : Cette première étape prend en entrée une abstraction des arbres $\{T_1, ..., T_n\}$ de $\mathcal{BA}$, où chaque arbre T_i est représenté comme une transaction dont les items $l * m$ représentent des paires (l, m) d'étiquettes de deux nœuds x et y tels que x est ancêtre de y. Nous noterons $\mathcal{I}$ l'ensemble de tous les items $l * m$ existants dans la base de données d'arbres $\mathcal{BA}$.

Dans notre exemple, les transactions correspondant aux arbres de la figure IV.1 sont :
T_1 = { Dealer*UsedCar, Dealer*NewCar, UsedCar*Model, UsedCar*Year, UsedCar*Price, NewCar*Model, NewCar*Price, Dealer*Model, Dealer*Year, Dealer*Price}

T_2 = { NetAds*UsedCar, UsedCar*Type, UsedCar*Model, UsedCar*GeneralInfo, GeneralInfo*Price, GeneralInfo*Year, UsedCar*Price, UsedCar*Year, NetAds*Model, NetAds*Price, NetAds*Year, Type*Model }

T_2' = { UsedCar*Model, Model*Year, Model*Price, UsedCar*Year, UsedCar*Price }

T_3 = { Book*Refs, Book*Price, Refs*Title, Refs*Author, Book*Title, Book*Author}

T_4 = { Book*List, Book*Title, List*Author, Book*Author }

Cette décomposition des arbres en items séparés correspondant aux paires d'étiquettes de nœuds ancêtres/descendants nous fait perdre les informations structurelles, mais nous permet d'utiliser des algorithmes standards de recherche d'itemsets fréquents. Ceux-ci nous permettent de découvrir des ensembles de paires d'étiquettes fréquentes dans les données d'entrée. La co-occurrence de paires de labels est considérée comme sémantiquement signifiante si elle se produit fréquemment dans les données d'entrée.

Méthode de clustering : Comme nous l'avons vu dans le chapitre III, il existe de très nombreuses méthodes pour calculer des itemsets fréquents. Lorsque nous avons implémenté TreeFinder, nous avons utilisé la seule que nous avions à notre disposition, à savoir l'algorihme Apriori le plus simple. Nous lui fournissons l'ensemble de transactions décrit ci-dessus, ainsi que le seuil de fréquence ε.

Sortie : Nous nous intéressons aux itemsets fréquents maximaux retournés par Apriori, ainsi qu'à leurs ensembles-support. Ce sont ces ensembles-support qui forment les clusters, les itemsets fréquents maximaux fournissant quant à eux une caractérisation de chaque cluster.

Par exemple, sur notre base de données d'exemple $\mathcal{BA}$ de la figure IV.1, avec un seuil de fréquence absolu $\varepsilon = 2$, les itemsets fréquents maximaux sont :
s_1 = {UsedCar*Model, UsedCar*Year, UsedCar*Price}
s_2 = {Book*Price, Book*Title, Book*Author}
Les ensembles-support, et donc les clusters retournés par cette première étape, sont :
$C_1 = ensSupport(s_1) = \{T_1, T_2, T_2'\}$
$C_2 = ensSupport(s_2) = \{T_3, T_4\}$

2 Calcul des arbres communs maximaux

Entrée : Cette étape prend en entrée les clusters produits par l'étape précédente, et calcule pour chaque cluster les plus gros arbres inclus dans tous les arbres du cluster.

Pour cela, nous représentons tous les arbres de chaque cluster sous forme de formules relationnelles.

Encodage relationnel :

L'encodage relationnel que nous considérons pour un arbre T est une conjonction d'atomes notée $Rel(T)$, décrivant la relation de parenté sur les nœuds de T. Nous introduisons aussi la

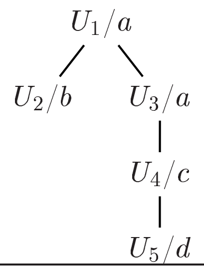
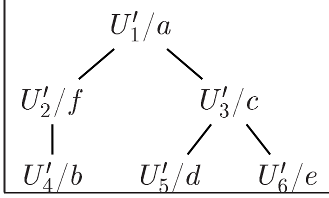
A_i	$Rel(A_i)$	$Rel^+(A_i)$
	$ab(U_1, U_2)$ $aa(U_1, U_3)$ $ac(U_3, U_4)$ $cd(U_4, U_5)$	$a{*}b(U_1, U_2)$ $a{*}a(U_1, U_3)$ $a{*}c(U_1, U_4)$ $a{*}d(U_1, U_5)$ $a{*}c(U_3, U_4)$ $a{*}d(U_3, U_5)$ $c{*}d(U_4, U_5)$
	$af(U'_1, U'_2)$ $ac(U'_1, U'_3)$ $fb(U'_2, U'_4)$ $cd(U'_3, U'_5)$ $ce(U'_3, U'_6)$	$a{*}f(U'_1, U'_2)$ $a{*}c(U'_1, U'_3)$ $a{*}b(U'_1, U'_4)$ $a{*}d(U'_1, U'_5)$ $a{*}e(U'_1, U'_5)$ $f{*}b(U'_2, U'_4)$ $c{*}d(U'_3, U'_5)$ $c{*}e(U'_3, U'_6)$

Fig. IV.3 – Deux arbres A_1 et A_2, et leur encodage relationel

représentation relationnelle $Rel^+(T)$, qui décrit la fermeture transitive de la relation de parenté, c'est-à-dire la relation d'ancestralité.

Définition 26 (Description relationnelle d'arbres étiquetés) *Soit T un arbre étiqueté. $Rel(T)$ est la conjonction de tous les atomes $ab(u, v)$, tels que u et v sont des nœuds de T, u d'étiquette a, v d'étiquette b, et u est le père de v.*

$Rel^+(T)$ est la conjonction d'atomes $a{*}b(u, v)$, tels que u et v sont des nœuds de T, u d'étiquette a, v d'étiquette b, et u est un ancêtre de v.

La figure IV.3 illustre les deux fonctions d'encodage *Rel* et Rel^+ pour deux arbres étiquetés.

LGG et θ-subsomption

Pour extraire les arbres maximaux inclus dans tous les arbres de $C_i = \{T_{i_1}, ..., T_{i_m}\}$, nous calculons la *Least General Generalization* [Plotkin 1970] des formules relationnelles encodant les arbres :

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

La *Least General Generalization* de deux formules relationnelles $Rel(f_1)$ et $Rel(f_2)$ est la formule la plus spécifique qui θ-subsume $Rel(f_1)$ et $Rel(f_2)$. [Plotkin 1970] a montré qu'à un renommage près, la *Least General Generalization* de deux formules logiques conjonctives sans symbole de fonctions est unique.

Définition 27 (θ-subsomption) *Soient $\mathcal{C}$ et $\mathcal{C}'$ deux formules de la logique du premier ordre. On dit que $\mathcal{C}$ θ-subsume $\mathcal{C}'$ s'il existe un mapping θ des variables de $\mathcal{C}$ dans les variables et constantes* *de $\mathcal{C}'$ tel que tout atome de $\mathcal{C}\theta$ apparaisse dans $\mathcal{C}'$.*

L'intérêt de la θ-subsomption est que pour des formules relationnelles encodant des arbres, elle est équivalente à l'inclusion stricte faible que nous considérons dans ce chapitre.

Proposition 1 *Soient T et T' deux arbres étiquetés. Il existe une inclusion stricte faible entre T et T' ssi $Rel^+(T)$ θ-subsume $Rel^+(T')$.*

Preuve : Supposons que T soit inclus de manière stricte faible dans T'. Alors il existe un mapping μ entre les nœuds de T et les nœuds de T', qui préserve la relation d'ancestralité, les labels des nœuds, et qui n'est pas nécessairement injectif. Si on utilise ce mapping μ comme le mapping θ de la θ-subsomption, on obtient immédiatement que $Rel^+(T)$ θ-subsume $Rel^+(T')$, puisque les relations d'ancestralités préservées par μ sont justement les atomes de $Rel^+(T)$ et $Rel^+(T')$.

Supposons réciproquement que $Rel^+(T)$ θ-subsume $Rel^+(T')$. Alors il existe un mapping non nécessairement injectif θ entre les nœuds de T et les nœuds de T' tel que tous les atomes de $Rel^+(T)\theta$ apparaîssent dans $Rel^+(T')$. Ces atomes correspondant à des relations d'ancestralité, cela signifie que par θ, toutes les relations d'ancestralité de T se retrouvent dans T'. θ préserve donc l'ancestralité, il préserve aussi de façon évidente les labels : par θ, $a * b(u, v)$ deviendra $a * b(\theta(u), \theta(v))$ donc les labels sont préservés. Donc T est inclus de manière stricte faible dans T'. □

On en déduit que les formules logiques produites par la *Least General Generalization* sont bien les plus gros arbres inclus de manière stricte faible dans tous les arbres de $C_i = \{T_{i_1}, ..., T_{i_m}\}$.

Si on reprend l'exemple de la figure IV.3, on a :

$$LGG(Rel^+(T_1), Rel^+(T_2)) = a^*b(U_1, U_2) \wedge a^*c(U_1, U_3) \wedge a^*d(U_1, U_4) \wedge c^*d(U_3, U_4).$$

Sortie : L'ensemble d'arbres résultant du décodage en arbres des *least general generalizers* des formules relationnelles encodant les arbres de chaque cluster.

Le point important est qu'à partir de $LGG(Rel^+(T_{i_1}), ..., Rel^+(T_{i_m}))$, qui est une conjonction d'atomes de la forme $l * m(U_i, U_j)$, modélisant la relation d'ancestralité, nous pouvons reconstruire les atomes définissant la relation parent sous-jacente. Cela vient du fait que de part la structure d'arbre de $\{T_{i_1}, ..., T_{i_m}\}$, la relation de parenté implicite p dont la fermeture transitive mène à $LGG(Rel^+(T_{i_1}), ..., Rel^+(T_{i_m}))$ a nécessairement une structure de forêt, et la relation d'ancestralité explicite correspondante a une structure de graphe orienté acyclique. C'est une traversée de cette structure de graphe orienté acyclique suivant un ordre topologique qui permet de reconstruire la relation parent à partir de la relation ancêtre qui existre dans $LGG(Rel^+(T_{i_1}), ..., Rel^+(T_{i_m}))$. Soit $star^{-1}(LGG(Rel^+(T_{i_1}), ..., Rel^+(T_{i_m})))$ la conjonction d'atomes de la forme $lm(U_i, U_j)$ résultante.

Dans l'exemple de la figure IV.3 :

$star^{-1}(LGG(Rel^*(T_1), Rel^*(T_2))) = ab(U_1, U_2) \wedge ac(U_1, U_3) \wedge cd(U_3, U_4)$.

La figure IV.4 récapitule les *least general generalizers* de l'exemple de la figure IV.3 ainsi que leur décodage en arbre.

3 Caractérisation formelle des résultats de TREEFINDER

La proposition suivante montre que les résultats de TREEFINDER sont corrects. Nous fournissons aussi une condition suffisante pour que ces résultats soient complets, c'est-à-dire comprennent *tous* les arbres fréquents maximaux présents dans les arbres donnés en entrée.

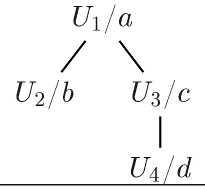
<table>
<tr><th>LGG</th><th>$star^{-1}$</th></tr>
<tr><td>$a{*}b(U_1, U_2)$
$a{*}c(U_1, U_3)$
$a{*}d(U_1, U_4)$
$c{*}d(U_3, U_4)$</td><td>$ab(U_1, U_2)$
$ac(U_1, U_3)$
$cd(U_3, U_4)$</td></tr>
<tr><td colspan="2">$Rel^{-1}(star^{-1}(LGG(Rel^+(T_1), Rel^+(T_2))))$</td></tr>
<tr><td colspan="2"></td></tr>
</table>

Fig. IV.4 – *least general generalizers* et leur décodage en arbre

Proposition 2 *Soit $\mathcal{BA}$ une base de données d'arbres, soit ε un seuil de fréquence, soient $ft_1, ..., ft_n$ les arbres obtenus en sortie de* TreeFinder *appliqué à $\mathcal{BA}$.*

Correction : *$ft_1, ..., ft_n$ sont des arbres ε-fréquents par identifiants pour $\mathcal{BA}$.*

Condition suffisante pour la complétude : *S'il n'y a pas de recouvrement entre les paires d'étiquettes de nœuds ancêtres/descendants des arbres des ensembles-support des arbres fréquents maximaux et les paires d'étiquettes de nœuds ancêtres/descendants des autres arbres de la base de données d'arbres, alors* TreeFinder *trouve tous les arbres fréquents maximaux.*

Preuve :

Correction

Soit ft un arbre produit par TreeFinder. Par construction, il est inclus dans tous les arbres d'un cluster C, et par construction, ce cluster contient au moins ε arbres d'identifiants différents. Donc ft est ε-fréquent par identifiants.

Complétude

Soit ft un arbre fréquent maximal. Soit $\{T_{i_1}, ..., T_{i_m}\}$ son ensemble-support. Si la condition énoncée ci-dessus est vérifiée, l'ensemble des items $l * m$ représentant l'ensemble des paires d'étiquettes de nœuds ancêtres/descendants de ft est un itemset fréquent qui a pour ensemble support exactement $\{T_{i_1}, ..., T_{i_m}\}$. $\{T_{i_1}, ..., T_{i_m}\}$ est donc un cluster qui sera trouvé par la première étape de TreeFinder, et la LGG de la deuxième étape trouvera nécessairement ft.

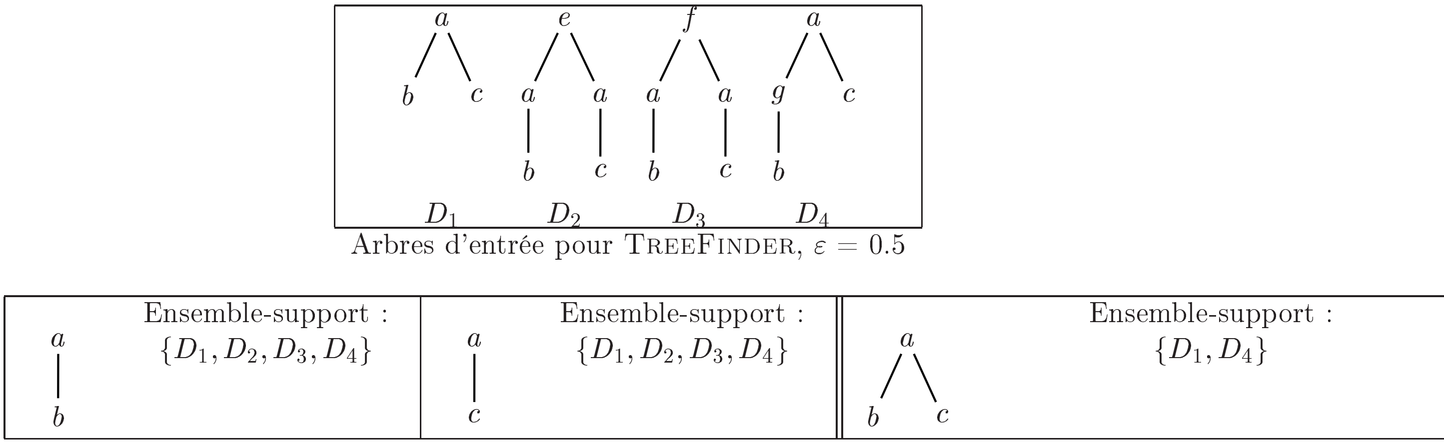
Donc TreeFinder trouvera bien tous les arbres fréquents maximaux.

□

L'exemple suivant montre pourquoi TreeFinder ne peut pas garantir de trouver tous les arbres ε-fréquents maximaux dans le cas général.

La figure IV.5 illustre un cas où TreeFinder appliqué avec un seuil de fréquence relatif de 0.5 va trouver un seul gros cluster contenant tous les arbres donnés en entrée, correspondant à l'itemset fréquent maximal $\{a * b, a * c\}$, qui apparaît dans tous les arbres.

Le résulat de TreeFinder sera donc deux arbres ft_1 et ft_2, réduits à une arête chacun, qui sont communs à tous les arbres donnés en entrée : ils sont 1.0-fréquents, et donc *a fortiori* 0.5-fréquents.



Arbres d'entrée pour TreeFinder, $\varepsilon = 0.5$

Sortie de TreeFinder : deux arbres 1.0-fréquents — Arbre 0.5-fréquent manqué par TreeFinder

Fig. IV.5 –

Toutefois, TreeFinder ne trouve pas l'unique arbre 0.5 fréquent dont la racine est étiquetée par a et qui a deux fils étiquetés par b et c respectivement.

Cet exemple est représentatif du fait que TreeFinder peut échouer à trouver tous les arbre ε-fréquents des données. Nous allons estimer de façon expérimentale dans la section suivante cette perte de complétude par comparaison avec l'algorithme idéal, auquel nous nous référerons sous le nom d'AprioriTree. AprioriTree est une extension simple de l'algorithme par niveaux Apriori aux structures d'arbres, détaillée dans l'algorithme 13.

Algorithme 13 AprioriTree

Entrée : Une base de données d'arbres $\mathcal{BA}$, un seuil de fréquence ε
Sortie : Tous les arbres ε-fréquents par identifiants de $\mathcal{BA}$

Trouver les arbres de taille 1 (1 seule arête) qui sont communs à au moins ε arbres données en entrée
$i \leftarrow 1$
Tant que il existe des arbres fréquents de taille i **Faire**
    Générer des arbres fréquents candidats de taille $i+1$ à partir des arbres fréquents de taille i et des arbres fréquents de taille 1.
    Tester l'inclusion de ces arbres fréquents candidats dans les données, pour ne retenir que ceux qui sont effectivement fréquents.
    $i \leftarrow i+1$
Fin Tant que

AprioriTree est une méthode naïve qui n'est pas applicable en pratique car elle ne passe pas à l'échelle : premièrement, l'étape de test repose intensivement sur l'inclusion d'arbres, qui est NP-complète. De plus, l'augmentation du nombre d'arbres fréquents candidats construits à partir d'un arbre à i arêtes est de $i \times e$, où e est le nombre d'arêtes fréquents. Par contraste, dans Apriori le nombre de candidats construits à partir d'un itemset de taille i est $e - i$, où e est le nombre d'items fréquents.

4 Résultats expérimentaux

Nous avons implémenté l'algorithme TreeFinder en C++, et sa validation expérimentale est détaillée dans cette section. Nous commencerons par exposer le but de nos expérimentations et le cadre expérimental que nous avons choisi. Puis nous reporterons et analyserons nos résultats en fonction du coût des calculs, et de la qualité des arbres fréquents obtenus.

4.1 But des expérimentations

Un des critères les plus importants pour évaluer un algorithme de fouille de données est sa robustesse et son passage à l'échelle en fonction des caractéristiques de ses données d'entrée. Ainsi, nous allons évaluer le coût de TreeFinder en fonction du nombre et de la taille des arbres donnés en entrée dans la section 4.3.

Un autre critère est la qualité des résultats fournis par TreeFinder. Considérons l'algorithme de référence AprioriTree. La proposition 2 a donné une condition suffisante pour que TreeFinder trouve exactement les mêmes arbres fréquents qu'AprioriTree. Toutefois dans le cas général TreeFinder manquera certains arbres fréquents, comme l'illustre la figure IV.5.

Le deuxième but de nos expériences sera donc d'estimer le pourcentage d'arbres fréquents manqués par TreeFinder, en comparaison avec l'algorithme de référence. Ce pourcentage, que nous appelerons *facteur de perte*, sera étudié expérimentalement dans la section 4.4.

Toutes les expériences considèrent des jeux de données artificiels de taille moyenne, suivant le cadre expérimental donné dans la section suivante.

4.2 Cadre expérimental

Afin de pouvoir contrôler précisément ces premières investigations, nous considérons des données artificielles, générées à partir d'arbres fréquents cibles prédéfinis, en utilisant un générateur aléatoire de structures arborescentes. Ce générateur fonctionne de la manière suivante :

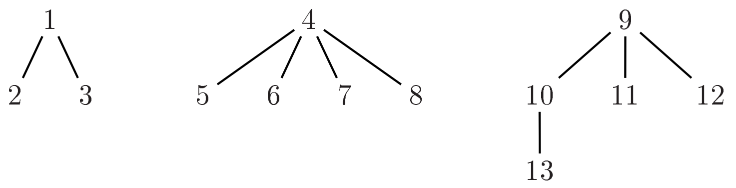
- Un ensemble d'arbres fréquents cibles $\mathcal{P} = \{P_1, ..., P_K\}$ est donné par l'utilisateur.
- Un ensemble d'étiquettes externes $\mathcal{L}$ est défini, disjoint (sauf mention explicite du contraire) des étiquettes utilisées dans les arbres fréquents cibles. La taille L de $\mathcal{L}$ est un paramètre fourni par l'utilisateur au générateur.
- Chaque arbre de la base de données d'arbre est généré récursivement. Soit u le premier nœud d'une liste de nœuds, initialisée à la racine. Deux options sont considérées : avec une probabilité p, une copie (perturbée) de l'un des arbres fréquents cibles est insérée en u. Autrement (probabilité $1-p$), le nombre de nœuds fils de u est tiré uniformément dans $[0, B]$, où B est le facteur de branchement maximal ; l'étiquette de u est tirée avec une probabilité uniforme dans $\mathcal{L}$; u est enlevé de la liste de nœuds et tous ses fils y sont ajoutés.
- L'insertion d'un arbre fréquent cible P, sélectionné uniformément dans $\mathcal{P}$, est effectuée en copiant au nœud u une copie perturbée de P. Un nombre m de nœuds additionnels est tiré aléatoirement dans $[0, \delta]$, où δ est un paramètre du générateur. Ces nœuds sont ensuite insérés à des positions aléatoires dans P, et leurs étiquettes sont sélectionnées aléatoirement dans $\mathcal{L}$.

Les paramètres du générateur d'arbres aléatoires sont résumés dans le tableau IV.1.

À cause de la nature stochastique de la génération des jeux de données, les résultats reportés sont la moyenne des les résultats obtenus pour dix exécutions indépendantes (mêmes paramètres mais graine de génération aléatoire différente).

K	Nombre d'arbres fréquents cibles
L	Nombre de labels externes
p	Probabilité d'inclure un arbre fréquent cible sous un nœud
δ	Nombre maximal de nœuds ajoutés pour perturber un arbre fréquent cible
B	Facteur de branchement maximal
D	Profondeur maximale d'un arbre

TAB. IV.1 – Paramètres du générateur d'arbres aléatoires



(a) P_1 (b) P_2 (c) P_3

FIG. IV.6 – Arbres fréquents cibles

4.3 Étude empirique des coûts de calcul

Nous avons effectué plusieurs expériences, en générant jusqu'à 10000 arbres artificiels, avec les paramètres suivants :

- L'ensemble d'arbres fréquents cibles est représenté dans la figure IV.6.
- Le nombre total d'étiquettes est 100. Sans perte de généralité, les étiquettes 1 à 13 sont les étiquettes des arbres fréquents cible, et $\mathcal{L}$ est l'ensemble des étiquettes $[14, 100]$. Dans ces tests, les étiquettes des arbres fréquents cibles ne sont pas disjointes des étiquettes externes, donc $L = 100$.
- La probabilité p d'inclure un arbre fréquent cible à chaque nœud est 0.2.
- Le nombre maximal de nœuds δ que l'on peut ajouter pour perturber un arbre fréquent cible est 25.
- Le facteur de branchement maximal B est 5.
- La profondeux maximale P est 3.

Une première série de résultats, obtenus avec le seuil de fréquence relatif 0.05, est donnée dans la figure IV.7.

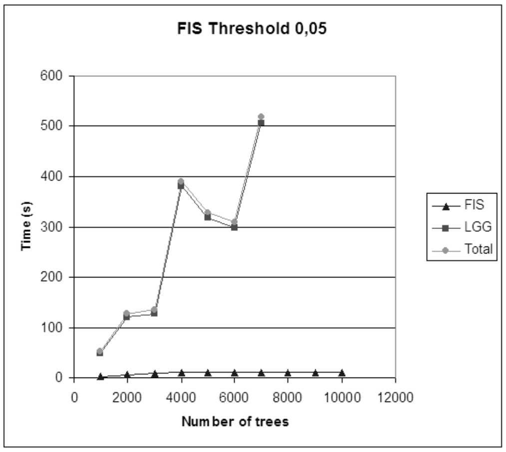
Le coût moyen (figure IV.7 a) est dominé par le coût de la *Least General Generalization* (LGG sur les courbes). Celle-ci est donc notre facteur limitant ; nous n'avons pu obtenir aucun résultat au delà de 8000 arbres à cause d'une saturation de la mémoire.

Des résultats similaires sont obtenus en variant la taille moyenne des arbres. Pour ces tests le paramètre p a été mis à 1.0, donc les arbres sont juste des copies perturbées des arbres fréquents cibles, et nous faisons varier la valeur de δ. L'étape de recherche d'itemset fréquents (FIS sur les courbes) dépend uniquement du nombres d'arêtes dans les arbres fréquents cibles. Mais les ensembles-support des ensembles d'arêtes fréquents deviennent vite très grands, ce qui implique que la *Least General Generalization* considère beaucoup d'arbres (Figure IV.7 b).

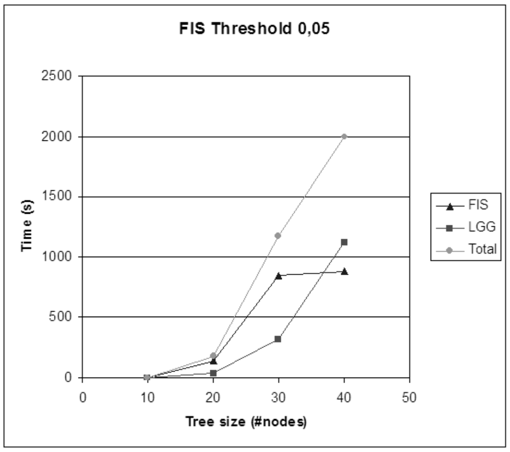
Un nouvelle fois, le facteur limitant en complexité est l'étape de *Least General Generalization*.

4.4 Étude empirique du facteur de perte de TreeFinder

Cette étude vise à mesurer le pourcentage d'arbres fréquents manqués par TreeFinder.



a) Le nombre d'arbres varie, taille d'arbre aléatoire



b) La taille des arbres varie, pour 1000 arbres

Fig. IV.7 – Tests de passage à l'échelle

Comme nous l'avons vu dans la proposition 2, le facteur de perte est zéro quand les arbres fréquents ne se recouvrent pas, c'est-à-dire n'ont aucune paire d'étiquettes des nœuds ancêtre/descendant en commun. Cette étude expérimentale considère donc cinq jeux d'arbres fréquents cibles, avec un taux de recouvrement entre les paires d'étiquettes es nœuds ancêtre/descendant des arbres de plus en plus élevé. Cinq séries d'expériences ont été effectuées, correspondant à ces cinq ensembles d'arbres fréquents cibles, représentés figure IV.8.

Recouvrement	P_1	P_2	P_3	P_4
$overlap_0$:	1 / \ 2 3	4 / \ 5 6	7 / \ 8 9	10 / \ 11 12
$overlap_1$:	1 / \ 2 3	1 / \ 3 4	5 / \ 6 7	8 / \ 9 10
$overlap_2$:	1 / \ 2 3	1 / \ 3 4	1 / \ 4 5	6 / \ 7 8
$overlap_3$:	1 / \ 2 3	1 / \ 3 4	1 / \ 4 5	1 / \ 5 6
$overlap_4$:	1 / \ 2 3	1 / \ 3 4	1 / \ 4 5	1 / \ 5 2

FIG. IV.8 – Arbres fréquents cibles avec recouvrement croissant

- $overlap_0$ correspond au cas de base, sans recouvrement entre les arêtes des arbres
- $overlap_1$ correspond à un faible recouvrement (P_1 et P_2 partagent l'arête $1-3$) ; le taux de recouvrement, défini comme le pourcentage moyen d'arêtes qu'un arbre fréquent partage avec un autre arbre fréquent, est de 25%.
- $overlap_2$ correspond à un recouvrement moyen (P_1 et P_2 partagent l'arête $1-3$; P_2 et P_3 partagent l'arête $1-4$) ; le taux de recouvrement est 50%.
- $overlap_3$ correspond à un recouvrement élevé (P_1 et P_2 partagent l'arête $1-3$; P_2 et P_3 partagent l'arête $1-4$; P_3 et P_4 partagent l'arête $1-5$) ; le taux de recouvrement est 75%.
- $overlap_4$ correspond à un recouvrement très élevé (P_1 et P_2 partagent l'arête $1-3$; P_2 et P_3 partagent l'arête $1-4$; P_3 et P_4 partagent l'arête $1-5$; P_4 et P_1 partagent l'arête $1-2$) ; le taux de recouvrement est 100%.

Les autres paramètres du générateur sont :

- Le nombre L d'étiquettes externes est 50.
- La probabilité p d'inclure un arbre fréquent cible à chaque nœud est 0.4.

TREEFINDER est exécuté pour les seuils de fréquence relatifs ε suivants : 0.2 ; 0.1 ; 0.05. La figure IV.9 montre le facteur de perte pour chaque seuil de fréquence ε et chaque ensemble d'arbres fréquents cible.

Le facteur de perte a été calculé de la manière suivante. Idéalement, il mesure le pourcentage d'arbres fréquents que retourne APRIORITREE, et qui sont manqués par TREEFINDER. Toutefois, en fonction de la complexité inhérente à APRIORITREE, tous les arbres fréquents apparaissant dans

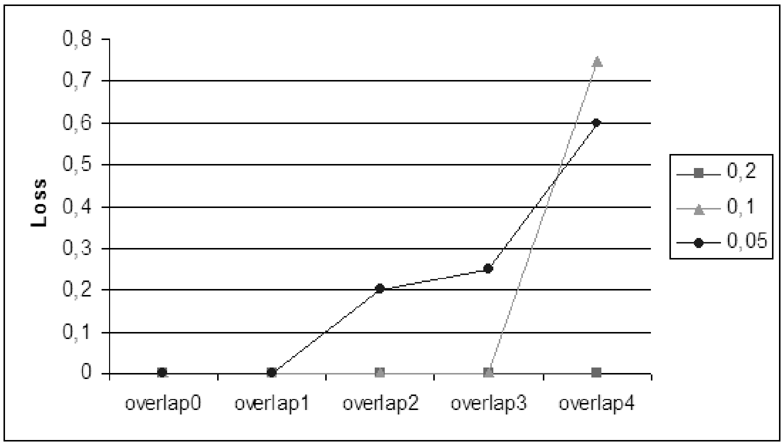


Fig. IV.9 – Analyse de sensibilité : facteur de perte de TreeFinder pour $\varepsilon = 0.05$, 0.1 et 0.2 en fonction du taux de recouvrement dans les ensembles d'arbres fréquents cibles

nos jeux de données ne peuvent pas être extraits avec une complexité raisonnable. Ce nombre est donc approximé durant l'étape de génération des jeux de données.

Pour un arbre généré T, soit $\mathcal{Q}(T)$ l'ensemble des arbres fréquents cibles P_i insérés dans T par le générateur.

Donc pour chaque arbre fréquent cible P_i on peut déterminer son ensemble-support à partir de tous les $Q(T)$ des arbres T produits par le générateur.

On en déduit le nombre des arbres fréquents :

$$AT(\varepsilon, \{P_1, ..., P_n\}) = |\bigcup_{i \in [1,n]} EnsembleSupport(P_i)|$$

Ce nombre est une borne inférieure du véritable nombre d'arbres fréquents dans la base de données d'arbres produite par le générateur, car il ne compte pas les éventuels artefacts de génération.

Pour chaque seuil de fréquence ε et ensemble d'arbres fréquents cibles $overlap_i$, le facteur de perte est défini par :

$$Perte(\varepsilon, overlap_i) = 1 - \frac{TF(\varepsilon, overlap_i)}{AT(\varepsilon, overlap_i)}$$

où $TF(\varepsilon, overlap_i)$ est le nombre (médian) d'arbres fréquents découverts par TreeFinder sur dix jeux de données générés indépendamment à partir de l'ensemble d'arbres fréquents cibles $overlap_i$, et $AT(\varepsilon, overlap_i)$ est la borne inférieure du nombre réel d'arbre fréquents, définie ci-dessus. Ce facteur de perte est donc une estimation pessimiste.

Comme prévu par la proposition 2, le facteur de perte est zéro dans le cas d'arbres fréquents cibles sans recouvrement. Dans les autres cas, deux types de comportement sont observés suivant le seuil de fréquence.

Pour un petit seuil de fréquence ($\varepsilon = 0.05$), le facteur de perte s'accroît lentement lorsque le taux de recouvrement augmente de faible à élevé ; la perte est de l'ordre de 20% pour un taux de recouvrement de 75%.

Pour un seuil de fréquence plus élevé ($\varepsilon = 0.1$), le facteur de perte reste nul jusqu'à ce que le taux de recouvrement devienne très élevé (tous les arbres fréquents cibles partagent une arête avec un autre arbre fréquent cible). Dans ce cas le facteur de perte augmente brutalement à 80%. Une interprétation possible est qu'entre $overlap_3$ et $overlap_4$ une discontinuité se produit : toutes les arêtes deviennent également fréquentes, cela explique que les ensembles d'arêtes ne puissent plus être filtrés par le seuil de fréquence ($TF(\varepsilon = 0.1, overlap_4) \approx TF(\varepsilon = 0.05, overlap_4)$). En même temps, le nombre d'arbres fréquents augmente alors que ε diminue, à cause de la monotonie ($AT(\varepsilon = 0.1, overlap_4) < AT(\varepsilon = 0.05, overlap_4)$), ce qui explique pourquoi le facteur de perte est plus mauvais pour $\varepsilon = 0.1$ que pour $\varepsilon = 0.05$ dans le cas de $overlap_4$.

Ceci peut être rapproché de l'exemple de la figure IV.5 : plus il y a de recouvrement entre les arbres fréquents cibles, plus l'ensemble-support des ensembles d'arêtes est grande, et donc plus les arbres fréquents découverts par la *Least General Generalization* sont généraux. Et donc plus grandes sont les chances que TreeFinder manque des arbres fréquents.

On peut remarquer que pour un seuil de fréquence plus élevé ($\varepsilon = 0.2$), le taux de perte reste nul.

5 Conclusion

Dans ce chapitre, nous avons présenté une première approche pour découvrir des arbres fréquents. Cette approche est correcte, mais dans le cas général elle n'est pas complète. Son intérêt est réutiliser des algorithmes classiques (Apriori, calcul de *Least General Generalization*) dont l'application est rendue possible par des encodages appropriés des arbres.

Nous allons présenter dans le chapitre suivant des algorithmes corrects et complets. Nous imposerons cependant comme restriction sur les arbres fréquents recherchés qu'un nœud ne peut pas avoir deux fils de même étiquette. De tels arbres fréquents seront appelés des patterns fréquents dans le chapitre suivant.

Chapitre V

Les algorithmes Dryal et Dryade

1 Objectif

Nous rappelons que l'objectif des algorithmes présentés dans ce mémoire est de découvrir des arbres fréquents dans une base de données d'arbres, comme illustré sur la figure V.1.

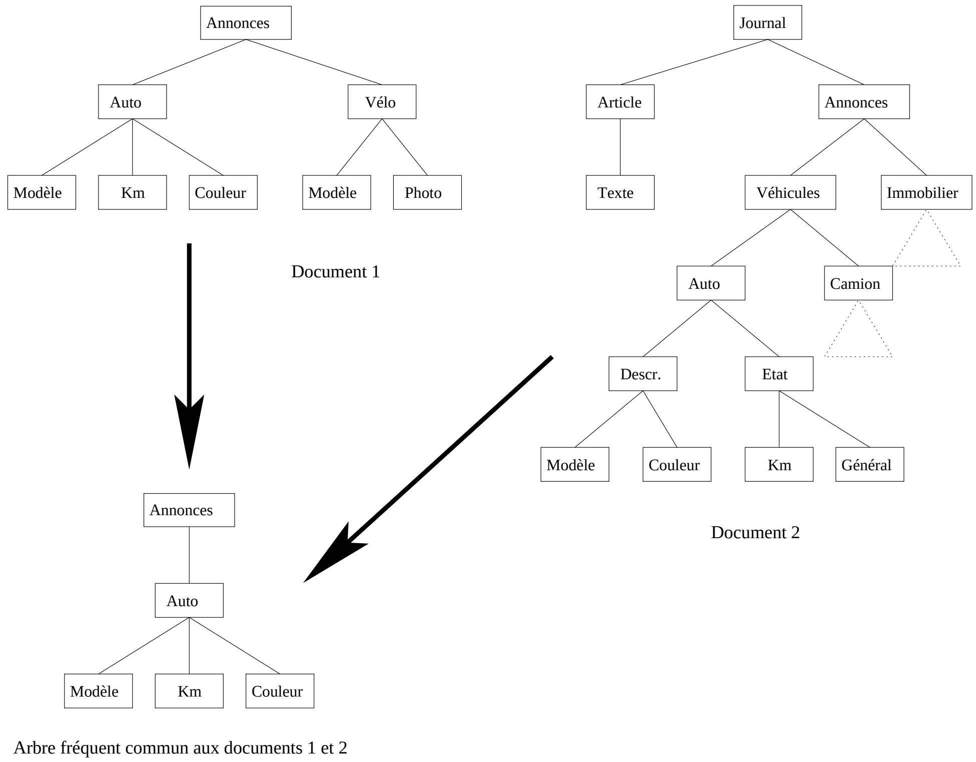


Fig. V.1 – Un exemple de découverte d'arbre fréquent

Comme mentionné précédemment, dans ce chapitre nous restreignons les arbres fréquents que nous voulons découvrir à des **patterns** fréquents, où un pattern est un arbre dont aucun nœud ne peut avoir deux fils de même étiquette. Nous reviendrons sur cette restriction lorsqu'elle interviendra dans les algorithmes, pour montrer le gain en complexité qu'elle permet.

Le présent chapitre est organisé comme suit :
- Nous présentons d'abord un premier algorithme de découverte d'arbres fréquents, l'algorithme DRYAL. Cet algorithme présente le principe de l'approche suivie.
- Puis nous présentons quelques paramétrages de l'algorithme DRYAL, lui permettant de prendre en compte différentes définitions de l'inclusion d'arbres.
- L'algorithme DRYADE[1] reprend l'approche de DRYAL et s'appuie sur une propriété algébrique (les ensembles fermés) pour permettre une résolution efficace au delà des limites de DRYAL.

Pour ces deux algorithmes, nous donnerons des preuves de correction et de complétude.
- La preuve de *correction* permet de s'assurer que tous les résultats donnés par nos algorithmes sont des patterns fréquents.
- La preuve de *complétude* permet de s'assurer que nos algorithmes trouvent **tous** les patterns fréquents présents dans les données.

Nous rappelons dans le tableau 1 les notations définies dans le chapitre II.

$\sqsubseteq$	Inclusion d'arbre (peut être stricte, exacte, ordonnée, faible)	Définition 9 page 17
$\mathcal{EM}_{\sqsubseteq}(T_1, T_2)$	Ensemble des mappings entre les arbres T_1 et T_2 tels que $T_1 \sqsubseteq T_2$	
$Locc_{\sqsubseteq}(T_1, T_2)$	Ensemble des nœuds de T_2 où aboutit la racine de T_1 par un mapping de $\mathcal{EM}_{\sqsubseteq}(T_1, T_2)$	Définition 10 page 21
$\mathcal{EM}_u(T_1, T_2)$	Pour un nœud $u \in Locc_{\sqsubseteq}(T_1, T_2)$, ensemble des mappings de $\mathcal{EM}_{\sqsubseteq}(T_1, T_2)$ conduisant la racine de T_1 sur u	Définition 10 page 21

TAB. V.1 – Rappel des notations

2 L'algorithme DRYAL

Pour toute cette section, nous considérerons comme inclusion d'arbre l'**inclusion stricte faible**, notée $\sqsubseteq_f$; les mappings supports de l'inclusion préservent de façon stricte les étiquettes, préservent la relation d'ancestralité (mais pas nécessairement la relation de parenté) et ne sont pas nécessairement injectifs.

Pour simplifier l'écriture, nous noterons $Locc(X, Y)$ pour $Locc_{\sqsubseteq_f}(X, Y)$.

2.1 Spécifications

2.1.1 Données d'entrée

- $\mathcal{BA} = (D, \Delta, \delta)$ une base de données d'arbres
- L un ensemble d'étiquettes
- ε un seuil de fréquence absolu

[1]Les dryades sont des créatures magiques qui vivent dans les arbres et pour les arbres. Une ancienne légende dit qu'elles sont toutes nées du sang de Dryal, une jeune mortelle ayant pénétré et péri dans une forêt enchantée.

2.1.2 Résultats attendus

– Tous les patterns fréquents présents dans $\mathcal{BA}$ avec un seuil minimal de ε.
– Pour chaque pattern fréquent PF, l'algorithme renvoie aussi $Locc(PF, D)$.

Notation : Pour une base de données d'arbres $\mathcal{BA}$ et un seuil de fréquence ε nous noterons $\mathcal{P}$ l'ensemble de tous les patterns fréquents de $\mathcal{BA}$ atteignant au moins le seuil de fréquence ε.

2.2 Principe de l'algorithme

Les algorithmes existants de découverte d'arbres fréquents, que nous avons décrits dans le chapitre III, sont des algorithmes de type "Générer et Tester". C'est-à-dire qu'ils créent de nombreux arbres candidats, puis testent si ces arbres candidats sont fréquents. De nouveaux arbres candidats sont obtenus par l'ajout d'un arc aux arbres fréquents détectés. Ce processus continue jusqu'à ce qu'on ne trouve plus de nouveaux arbres fréquents.

Nous avons également vu dans le chapitre III que cette méthode est coûteuse, car elle explore souvent une grande partie de l'espace de recherche (arbres fréquents + tous les candidats infréquents). Cela se traduit par une perte d'efficacité, surtout si les arbres fréquents à trouver sont complexes.

Nous avons enfin vu que l'enjeu essentiel pour tous les algorithmes de découverte de motifs fréquents est de tirer partie de la propriété de monotonicité sur la fréquence pour élaguer au maximum l'espace de recherche. Lorsque les motifs sont structurés, outre la propriété de monotonicité, on peut aussi tirer partie de la structure des données pour l'élagage.

Ces heuristiques vont constituer le point fort de l'approche présentée.

Nous proposons un algorithme constructif, tenant compte de la contrainte de fréquence et de la structure des données pour générer un ensemble de patterns candidats aussi restreint que possible, tout en restant complet.

L'algorithme ainsi présenté obtient tous les patterns fréquents présents dans les données dans une approche par niveau, en structurant graduellement les patterns candidats.

Notre méthode est la suivante : nous commençons par découvrir tous les patterns fréquents de profondeur 1. Puis nous accrochons ensembles certains de ces patterns, de manière à obtenir de nouveaux patterns plus complexes, mais toujours fréquents. Par accrochages successifs de patterns fréquents de profondeur 1 sur des patterns fréquents de plus en plus complexes, notre algorithme construit tous les patterns fréquents présents dans les données.

Les paragraphes suivants détaillent cette démarche, en commençant par quelques définitions.

2.3 Définitions

Notre point de départ est l'ensemble PFA_1 des patterns fréquents de profondeur 1, qui seront la base des accrochages :

Définition 28 **(PFA_1)** *On appelera PFA_1 l'ensemble des patterns fréquents par occurrences de profondeur 1.*

$PFA_1 = \{P \mid |Locc(P, D)| \geq \varepsilon \text{ et } profondeur(P) = 1\}$

Pour la suite, lorsqu'aucune confusion ne sera possible, on utilisera le terme *pattern fréquent* pour désigner un pattern fréquent par occurrences.

Exemple : Reprenons l'exemple récapitulatif du chapitre II, figure V.2. Nous utiliserons seulement le datatree D_X, car les identifiants de documents ne nous servent pas dans ce chapitre.

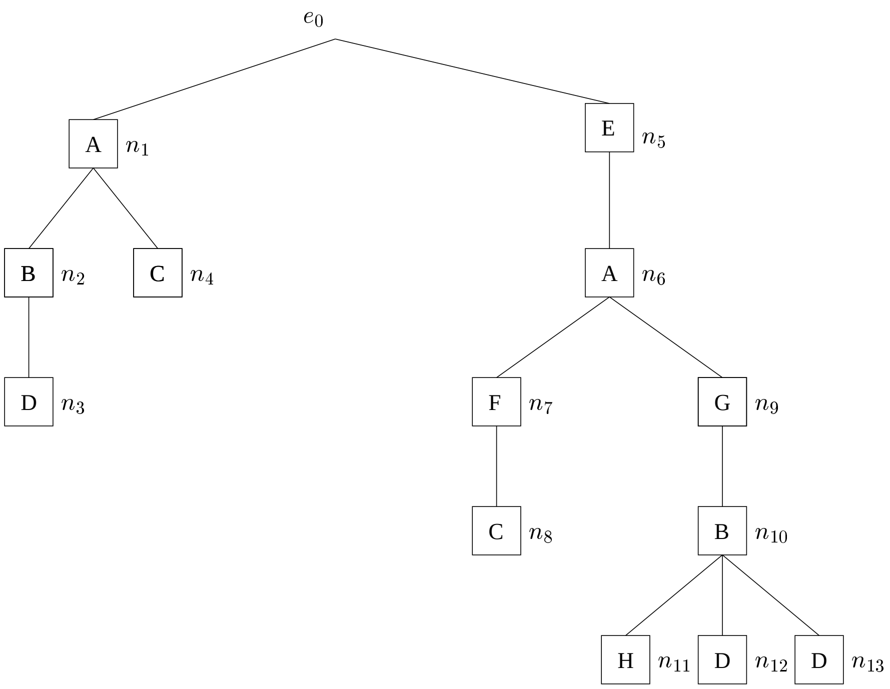


FIG. V.2 – Datatree exemple.

Avec le seuil $\varepsilon = 2$, l'ensemble $PFA_1 = \{P_{11}, P_{12}, P_{13}, P_{14}, P_{15}, P_{16}, P_{17}, P_{18}\}$ extrait de ce datatree est représenté sur la figure V.3.

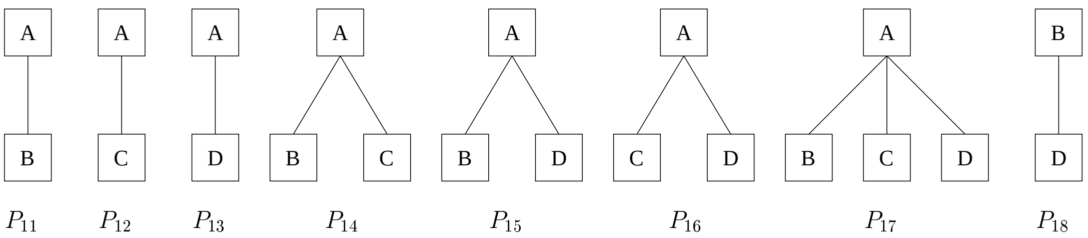


FIG. V.3 – PFA_1 pour D_X

Nous définissons maintenant l'opérateur d'accrochage entre patterns.

Définition 29 (Accrochage) *Soit P un pattern fréquent quelconque, et soit $P_1 \in PFA_1$. On dit que P et P_1 sont* accrochables, *noté $accrochables(P, P_1)$, si l'étiquette de la racine de P_1 est la même que celui d'au moins une des feuilles de P.*

L'ensemble des nœuds de N_P où P_1 peut s'accrocher est $noeuds\_accrochage(P, P_1)$.

L'ensemble de patterns résultant de l'accrochage de P et P_1 est noté $P \oplus P_1$.

On a $|noeuds\_accrochage(P, P_1)| = |P \oplus P_1|$.

Soit $na \in noeuds\_accrochage(P, P_1)$. L'unique pattern résultant de l'accrochage de P et P_1 sur na est noté : $P \oplus_{na} P_1$.

Notons que l'opérateur $\oplus$ n'est ni commutatif, ni associatif.

Notation : Soit un ensemble de patterns quelconques $\{P'_1, ..., P'_n\}$ et soit $P_1 \in PFA_1$. Alors $\{P'_1, ..., P'_n\} \oplus P_1 = \bigcup_{i=1}^{n} P'_i \oplus P_1$.

La figure V.4 montre deux exemples d'accrochage, le premier entre deux patterns de profondeur 1, le second entre un pattern quelconque et un pattern de profondeur 1.

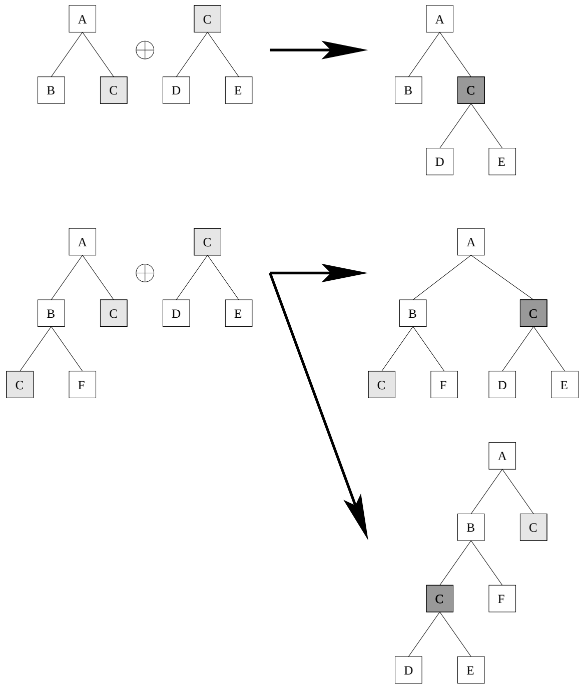


FIG. V.4 – Deux exemples d'accrochages

Notons que l'accrochage de deux patterns fréquents PF_1 et PF_2 ne donne pas nécessairement un pattern fréquent. La définition suivante permet de sélectionner des patterns **fréquents** issus d'accrochages successifs de patterns fréquents.

Définition 30 (PFA_i) *Soit $i \geq 2$. PFA_i est l'ensemble de tous les patterns fréquents résultant de l'accrochage successif de i patterns fréquents de profondeur 1.*

Il est défini récursivement de la façon suivante : pour tout pattern $P_i \in PFA_i$, il existe un pattern $P_{i-1} \in PFA_{i-1}$ et un pattern $P_1 \in PFA_i$ tel que $P_i = P_{i-1} \oplus P_1$.

Exemple : La figure V.5 montre l'ensemble $PFA_2 = \{P_{21}, P_{22}, P_{23}, P_{24}\}$ pour D_X.

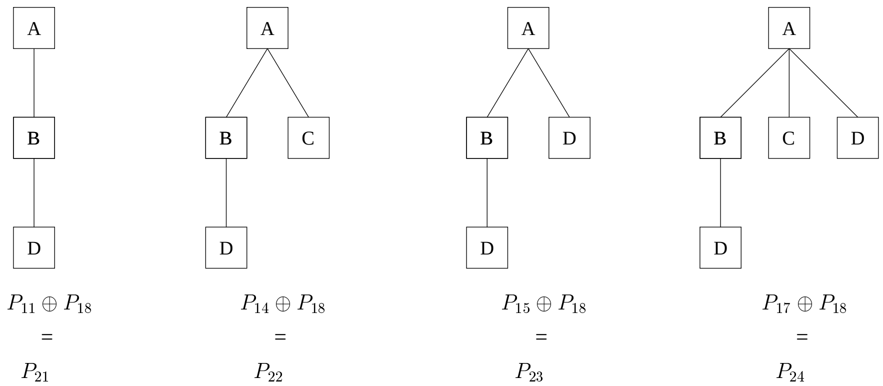


FIG. V.5 – PFA_2 pour D_X

Soit $P_i \in PFA_i$ et $P_1 \in PFA_1$. Soit $P \in P_i \oplus P_1$, pour déterminer sa fréquence et ainsi son appartenance à PFA_{i+1} nous avons donc besoin de calculer $Locc(P, D)$. Le point important est de pouvoir déterminer $Locc(P, D)$ à partir de $Locc(P_i, D)$ et $Locc(P_1, D)$, et ainsi éviter de coûteux tests d'inclusion d'arbre.

On a besoin pour cela de définir la notion d'ancre dans les données d'un accrochage, c'est à dire pour l'accrochage de P_1 sur P_i l'ensemble des couples d'occurrences de P_i et P_1 caractérisant les sous-arbres des données auxquels l'accrochage de P_1 sur P_i peut être associé par mapping.

Définition 31 (Accrochage ancré dans les données) *Soient $P_i \in PFA_i$ et $P_1 \in PFA_1$ deux patterns accrochables. Soit $na \in N_{P_i}$ tel que $na \in noeuds\_accrochage(P_i, P_1)$.*

L'accrochage de P_1 sur P_i par na est ancré dans les données s'il existe $n_i \in Locc(P_i, D)$ et $n_1 \in Locc(P_1, D)$ tels qu'il existe $\mu_i \in \mathcal{EM}_{n_i}(P_i, D)$ et $\mu_1 \in \mathcal{EM}_{n_1}(P_1)$ tq $\mu_i(na) = \mu_1(racine(P_1))$.

*L'ensemble de tels couples (n_i, n_1) est appelé l'**ancre dans les données** de $P_i \oplus_{na} P_1$, et est noté $Ancre_{na}(P_i, P_1)$.*

Par extension $Ancre(P_i, P_1) = \bigcup_{na \in noeuds\_accrochage(P_i,P_1)} Ancre_{na}(P_i, P_1)$.

Cette définition est illustrée sur la figure V.6.

Pour un exemple plus concret, dans le cas de D_X on peut en examinant la figure V.2 déduire $Ancre_B(P_{14}, P_{18}) = \{(n1, n2), (n6, n10)\}$.

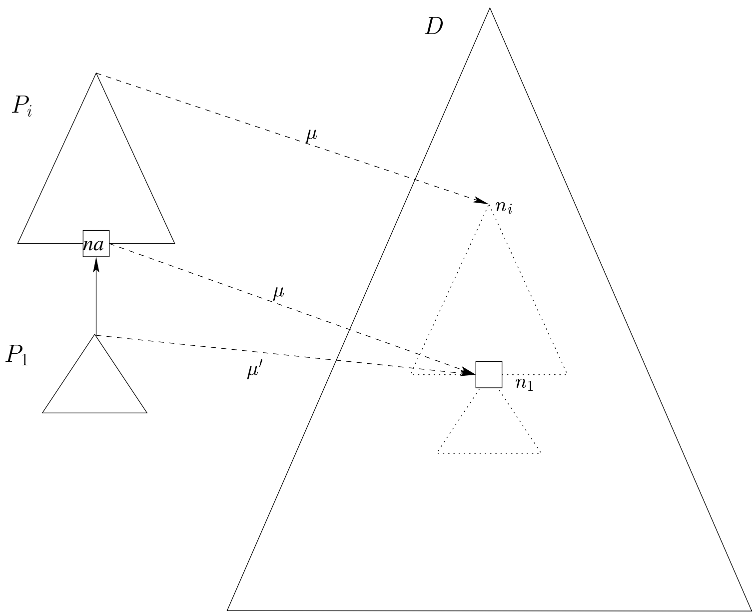


Fig. V.6 – Illustration de l'ancrage dans les données d'un accrochage

On peut en déduire comment calculer l'ensemble des occurrences de $P_i \oplus P_1$ à partir de l'ensemble des occurrences de P_i et de l'ensemble des occurrences de P_1 :

Propriété 7 *Soient $P_i \in PFA_i$ et $P_1 \in PFA_1$ deux patterns accrochables. Soit $na \in N_{P_i}$ tq $na \in noeuds\_accrochage(P_i, P_1)$. Soit $n \in Locc(P_i \oplus_{na} P_1, D)$. On a :*

$$\mathcal{EM}_n(P_i \oplus_{na} P_1, D) = \{\mu_i \circ \mu_1 \mid \mu_i \in \mathcal{EM}_n(P_i, D) \text{ et } \mu_1 \in \mathcal{EM}_{\mu_i(na)}(P_1, D)\}$$

On en déduit :

$$Locc(P_i \oplus_{na} P_1, D) = \{n \in Locc(P_i, D) \mid \exists m \in Locc(P_1, D) \text{ tq } (n, m) \in Ancre_{na}(P_i, P_1)\}$$

Preuve : Soit $P_{i+1} = P_i \oplus_{na} P_1$. Montrons que :

- $\mathcal{EM}_n(P_{i+1}, D) \subseteq \{\mu_i \circ \mu_1 \mid \mu_i \in \mathcal{EM}_n(P_i, D)$ et $\mu_1 \in \mathcal{EM}_{\mu_i(na)}(P_1, D)\}$: trivial, comme $P_i \sqsubseteq P_{i+1}$ et de même $P_1 \sqsubseteq P_{i+1}$, on peut décomposer chaque mapping μ_i de P_{i+1} en un mapping de P_i et un mapping de P_1.
- $\mathcal{EM}_n(P_{i+1}, D) \supseteq \{\mu_i \circ \mu_1 \mid \mu_i \in \mathcal{EM}_n(P_i, D)$ et $\mu_1 \in \mathcal{EM}_{\mu_i(na)}(P_1, D)\}$: Puisque P_i et P_1 sont accrochables sur na, alors $(n, \mu_i(na)) \in Ancre(P_i, P_1)$. Donc $\mu_i \circ \mu_1$ est bien dans $\mathcal{EM}_n(P_{i+1}, D)$.

□

2.4 Algorithme

L'algorithme que nous allons présenter dans cette section construit tous les ensembles de patterns fréquents par accrochage PFA_i.

Notre algorithme est décrit dans l'algorithme 14. Il prend en entrée une base de données d'arbres $\mathcal{BA}$ et un seuil de fréquence absolu ε.

Le résultat de l'algorithme est :

- un ensemble de patterns fréquents, qui est l'union d'ensembles $ER_1, ..., ER_k$. Pour tout $i \in [1, k]$, ER_i est l'ensemble des patterns fréquents résultant d'accrochages successifs de i patterns fréquents de profondeur 1 découverts par l'algorithme.
- pour tout pattern fréquent P découvert, l'algorithme fournit également $Locc(P, D)$, et pour tout $n \in Locc(P, D)$, l'algorithme produit $\mathcal{EM}_n(P, D)$.
  Dans les algorithmes, pour représenter l'ensemble des occurrences d'un pattern fréquent P nous utiliserons une structure de données regroupant les occurrences et les mappings. Pour P, cette structure est notée *P.loc*. On a : $P.loc = \{Occ_1, ..., Occ_n\}$. Les objets Occ_i contiennent une racine $Occ_i.racine$ qui est un nœud de D, et un ensemble de mappings $Occ_i.mappings$. On a
  $$\bigcup_i Occ_i.racine = Locc(P, D)$$
  et
  $$\forall i \in [1, n] \;\; Occ_i.mappings = \mathcal{EM}_{Occ_i.racine}(P, D)$$

La première étape de l'algorithme est de trouver l'ensemble ER_1. La procédure *DécouvertePFA*$_1$ est détaillée dans l'algorithme 15. Le principe est simple : pour chaque étiquette l on veut découvrir tous les patterns de profondeur 1 ayant une racine de cette étiquette (boucle principale). On crée donc une matrice dont chaque ligne (transaction) correspond à un nœud n d'étiquette l dans les données. Les colonnes (items) correspondent aux étiquettes des nœuds descendants de n. Puis on applique l'algorithme ECLAT (cf chapitre III) à cette matrice. Cet algorithme nous donne les ensembles fréquents d'étiquettes de nœuds descendants d'un nœud d'étiquette l. On peut donc, pour chacun de ces ensembles fréquents d'étiquettes, créer un pattern fréquent de profondeur 1, de racine l et dont les étiquettes des feuilles sont les étiquettes dans l'ensemble fréquent (ligne 15).

Grâce aux ensembles-support[2] fournies par l'algorithme Eclat, on peut ensuite facilement trouver d'une part les nœuds qui sont les occurrences de racine (ligne 18), d'autre part construire des mappings entre le pattern fréquent de profondeur 1 trouvé et ses instances dans les données (ligne 19).

La découverte de l'ensemble ER_1 est le point de départ de notre algorithme. Deux caractéristiques importantes doivent être notées :

[2]Rappelons que pour un itemset fréquent, son ensemble-support est l'ensemble des transactions le contenant.

– Cette procédure utilise un algorithme de type Apriori classique. Nous avons choisi Eclat car son implantation verticale nous permet d'avoir les ensembles-support dont nous avons besoin, tout en bénéficiant d'une bonne efficacité. Le fait d'utiliser un algorithme de type Apriori nous permet d'avoir accès aux très nombreuses améliorations d'Apriori existantes et à venir.
– On peut voir ici l'effet de la restriction que nous avons imposée sur les arbres fréquents découverts. Nous rappelons que cette restriction est de découvrir des *patterns* fréquents, c'est-à-dire des arbres étiquetés dont aucun nœud ne peut avoir deux fils de même étiquette. Sans cette restriction, l'algorithme de type Apriori utilisé devrait être capable de prendre en entrée une matrice de valeurs non plus booléennes mais entières (pour gérer les répétitions d'étiquettes), et de découvrir des motifs fréquents avec des répétitions d'items. Pour évaluer l'augmentation de complexité, on peut se représenter l'utilisation de valeurs entières dans la matrice par une matrice booléenne avec beaucoup plus de colonnes : l'étiquette l aurait autant de colonnes qu'il peut avoir de répétitions. Or l'algorithme Apriori est justement exponentiel en fonction du nombre d'items. Toute augmentation du nombre de colonnes se traduit par une importante perte d'efficacité.

Algorithme 14 Dryal

Entrée : $\mathcal{BA}$ une base de données d'arbres, ε un seuil absolu, L un ensemble d'étiquettes
Sortie : $\bigcup ER_i$ ensemble de patterns fréquents résultats, et pour chaque pattern fréquent P découvert $P.loc$.

1: $ER_1 \leftarrow$ DécouvertePFA_1 // *voir algorithme 15*
2: $i \leftarrow 1$
3: **Tant que** $ER_i \neq \emptyset$ **Faire**
4: $\quad ER_{i+1} \leftarrow \emptyset$
5: $\quad$ **Pour tout** $P \in ER_i$ **Faire**
6: $\quad\quad$ **Pour tout** $P_1 \in ER_1$ **Faire**
7: $\quad\quad\quad$ **Si** $accrochable(P, P_1)$ **Alors**
8: $\quad\quad\quad\quad ER_{i+1} \leftarrow ER_{i+1} \cup Accrochage(P, P_1)$ // *voir algorithme 16*
9: $\quad\quad\quad$ **Fin Si**
10: $\quad\quad$ **Fin Pour**
11: $\quad$ **Fin Pour**
12: $\quad i \leftarrow i + 1$
13: **Fin Tant que**
14: **Retourner** $\bigcup_{j=1}^{i} ER_j$

À partir du moment où nous avons trouvé l'ensemble ER_1 des patterns fréquents de profondeur 1, les ensembles $ER_{i+1} (i \geq 1)$ se découvrent itérativement en accrochant à chaque étape i les patterns de ER_i et les patterns de ER_1. Ceci est réalisé dans l'algorithme principal par la double boucle des lignes 5 à 11. Pour chaque couple d'éléments $(P_i, P_1) \in ER_i \times ER_1$, on détermine s'ils sont accrochables, c'est-à-dire si P_i a au moins une feuille de même étiquette que la racine de P_1. Si c'est le cas, on utilise la procédure d'accrochage détaillée dans l'algorithme 16.

Pour chaque nœud d'accrochage potentiel n, la procédure *CalculerLocAccr* décrite dans l'algorithme 17 vérifie qu'il existe suffisamment d'occurrences dans les documents supportant l'accrochage sur ce nœud. Si oui, un nouveau pattern fréquent est créé, résultant de l'accrochage de P_1 sur P_i par le nœud n. Les occurrences de ce nouveau pattern sont obtenues en retenant les racines des occurrences de P_i qui supportent l'accrochage entre P_i et P_1 par n, et en composant les mappings associés à ces racines pour P_i avec les mappings des occurrences correspondantes de P_1.

Algorithme 15 DécouvertePFA_1

Entrée : $\mathcal{BA}$ une base de données d'arbres, ε un seuil absolu, L un ensemble d'étiquettes

Sortie : ER_1 l'ensemble des patterns fréquents de profondeur 1, et pour chaque pattern fréquent P, $P.loc$

$ER_1 \leftarrow \emptyset$
Pour tout étiquette $l \in L$ **Faire**
    // Construction de la matrice
    $M_l \leftarrow$ tableau à deux dimensions avec $|\{n \in N_D | label(n) = l\}|$ lignes et $|L|$ colonnes
    Pour tout $n \in N_D$ tq $etiquette(n) = l$ **Faire**
        Pour tout $l' \in L$ **Faire**
            Si $\exists m \in N_D$ tq $ancetre(n, m)$ et $etiquette(m) = l'$ **Alors**
                $M_l[n, l'] = 1$
            Fin Si
        Fin Pour
    Fin Pour
    $EEL \leftarrow Eclat(M_l, \varepsilon)$ *// EEL est un ensemble d'ensemble d'étiquettes obtenus par l'algorithme // ECLAT. Chaque ensemble d'étiquettes est associé à son ensemble-support.*
    // Création des patterns fréquents
    Pour tout $EL \in EEL$ **Faire**
        $P_{new} \leftarrow$ pattern de profondeur 1, étiquette de la racine $= l$, étiquettes des feuilles $= EL.itemset$
        $P_{new}.loc \leftarrow \emptyset$
        Pour tout $n \in EL.ensSupport$ **Faire**
            $NewOcc.racine = n$
            $NewOcc.mappings = \{\mu \mid \mu(racine(P_{new})) = n$ et $\forall m \in feuilles(P_{new})\ \ etiquette(m) = etiquette(\mu(m)) \wedge ancetre(n, \mu(m))\}$
            $P_{new}.loc \leftarrow P_{new}.loc \cup NewOcc$
        Fin Pour
        $ER_1 \leftarrow ER_1 \cup P_{new}$
    Fin Pour
Fin Pour
Retourner ER_1

Algorithme 16 Accrochage

Entrée : P_i et P_1 accrochables
Sortie : un ensemble de patterns fréquents de profondeur $i+1$ résultant de l'accrochage de P_1 sur P_i

Pour tout $n \in noeuds\_accrochage(P_i, P_1)$ **Faire**
  $(LOC, Ancre_n(P_i, P_1)) \leftarrow CalculeLocAccr(P_i, P_1, n)$
  Si $|LOC| \geq \varepsilon$ **Alors**
    // L'accrochage sur ce nœud n est fréquent : on le traite
    $P_{new} \leftarrow P_i \oplus_n P_1$
    $P_{new}.loc \leftarrow \emptyset$
    Pour tout $(Occ_i, Occ_1) \in P_i.loc \times P_1.loc$ tq $(Occ_i.racine, Occ_1.racine) \in Ancre_n(P_i, P_1)$ **Faire**
      $NewOcc.racine \leftarrow Occ_i.racine$
      $NewOcc.mappings \leftarrow \emptyset$
      Pour tout $(\mu, \mu') \in Occ_i.mappings \times Occ_1.mappings$ tq $\mu(n) = Occ_1.racine$ **Faire**
        $NewOcc.mappings \leftarrow NewOcc.mappings \cup \mu \circ \mu'$
      Fin Pour
      $P_{new}.loc \leftarrow P_{new}.loc \cup NewOcc$
    Fin Pour
    $Result \leftarrow Result \cup P_{new}$
  Fin Si
Fin Pour
Retourner $Result$

Algorithme 17 CalculerLocAccr

Entrée : P_i et P_1 accrochables, $na \in noeuds\_accrochage(P_i, P_1)$
Sortie : LOC l'ensemble des occurrences dans N_D de $P_i \oplus_{na} P_1$, et $Ancre_{na}(P_i, P_1)$

$LOC \leftarrow \emptyset$
$Ancre_{na}(P_i, P_1) \leftarrow \emptyset$
Pour tout $Occ_i \in P_i.loc$ **Faire**
  $OccMappingOK \leftarrow FAUX$
  Pour tout $Occ_1 \in P_1.loc$ **Faire**
    Si $\exists \mu \in Occ_i.mappings$ tq $\mu(na) = Occ_1.racine$ **Alors**
      $OccMappingOK \leftarrow VRAI$
      $Ancre_{na}(P_i, P_1) \leftarrow Ancre_{na}(P_i, P_1) \cup \{(Occ_i.racine, Occ_1.racine)\}$
    Fin Si
  Fin Pour
  Si $OccMappingOK$ **Alors**
    $LOC \leftarrow LOC \cup \{Occ_i.racine\}$
  Fin Si
Fin Pour
Retourner LOC et $Ancre_{na}(P_i, P_1)$

2.5 Propriété de DRYAL

THÉORÈME 1 *L'algorithme* DRYAL *est correct et complet, c'est-à-dire que l'ensemble des patterns retournés par* DRYAL *pour une base de données d'arbres $\mathcal{BA}$ et un seuil ε est exactement l'ensemble $\mathcal{P}$ de tous les patterns fréquents de $\mathcal{BA}$ pour le seuil ε.*

Preuve : Nous allons commencer par montrer la correction, puis nous montrerons la complétude.

Correction :

Montrons par récurrence sur i que tout $P \in ER_i$ est fréquent, et que $\forall P \in ER_i$ la structure $P.loc$ représente bien l'ensemble des occurrences et des mappings de P dans les données.

- cas $i = 1$: Comme la procédure DécouvertePFA_1 est basée sur un algorithme de type APRIORI, que cet algorithme doit trouver tous les patterns de profondeur 1 ayant une racine d'un label donné, et qu'APRIORI est correct et complet, alors pour chaque label on est sûr d'obtenir tous les patterns de profondeur 1 dont la racine est ce label. On en déduit $ER_1 = PFA_1$.
  Pour chaque pattern fréquent découvert P_{new}, la structure de données $P_{new}.loc$ construite par les lignes 17 à 19 de la procédure DécouvertePFA_1 regroupe bien l'ensemble des occurrences (ligne 18) et des mappings (ligne 19) de P_{new} dans les données.
- cas $i + 1$. Supposons la propriété vraie au rang i, on va montrer qu'elle est toujours vraie au rang $i + 1$.
  Soit $P' \in ER_{i+1}$. D'après notre algorithme, il existe $P \in ER_i$ et $P_1 \in ER_1$ accrochables tels que $P' \in Accrochage(P, P_1)$, de plus P et P_1 sont fréquents par hypothèse de récurrence.
  Toujours en suivant notre algorithme, il existe au moins ε occurrences de P et P_1 où l'accrochage des deux dans les données est fréquent. Donc, par définition d'un arbre fréquent, P' est fréquent.
  Montrons maintenant que le $\cup_{Occ \in P.loc} Occ.racine = Locc(P', D)$, et $\cup_{Occ \in P.loc} Occ.mappings = \mathcal{EM}(P', D)$. Supposons par l'absurde que ce ne soit pas le cas.
  - 1er cas : il existe une occurrence qui n'est pas découverte. Or toutes les occurrences de P' proviennent de P, et par hypothèse de récurrence toutes les occurrences de P sont découvertes. Donc ce cas ne peut pas arriver.
  - 2ème cas : pour une occurrence r, il existe un mapping $\mu \in \mathcal{EM}_r(P, D)$ qui n'est pas découvert. On peut restreindre ce mapping μ sur les nœuds de P_i en μ_i, et sur les nœuds de P_1 en μ_1, avec $\mu = \mu_i \circ \mu_1$. Si μ n'est pas découvert, alors μ_i et μ_1 ne sont pas découverts non plus. Ce qui contredit l'hypothèse de récurrence.

Complétude :

Cette preuve s'effectue en deux temps :

- (i) On prouve d'abord que l'union des ensemble PFA_i de patterns fréquents par accrochage donne l'ensemble de tous les patterns fréquents (un pattern fréquent peut s'écrire comme une suite d'accrochages de patterns fréquents de PFA_1)
- (ii) On prouve ensuite que $\forall i \;\; ER_i = PFA_i$ (Rappelons que PFA_i est l'ensemble des patterns fréquents résultant de l'accrochage de i patterns de profondeur 1 de PFA_1).

Le point (i) est trivial : tout pattern peut s'écrire comme une suite d'accrochages de patterns de profondeur 1. Si un pattern est fréquent, alors tous les patterns qui le composent sont fréquents aussi, donc tous les patterns de profondeur 1 le composant sont bien dans PFA_1.

Pour le point (ii) : la correction a montré que $\forall i \;\; ER_i \subseteq PFA_i$. Il ne nous reste plus qu'à prouver que la réciproque est vraie aussi.

Par récurrence :

- Cas $i = 1$: on a vu plus haut que $ER_1 = PFA_1$

– Cas général : supposons qu'au rang i on aie $PFA_i \subseteq ER_i$. Montrons que ceci est vrai au rang $i+1$.
Soit $P_{i+1} \in E_{i+1}$ ($i \neq 0$). Il existe $P_i \in PFA_i$, $P_1 \in PFA_1$ tels que $P_{i+1} = P_i \oplus_n P_1$, avec $n \in N_{P_i}$.
Par hypothèse de récurrence, $P_i \in ER_i$ et $P_1 \in ER_1$. Ils sont accrochables sur n donc l'algorithme va dans la procédure *Accrochage*.
Comme $P_{i+1} \in PFA_{i+1}$, il est fréquent, donc $|LOC| \geq \varepsilon$.
D'où $P_{i+1} \in ER_{i+1}$. cqfd.

□

3 Paramétrages de Dryal

L'algorithme Dryal est facilement modifiable pour prendre en compte d'autres définitions d'inclusion que l'inclusion stricte faible utilisée dans la section précédente. Les sous-sections qui suivent montrent les modifications à faire pour utiliser d'autres définitions de l'inclusion.

3.1 Dryal$_{fils}$

La préservation de la relation d'ancestralité dans la définition d'inclusion stricte faible augmente de manière exponentielle le nombre de résultats à trouver. Le coût de l'algorithme avec cette définition est donc intrinsèquement très élevé.

On peut aisément restreindre Dryal à une définition d'inclusion plus contrainte, comme la relation d'inclusion stricte exacte faible (définition 9 du chapitre II), qui ne préserve que la relation de parenté directe.

Pour cela, les seuls changements à faire ont lieu au niveau de la procédure DécouvertePFA_1. Il faut en effet construire des patterns de profondeur 1 basés uniquement sur la relation de parenté directe. Ensuite, le mécanisme d'accrochage est le même.

Le changement dans la procédure DécouvertePFA_1 est au niveau de la ligne 7 : il suffit de remplacer $ancetre(n,m)$ par $parent(n,m)$.

Nous appellerons l'algorithme Dryal ainsi paramétré Dryal$_{fils}$.

3.2 Dryal$_{filsN}$

En s'inspirant du paramétrage précédent, on peut vouloir un compromis entre la souplesse de la relation d'inclusion utilisée et les performances du programme.

Étant donné un entier N, on va alors choisir une définition d'inclusion qui considère au plus N niveaux d'ancestralité. Par exemple $N = 1$ correspond à la relation de parenté directe et $N = 2$ considère les parents et les grand-parents. Là encore, le changement a lieu au niveau de la ligne 7 de DécouvertePFA_1 : on remplace $ancetre(n,m)$ par $ancetre_{\leq N}(n,m)$ où $ancetre_{\leq N}$ considère toutes les relations d'ancestralité d'au plus N niveaux.

3.3 Dryal$_{injectif}$

Nous pouvons aussi désirer rajouter une autre contrainte sur la définition d'inclusion que nous utilisons, à savoir la contrainte d'injectivité. Les définitions d'inclusion que nous utiliserons alors ne seront plus des définitions "faibles" au sens de la définition 9 du chapitre II.

L'intérêt de considérer des définitions d'inclusion fondées sur des mappings injectifs est double :

- Les patterns obtenus auront exactement la forme qu'ils ont dans les documents. Sans l'injectivité, on autorise en effet certaines formes des patterns qui peuvent troubler l'intuition de l'utilisateur, dès lors que la définition d'inclusion n'est pas exacte (c'est-à-dire utilise des relations d'ancestralité).
  Dans notre exemple, considérons les patterns fréquents de PFA_2 représentés sur la figure V.5. Au sens d'une définition faible de l'inclusion, avec un seuil $\varepsilon = 2$, le pattern P_{24} est fréquent. Toutefois, cela implique que pour l'occurrence de P_{24} de racine n_1 dans les données, les deux nœuds de P_{24} d'étiquette D soient "mappés" sur le nœud n_3 des donnés (voir figure V.2). Ceci n'est pas forcément très intuitif, et un tel pattern ne serait pas considéré comme fréquent avec une définition d'inclusion prenant en compte l'injectivité.
- Tenir compte de l'injectivité des mappings augmente certains coûts du programme, mais permet d'en diminuer d'autres, en diminuant le nombre de patterns fréquents redondants générés. Considérons l'exemple de la figure V.7. On a en haut une base de données d'arbre, et nous avons représenté quelques patterns fréquents de profondeur 1, pour $\varepsilon = 2$. La fréquence du pattern P_{21}, construit par accrochage de P_{12} sur P_{11}, repose sur le mapping des deux nœuds d'étiquette D de P_{21} sur un même nœud d'étiquette D dans les données. Le fait que P_{21} ait deux nœuds d'étiquette D va rendre possible l'accrochage sur ces deux nœuds du pattern P_{13} et générer les deux patterns fréquents P_{31} et P_{32}. Chacun de ces patterns devant être considéré dans les calculs ultérieurs, le surcoût est important. Si par contre on impose l'injectivité des mappings, on supprime une des occurrences du nœud d'étiquette D dans P_{21}, et donc un seul accrochage de P_{13} est possible.

Les nœuds redondants introduits dans les patterns proviennent de l'utilisation de la relation d'ancestralité. Par exemple, dans la figure V.8, on peut voir une base de donnés d'arbres et tous les patterns fréquents de profondeur 1 associés, pour $\varepsilon = 2$ (on utilise ici la version de DRYAL exploitant l'ancestralité). Si on accroche le pattern 9 sur le pattern 6, puis le pattern 11 sur le pattern ainsi obtenu, on obtient P_C, très proche des données. Mais on peut aussi accrocher le pattern 9 sur le pattern 3. On voit que le nœud d'étiquette C est dupliqué : il a une version qui provient du pattern 3 et qui représente une relation d'ancestralité dans le pattern, et une version qui provient du pattern 9 et qui ici est à sa place en profondeur 2.

Pour que DRYAL se limite à faire les accrochages qui satisfont l'injectivité des mappings correspondants, il faut donc éviter d'effectuer tous les accrochages qui produiraient des nœuds redondants.

Pour cela, on introduit la fonction $VerifieInjectivite$, décrite dans l'algorithme 18. Étant donné deux patterns P_i et P_1 accrochables, un nœud d'accrochage na et les ensembles $Ancre_{na}(P_i, P_1)$ et LOC calculés par $CalculerLocAccr$, cette fonction renvoie l'ensemble $NewLOC$ des occurrences de LOC vérifiant la contrainte d'injectivité pour l'accrochage $P_i \oplus_{na} P_1$.

Le principe est le suivant : pour chaque occurrence de P_i où s'accroche P_1 sur na, on cherche s'il existe un mapping de $P_i \oplus_{na} P_1$ dans les données qui vérifie la contrainte d'injectivité. Comme à ce point de l'algorithme on ne dispose pas des mappings de $P_i \oplus_{na} P_1$, notre analyse se fonde sur les mappings de P_i et ceux de P_1. Pour une feuille donnée de P_i, si dans $P_i \oplus_{hn} P_1$ cette feuille ne fait que refléter une relation d'ancestralité entre son père et une feuille provenant de P_1 (non respect de la contrainte d'injectivité), cela se traduit dans les données par une égalité entre l'ensemble des nœuds de D atteignables par mapping à partir de la feuille de P_i, et l'ensemble des nœuds de D atteignables par mapping à partir de la feuille correspondante de P_1.

Il ne reste qu'à modifier la procédure $Accrochage$ (algorithme 16) pour utiliser $VerifieInjectivite$.

On rajoute une ligne 3' : $LOC \leftarrow VerifieInjectivite(P_i, P_1, na, Ancre_{na}(P_i, P_1), LOC)$.

Cette ligne assure que seuls les accrochages vérifiant la condition d'injectivité pour au moins ε de leurs occurrences seront effectués.

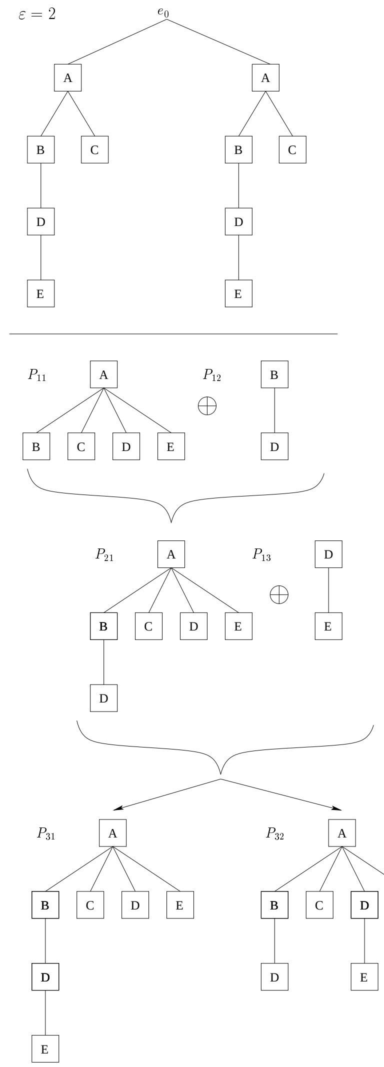


Fig. V.7 – P_{31} et P_{32} sont redondants

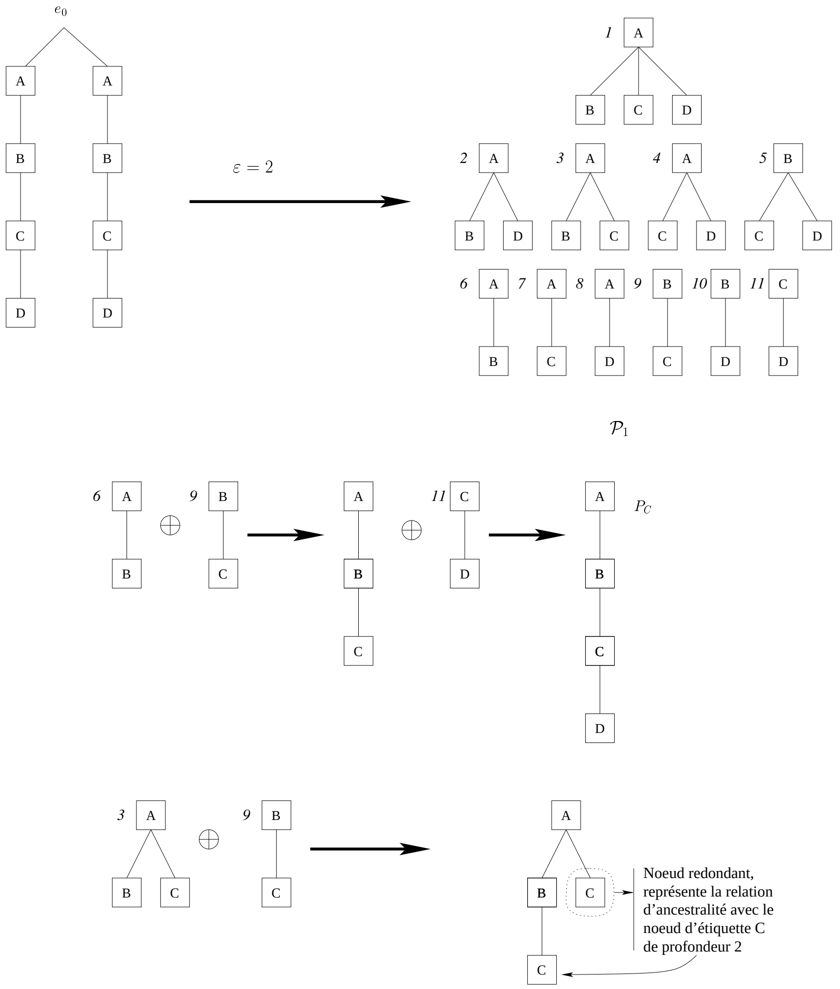


Fig. V.8 – Une base de données d'arbres et tous les patterns fréquents de profondeur 1 associés

Algorithme 18 $VerifieInjectivite$

Entrée : P_i et P_1 accrochables, na nœud d'accrochage, $Ancre_{na}(P_i, P_1)$, LOC ensemble d'occurrences de P_i où s'accroche P_1

Sortie : $NewLoc$ la liste des occurrences de LOC vérifiant la contrainte d'injectivité.

$NewLOC \leftarrow \emptyset$
Pour tout $n_i \in LOC$ **Faire**
    $ToutesFeuillesOK \leftarrow VRAI$
    Pour tout feuille $f \in N_{P_i}$ **Faire**
        $f' \leftarrow$ feuille de P_1 de même label que f
        $CetteFeuilleOK \leftarrow FAUX$
        Pour tout $\mu \in \mathcal{EM}_{n_i}(P_i, D)$ **Faire**
            Pour tout $\mu' \in \mathcal{EM}_{n_1}(P_1, D)$ avec $(n_i, n_1) \in Ancre_{na}(P_i, P_1)$ **Faire**
                Si $\mu'(f') \subset \mu(f)$ **Alors**
                    $CetteFeuilleOK \leftarrow VRAI$
                Fin Si
            Fin Pour
        Fin Pour
        $ToutesFeuillesOK \leftarrow ToutesFeuillesOK\ ET\ CetteFeuilleOK$
    Fin Pour
    Si $ToutesFeuillesOK$ **Alors**
        $NewLOC \leftarrow NewLOC \cup \{n_i\}$
    Fin Si
Fin Pour
Retourner $NewLoc$

4 DRYADE

4.1 Motivations

L'algorithme précédent, DRYAL, n'exploite pas le fait que si un pattern A est fréquent, alors nécessairement tous les patterns inclus dans A seront fréquents (comme le montre la figure V.9). Pourquoi alors consommer du temps de calcul pour générer tous ces patterns fréquents inclus dans A ?

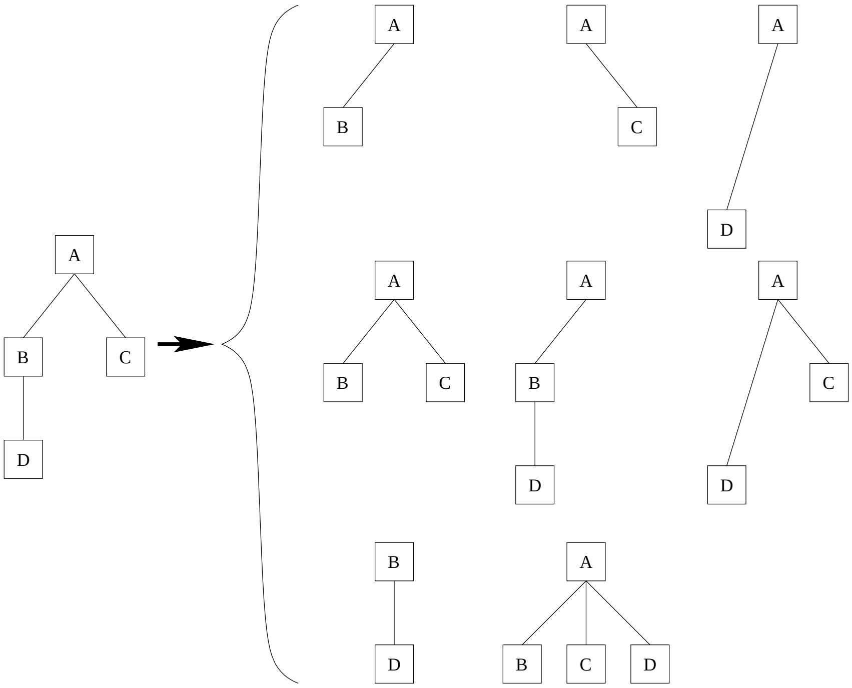


FIG. V.9 – Si le pattern de gauche est fréquent, tous les patterns de droite sont fréquents aussi.

Ce défaut est commun à tous les algorithmes de découverte de motifs fréquents. Il est particulièrement important dans notre problème, car la définition d'inclusion basée sur une relation d'ancestralité augmente exponentiellement la quantité des patterns fréquents que l'on peut découvrir dans les données.

Dans le cas classique des bases de données transactionnelles, nous avons vu dans le chapitre III qu'il existait une solution pour s'affranchir des nombreuses redondances existant dans les résultats. Cette solution est de calculer des itemsets fréquents **fermés**.

Nous allons donc modifier notre algorithme pour pouvoir trouver non plus des patterns fréquents, mais des patterns fréquents **fermés**, dont nous donnerons la définition exacte dans la section suivante. À notre connaissance, cette solution n'a jamais été appliquée à la découverte de patterns d'arbre fréquents. Ce travail sur les patterns d'arbre fréquents fermés est donc une de nos contributions principales.

Pour toute la suite de ce chapitre, nous utiliserons la définition d'inclusion injective et ne préservant que la relation d'ancestralité (non bornée). Nous avons présenté dans la section précédente l'algorithme DRYAL$_{injectif}$ qui calcule l'ensemble de tous les patterns fréquents correspondant à cette définition d'inclusion, ainsi que l'intérêt d'utiliser cette définition. Dans cette section, nous présentons DRYADE, qui calcule l'ensemble des patterns fréquents **fermés** correspondant à cette définition.

4.2 Nouvelles définitions

Nous définissons formellement la notion d'ensemble fermé de patterns fréquents. L'intuition de notre définition est que pour qu'un pattern fréquent P' soit fermé, il ne faut pas que ses occurrences dans le datatree soient les mêmes que ses occurrences dans le projeté d'un pattern P dans lequel il est inclus. En effet dans ce cas le pattern P' est redondant par rapport à P : il est inclus dans P et il n'a pas d'occurrences dans les données autres que celles induites par son inclusion dans P.

Définition 32 (Ensemble fermé de patterns fréquents) *Soit E un ensemble de patterns fréquents avec un seuil ε. On dit que E est fermé si pour tout pattern fréquent P de E, il n'existe pas de pattern fréquent $P' \in E$ tel que :*

i. $P' \sqsubset P$

ii. $Locc(P', D) = Locc(P', projete(P, D))$

Notation : Par extension, nous dirons d'un pattern fréquent d'un ensemble fermé de patterns fréquents que c'est un pattern fréquent **fermé**.

Exemple : Pour notre datatree d'exemple D_X (figure V.2), considérons PFA_1 l'ensemble des patterns fréquents de profondeur 1. La figure V.10 montre les ensemble d'occurrences de ces patterns, puis l'ensemble $Ferme(PFA_1)$ que l'on peut en déduire.

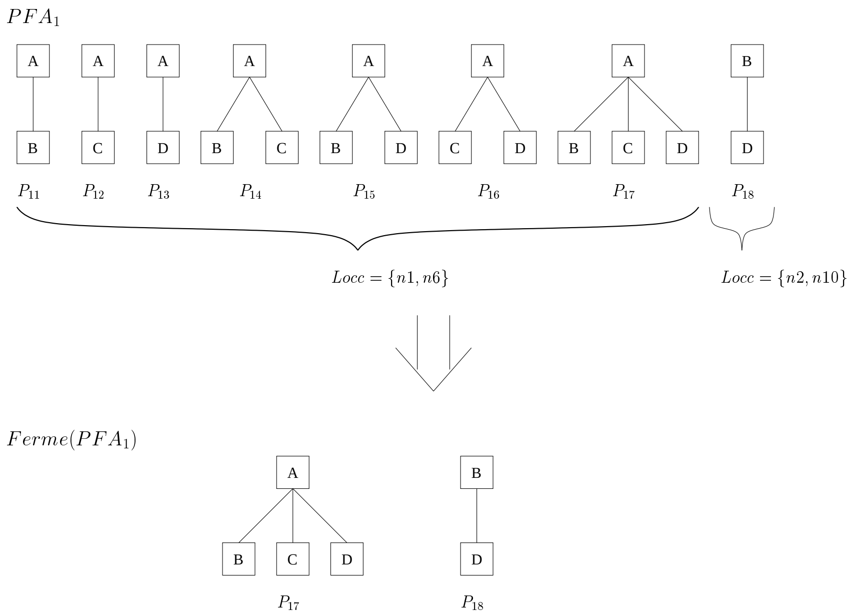


Fig. V.10 – Pour D_X, PFA_1 et $Ferme(PFA_1)$

Comme on peut le voir, à cause de la condition **i**, quand un pattern fréquent est inclus dans un autre il a de fortes chances d'être éliminé de l'ensemble fermé.

On ne garde un pattern fréquent P' inclus dans un pattern fréquent P que lorsque la condition **ii** n'est pas vérifiée. Ce cas est illustré dans la figure V.11 : le pattern fréquent P_2 est inclus dans le pattern fréquent P_1, toutefois il est fermé car son occurrence n_2 n'apparaît pas dans le projeté de P_1 dans les données, qui est l'arbre de racine e_0 privé du sous-arbre de racine n_2.

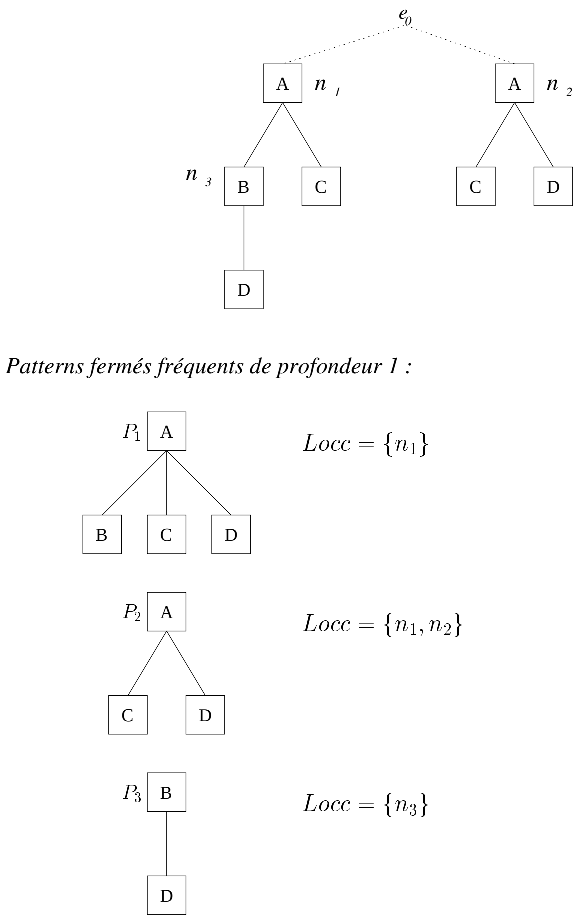


FIG. V.11 – $\varepsilon = 1$

Définition 33 (Fermeture d'un ensemble de patterns fréquents) *Soit E un ensemble de patterns fréquents. On appelle $Ferme(E)$ le plus grand ensemble (au sens de l'inclusion ensembliste) inclus dans E tel que $Ferme(E)$ soit un ensemble fermé de patterns fréquents.*

Désormais, nous ne cherchons plus l'ensemble de tous les patterns fréquents $\mathcal{P}$, mais le plus grand ensemble fermé de patterns fréquents inclus dans cet ensemble, c'est-à-dire que nous recherchons $Ferme(\mathcal{P})$. Les patterns **cible** seront donc désormais les patterns de $Ferme(\mathcal{P})$.

Une solution "simple" pour trouver $Ferme(\mathcal{P})$ est de calculer $\mathcal{P}$, puis d'en déduire $Ferme(\mathcal{P})$. Cette solution est contre-productive : on doit payer le très long temps de calcul nécessaire à la découverte de $\mathcal{P}$, puis un temps supplémentaire pour ne garder que les patterns fermés.

L'idée principale de l'optimisation par les fermés est d'éviter tous les calculs redondants. On va donc directement calculer $Ferme(\mathcal{P})$. Pour cela, nous continuerons à appliquer notre stratégie d'accrochages, mais avec des ensembles de patterns fréquents beaucoup plus réduits. C'est-à-dire que nous allons imposer que l'ensemble des patterns fréquents maintenu par l'algorithme soit **toujours un ensemble fermé.**

Pour illustrer l'avantage de cette méthode, considérons l'exemple de la figure V.12.

Pour la base de données d'arbres en haut de cette figure, nous avons représenté à droite l'ensemble $\mathcal{P}_1$ des patterns fréquents de profondeur 1 qu'il contient, et à gauche l'ensemble $Ferme(\mathcal{P}_1)$. Tous les patterns sont numérotés.

A gauche, le nombre d'accrochages que l'on peut faire est très limité :
- P_2 et P_3 sont accrochables sur P_1
- P_3 est accrochable sur P_2

Pour pouvoir accrocher uniquement des patterns de profondeur 1 sur des patterns de profondeur quelconque, on ne peut accrocher que P_2 sur P_1, puis P_3 sur $P_1 \oplus P_2$. On génère donc un nombre minimal de patterns fréquents. Toutefois, pour obtenir l'unique pattern de $Ferme(\mathcal{P})$, il faut éliminer des patterns fréquents produits les feuilles "en trop" qui ne respectent pas la contrainte d'injectivité (voir figure V.13).

Par contre, à droite dans la figure V.12, un grand nombre d'accrochages peuvent être effectués :
- P_{12} peut s'accrocher sur P_5
- P_{14} peut s'accrocher sur P_6
- P_8, P_{12}, P_{13} peuvent s'accrocher sur P_9
- P_{14} peut s'accrocher sur P_{10} et P_{12}

Pour tous ces accrochages possibles, des patterns fréquents vont être produits. Ces patterns fréquents sont corrects, mais sont tous inclus dans l'unique pattern de $Ferme(\mathcal{P})$, qui lui résulte simplement de l'accrochage de P_{12} sur P_9, puis de P_{14} sur le pattern précédent. Les générer tous n'est donc pas utile.

Nous pouvons donc voir que l'ensemble fermé des patterns fréquents de profondeur 1 étant très restreint, il évite de générer des patterns fréquents "inutiles", et donc de faire des calculs redondants. Ce principe est généralisable aux ensemble fermés de patterns fréquents de profondeur quelconque.

Toutefois, les accrochages sur des patterns fermés fréquents posent un problème non trivial pour respecter la contrainte d'injectivité. Pour bien comprendre ce problème et son origine, ainsi que la manière dont nous allons le résoudre, le paragraphe suivant donne des explications détaillées sur l'ensemble fermé des patterns fréquents de profondeur 1.

4.2.1 Ensemble fermé des patterns fréquents de profondeur 1

Reprenons l'exemple de la figure V.12. Soit P_C l'unique pattern fréquent de $Ferme(\mathcal{P})$.

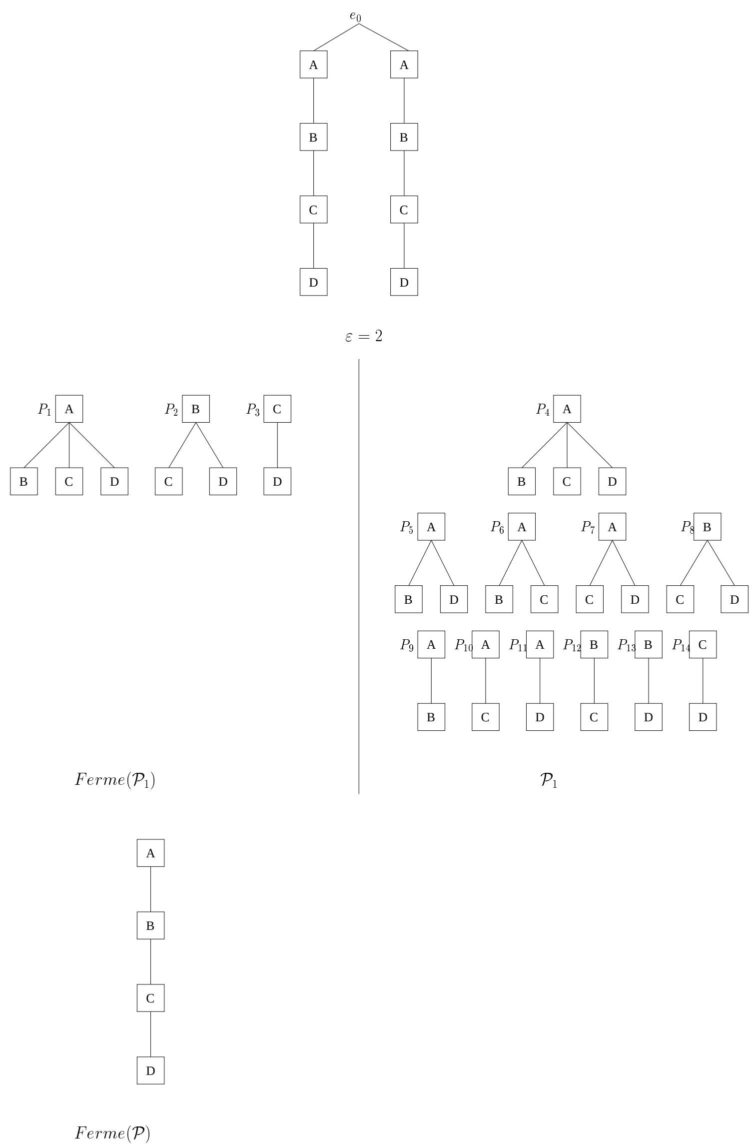


Fig. V.12 – Comparaison de $Ferme(\mathcal{P}_1)$ et $\mathcal{P}_1$

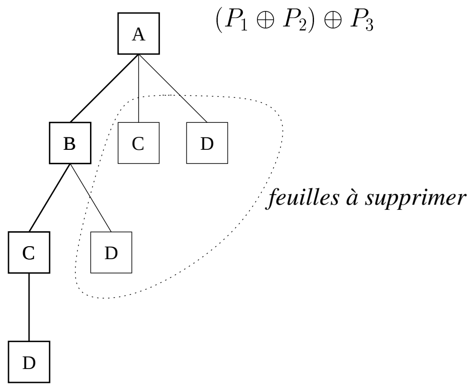


Fig. V.13 – Pattern produit par l'accrochage de P_2 sur P_1 puis P_3 sur $P_1 \oplus P_2$.

Intéressons nous plus particulièrement aux patterns fréquents de $\mathcal{P}_1$ dont la racine porte l'étiquette A. On trouve les patterns fréquents P_4, P_5, P_6, P_7, P_9, P_{10} et P_{11}. Tous ces patterns ont le même ensemble d'occurrences. Les patterns P_5, P_6, P_7, P_9, P_{10} et P_{11} représentent toutes les façons possibles de retirer un puis deux nœuds au pattern P_4. Ils sont donc redondants, car à partir de P_4 il est très facile de les retrouver.

Sans surprise, le seul pattern dont la racine porte l'étiquette A dans $Ferme(\mathcal{P}_1)$ est donc le pattern P_1, qui correspond au pattern P_4 de $\mathcal{P}_1$. **Les feuilles de ce pattern correspondent à toutes les relations d'ancestralité entre la racine de P_C et les autres nœuds de P_C.**

Comprendre ceci est capital pour comprendre le reste de l'algorithme Dryade. Nous pouvons le formaliser comme suit :

Définition 34 (Applatissement) *Soit P un pattern. On appelle* ***applatissement*** *de P, noté $plat(P)$, l'unique pattern de profondeur 1 tel qu'il existe un mapping μ surjectif entre les nœuds de P et les nœuds de $plat(P)$ vérifiant :*
- $\mu(racine(P)) = racine(plat(P))$
- $\forall n \in N_P \;\; label(n) = label(\mu(n))$
- $\forall n \in N_P$ *tel que* $ancetre(racine(P), n)$: $parent(racine(plat(P)), \mu(n))$

Propriété 8 *Pour tout pattern fréquent $P \in Ferme(\mathcal{P})$, $plat(P) \in Ferme(\mathcal{P}_1)$*

Preuve : Par définition des patterns fréquents de profondeur 1, le pattern $plat(P)$ existe dans $\mathcal{P}_1$: c'est le pattern de profondeur 1 construit avec la racine de P et comme feuilles exactement toutes les étiquettes des nœuds de P, sans redondances. Ce pattern est gardé dans $Ferme(\mathcal{P})$ car tous les patterns de même racine et même ensemble d'occurrences mais avec moins de nœuds sont éliminés par lui lors du calcul des fermés, par contre lui ne peut être éliminé par personne. Il est en effet maximal au sens de l'inclusion des patterns. □

La propriété 8 permet de bien comprendre la raison des problèmes de la satisfaction de la contrainte d'injectivité. Étant donné qu'un pattern fréquent de profondeur 1 contient toutes les relations d'ancestralité des patterns cibles qu'il va permettre de construire, lorsqu'on lui accroche un autre pattern de profondeur 1, les relations avec les nœuds de cet autre pattern se retrouvent en double : dans les nœuds du pattern accroché et dans les nœuds du pattern sur lequel on fait l'accrochage.

Nous avons déjà rencontré ce problème dans la section 3.3. Pour le résoudre, nous détections les accrochages illicites du point de vue de la contrainte d'injectivité, afin de ne pas les effectuer.

Mais ici nous ne pouvons pas mettre en oeuvre ce type de solution : nous avons vu que les patterns de profondeur 1 dont nous disposons entraînent **toujours** un non-respect de la contrainte d'injectivité lors de l'accrochage.

Il nous faut donc effectuer l'accrochage, puis éliminer les feuilles redondantes dans les patterns fréquents produits, afin d'obtenir des patterns fréquents fermés.

Les feuilles redondantes se définissent comme suit :

Définition 35 (Feuille redondante) *Soient P_i et P_1 deux patterns fréquents fermés accrochables, et soit $na \in noeuds\_accrochage(P_i, P_1)$. Soit $P_{i+1} = P_i \oplus_{na} P_1$.*

On dit qu'une feuille F de P_{i+1}, provenant de P_i, est ***redondante*** *si pour toute occurrence $o \in Locc(P_{i+1}, D)$ et pour tout mapping $\mu \in \mathcal{EM}_o(P_{i+1}, D)$, on a $\mu(F) = \mu(F_1)$, où F_1 est la feuille de P_{i+1} issue de P_1 de même étiquette que F.*

De la définition précédente nous déduisons que pour obtenir des patterns fréquents fermés à partir de l'accrochage de deux patterns fréquents fermés, il faut d'abord effectuer l'accrochage des patterns fréquents fermés, puis ensuite supprimer les feuilles redondantes dans le pattern obtenu, les feuilles à supprimer variant suivant les occurrences.

Par exemple dans la figure V.13, les feuilles à supprimer sont entourées en pointillés.

4.3 Principe de l'algorithme

4.3.1 Principe général

L'idée sur laquelle est fondée l'algorithme DRYADE est de créer l'ensemble fermé de tous les patterns fréquents, en commençant par créer l'ensemble fermé des patterns fréquents de profondeur 1 ($Ferme(\mathcal{P}_1)$), puis en créant par accrochages successifs les patterns fréquents fermés de profondeur supérieure.

Dès que l'on obtient $Ferme(\mathcal{P}_1)$, on a une approximation très grossière du résultat final, $Ferme(\mathcal{P})$. Dans les patterns de $Ferme(\mathcal{P}_1)$ se trouvent en effet les applatissements des patterns de $Ferme(\mathcal{P})$. Les informations sur la structure vont être gagnées progressivement par l'accrochage de patterns de profondeur 1, et la suppression des feuilles redondantes.

4.3.2 Différences par rapport au principe de DRYAL

À cause de la recherche d'un ensemble **fermé** de patterns fréquents, la stratégie d'accrochage que nous utilisions dans DRYAL va devoir être entièrement revue, pour trois raisons :

Suppression des feuilles : Chaque accrochage effectué devra être suivi d'une étape de suppression des feuilles redondantes dans le pattern fréquent obtenu, en vue d'obtenir un ou plusieurs patterns fréquents fermés. Nous reviendrons plus tard sur la mise en oeuvre exacte de cette étape.

Accrochages multiples : Dans DRYAL, nous accrochions les patterns de profondeur 1 un par un. Cela n'est pas possible pour la recherche de patterns fermés. Par exemple, considérons le nouveau datatree exemble D_Y, représentée sur la figure V.14. Pour un seuil $\varepsilon = 2$, nous avons représenté sur la figure V.15 l'ensemble $\mathcal{F}_1 = Ferme(\mathcal{P}_1)$.

Si on suit la méthode de DRYAL qui accroche les patterns de profondeur 1 un par un, on obtient les accrochages de la figure V.16.

On peut voir deux problèmes. Le premier est que l'on génère deux fois le pattern P_2. Cela est inefficace. Le deuxième problème est encore plus grave : on génère les patterns P_{2a} et P_{2b}, alors que ceux-ci ne satisfont pas la condition de fermeture (ils sont tous les deux strictement inclus dans le pattern P_2, avec le même ensemble d'occurrences).

Pour résoudre ce problème efficacement, nous allons devoir, pour un pattern fréquent fermé P, lui accrocher en même temps plusieurs patterns fréquents fermés de profondeur 1. Dans le cas de la figure V.16, il faudra accrocher en même temps P_{12} et P_{13} sur P_{11}.

Ordre des accrochages : Le respect de la condition de fermeture nous impose une troisième divergence par rapport à DRYAL. Revenons encore une fois à l'exemple de la figure V.12. Nous pourrions accrocher P_2 à P_1, et aussi P_3 à P_2 pour faire un autre pattern. Sur $P_1 \oplus P_2$, nous pourrions ensuite accrocher P_3. Après les suppressions de feuilles adéquates, on obtiendrait deux patterns, le pattern de $Ferme(\mathcal{P})$ $(P_1 \oplus P_2) \oplus P_3 : A - B - C - D$, et aussi un pattern $P_2 \oplus P_3 : B - C - D$. Or ce dernier pattern est fréquent mais pas fermé.

Pour éviter ce type de problèmes, nous devons respecter un certain ordre pour accrocher les patterns les uns sur les autres, en prenant comme règle de retarder au maximum les accrochages tant qu'ils sont possibles.

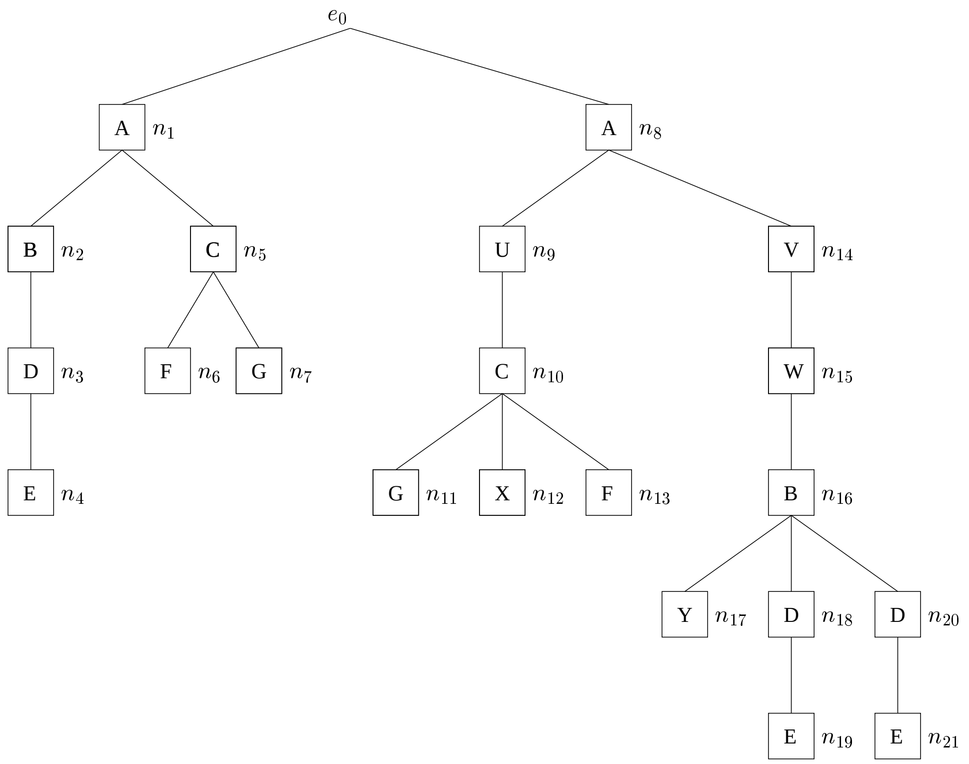


Fig. V.14 – D_Y

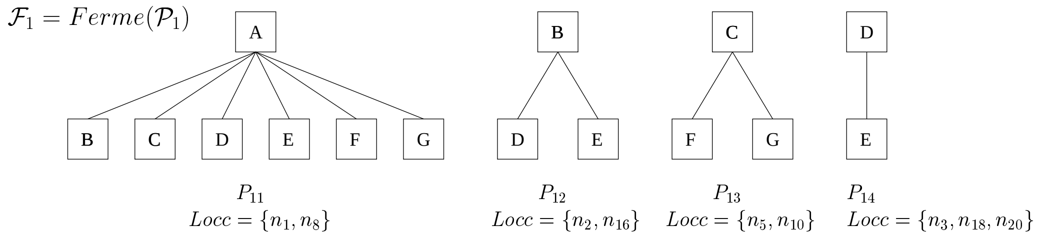


Fig. V.15 – Pour $\varepsilon = 2$, ensemble fermé des patterns fréquents de profondeur 1 de D_Y

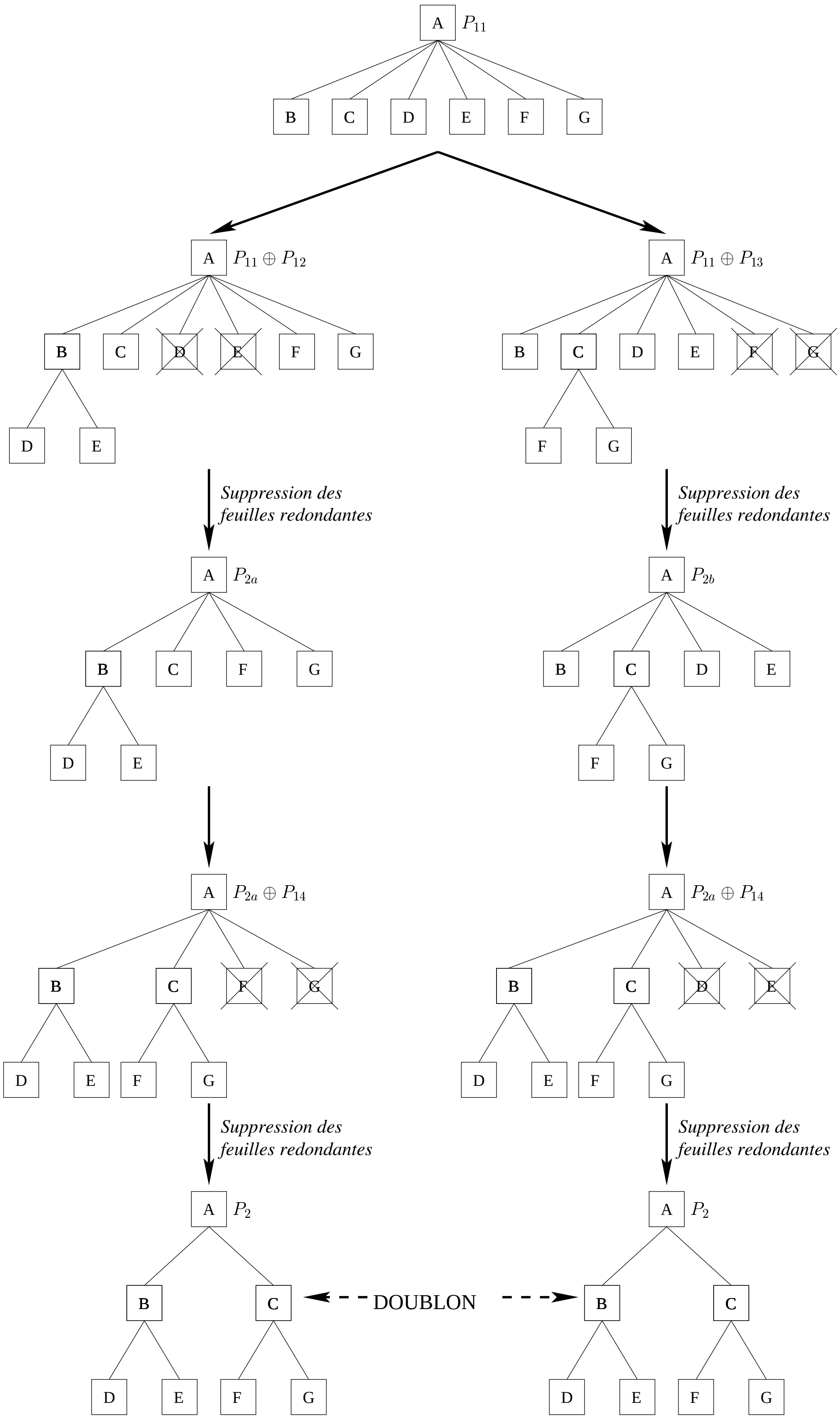


Fig. V.16 – Accrochages avec la méthode de Dryal

A chaque itération de DRYADE, les patterns sur lesquels nous devons accrocher d'autres patterns seront appelés des patterns **racine**. Pour les définir, nous rendons explicite la structure de graphe des accrochages existant entre les patterns, ce qui nous permet de faire apparaître une "stratification" entre les patterns.

Définition 36 (Graphe des accrochages) *Soit EP un ensemble de patterns. Nous appellerons **graphe des accrochages** le graphe orienté étiqueté $\mathcal{G}_{EP} = (EP, \mathcal{A})$ tel que :*
- *l'ensemble des nœuds du graphe est EP*
- *Il existe un arc de P_1 vers P_2 (c'est-à-dire $(P_1, P_2) \in \mathcal{A}$) si $Ancre(P_2, P_1) \neq \emptyset$. Dans ce cas, l'arc (P_1, P_2) est étiqueté par l'ensemble $Ancre(P_2, P_1)$.*

Exemple : Avec le datatree D_Y, le graphe des accrochages des patterns fréquents fermés de $\mathcal{F}_1$ (figure V.15) est représenté sur la figure V.17.

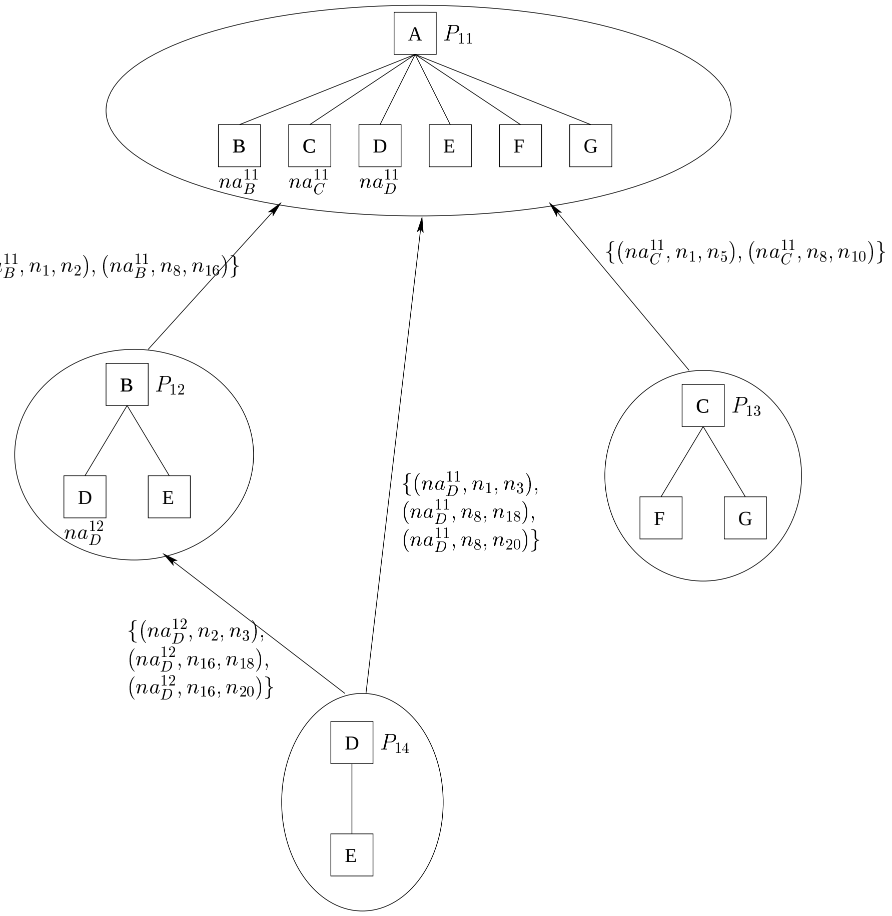


FIG. V.17 – Le graphe des accrochages de $\mathcal{F}_1$ ($\mathcal{F}_1$ est représenté figure V.15)

Définition 37 (Prédécesseurs et successeurs) *Soit EP un ensemble de patterns. Soit $\mathcal{G}_{EP} = (EP, \mathcal{A})$ le graphe des accrochages de EP. Pour tout pattern $P \in EP$, on notera :*
- *$pred(P) = \{P' \mid (P', P) \in \mathcal{A}\}$, ce sont donc tous les patterns qui s'accrochent sur P.*
- *$succ(P) = \{P' \mid (P, P') \in \mathcal{A}\}$, ce sont donc tous les patterns sur lesquels P s'accroche.*

Définition 38 (Classe d'équivalence de cycle) *Soit EP un ensemble de patterns, et $\mathcal{G}_{EP}$ le graphe des accrochages associé. Les* ***classes d'équivalence de cycle*** *de EP sont les classes d'équivalence de la relation d'équivalence $\equiv$ définie sur les nœuds de $\mathcal{G}_{EP}$ de la façon suivante :*

$N_1 \equiv N_2$ ssi il existe un cycle dans $\mathcal{G}_{EP}$ passant par N_1 et N_2

L'ensemble des classes d'équivalences de cycle de EP est noté $\mathcal{CE}(EP)$.

Définition 39 (Graphe quotient des accrochages) *Soit EP un ensemble de patterns. Nous appellerons* ***graphe quotient des accrochages*** *de EP, noté $\mathcal{GQ}_{EP} = (\mathcal{CE}(EP), \mathcal{A}')$, le graphe vérifiant :*

- *l'ensemble des nœuds du graphe est $\mathcal{CE}(EP)$.*
- *il existe un arc entre deux nœuds CE_1 et CE_2 de $\mathcal{CE}(EP)$ si et seulement si il existe un arc dans $\mathcal{G}_{EP}$ entre un pattern élément de CE_1 et un pattern élément de CE_2.*

Propriété 9 *Pour un ensemble de patterns EP, son graphe quotient des accrochages est un graphe orienté acyclique. Il existe donc des nœuds de $\mathcal{GQ}_{EP}$ qui n'ont pas de père, et qui seront appelés les racines du graphe quotient des accrochages.*

Définition 40 (Pattern racine) *Soit EP un ensemble de patterns, soit $\mathcal{GQ}_{EP}$ son graphe quotient. Un* ***pattern racine*** *de EP est un pattern qui :*

- *appartient à une classe d'équivalence racine du graphe quotient des accrochages*
- *et qui a au moins ε* ***occurrences de racine*** *dans les données.*

L'ensemble des ***occurrences de racine*** *d'un pattern $P \in EP$ est l'ensemble $OR(P)$ défini par :*

$$OR(P) = \{o \in Locc(P, D) \mid \forall P' \in succ(P) \quad \nexists o' \in Locc(P', D) \ tq \ (o', o) \in Ancre(P', P)\}$$

Remarque 5 *Si le graphe des accrochage d'un ensemble de patterns n'a pas de cycles, alors les patterns racines sont les racines du graphe d'accrochage.*

4.3.3 Pseudo-code et détails de l'algorithme DRYADE

DRYADE est exposé dans l'algorithme 19. Les notations utilisées sont résumées dans le tableau V.2.

$\mathcal{F}_D^i$	Ensemble fermé des patterns fréquents en cours de construction à l'itération i
$\mathcal{G}$	Graphe des accrochages de $\mathcal{F}_D^i$
EPR	Ensemble des patterns racines de $\mathcal{F}_D^i$
$ECAI$	Ensemble des contextes d'accrochage immédiat sur un pattern racine
$NewPatternsInfos$	Nouveaux patterns fréquents de profondeur $i+1$ et les patterns dont ils proviennent par accrochage.

TAB. V.2 – Notations pour l'algorithme DRYADE

Nous allons dans les paragraphes suivants expliquer point par point le fonctionnement de DRYADE.

Déterminer $Ferme(\mathcal{P}_1)$ et $\mathcal{G}$

Nous avons vu dans l'algorithme 15 (DécouvertePFA_1) une méthode pour déterminer l'ensemble $\mathcal{P}_1$ des patterns fréquents de profondeur 1. Pour que cet algorithme calcule $Ferme(\mathcal{P}_1)$, il suffit d'une

Algorithme 19 DRYADE

Calculer $\mathcal{F}_D^1 = Ferme(\mathcal{P}_1)$
$\mathcal{G} \leftarrow CalculeGraphe(\mathcal{F}_D^1)$ // *Voir algorithme 20*
$Continuer \leftarrow VRAI$
$i \leftarrow 1$
Tant que $Continuer$ **Faire**
  $NewPatternsInfos \leftarrow \emptyset$
  $Continuer \leftarrow FAUX$
  $EPR \leftarrow DonneRacines(\mathcal{G})$ // *Voir algorithme 21*
  $\mathcal{F}_D^{i+1} \leftarrow \mathcal{F}_D^i$
  Pour tout $PR \in EPR$ **Faire**
    $ECAI \leftarrow PatternsAAccrocher(PR)$ // *Voir algorithme 23*
    Si $ECAI \neq \emptyset$ **Alors**
      $Continue \leftarrow VRAI$
    Fin Si
    $(NewP, NewPInfos) \leftarrow AccrochageSuppr(PR, ECAI)$ // *Voir algorithme 27*
    $\mathcal{F}_D^{i+1} \leftarrow \mathcal{F}_D^{i+1} \cup NewP$
    $NewPatternsInfos \leftarrow NewPatternsInfos \cup NewPInfos$
  Fin Pour
  $MiseAJour(\mathcal{F}_D^{i+1}, \mathcal{G}, NewPatternsInfos)$ // *Voir algorithme 29*
  $i \leftarrow i + 1$
Fin Tant que
Retourner $\mathcal{F}_D^i$

modification mineure. Pour calculer l'ensemble des ensembles de feuilles de profondeur 1 fréquentes, DécouvertePFA_1 utilise un algorithme de type APRIORI exhaustif, en l'occurrence ECLAT. Comme récapitulé dans le tableau V.3, les transactions sont tous les nœuds n des données d'étiquette l, avec l fixé, et les items sont toutes les étiquettes des nœuds descendants de n. Dans le cadre de DRYADE, nous sommes intéressés par l'ensemble **fermé** des ensembles de feuilles de profondeur 1. Il faut donc remplacer ECLAT par un algorithme calculant des itemsets fréquents fermés. Pour avoir facilement accès aux ensembles-support, le nouvel algorithme doit être vertical : l'algorithme CHARM [Zaki et Hsiao 2002] est donc un bon candidat.

Transactions	Items
nœud n de $\mathcal{BA}$ d'étiquette l	étiquettes des nœuds descendant de n

TAB. V.3 – Frequent Item Sets pour la détection des patterns fréquents de profondeur 1 dont la racine est d'étiquette l, pour une base de données d'arbres $\mathcal{BA}$

Propriété 10 *Soit $\mathcal{F}_D^1$ le résultat de l'algorithme DécouvertePFA_1 où l'on remplace* ECLAT *par* CHARM. *Alors $\mathcal{F}_D^1 = Ferme(\mathcal{P}_1)$.*

Preuve : Nous avons vu dans la section précédente que DécouvertePFA_1 calcule bien $\mathcal{P}_1$ avec l'algorithme ECLAT. La fermeture de $Ferme(\mathcal{P}_1)$ dépend de l'inclusion entre éléments de $\mathcal{P}_1$. Deux patterns de $\mathcal{P}_1$ de racines différentes ne peuvent être inclus l'un dans l'autre. Deux éléments de $\mathcal{P}_1$ de même racine sont inclus l'un dans l'autre si leurs ensembles de feuilles sont inclus par l'inclusion ensembliste. On en déduit que dans $\mathcal{F}_1 = Ferme(\mathcal{P}_1)$, pour les patterns de même étiquettes de racines les ensembles de feuilles sont fermés.

Or pour une racine d'étiquette donnée, ce sont précisément les ensembles de feuilles qui sont calculés par les algorithmes ECLAT ou CHARM. Puisque l'algorithme CHARM produit des ensembles de feuilles fermés, on en déduit que l'on obtient bien $\mathcal{F}_1 = Ferme(\mathcal{P}_1)$ en sortie de la procédure DécouvertePFA_1 où ECLAT a été remplacé par CHARM. $\square$

On obtient donc un ensemble $\mathcal{F}_D^1 = Ferme(\mathcal{P}_1)$. Nous connaissons donc déjà les nœuds du graphe des accrochages de $\mathcal{F}_D^1$: ce sont les patterns de $\mathcal{F}_D^1$. Pour trouver les arcs correctement étiquetés, il faut pour tout couple de patterns accrochables $(P_1, P_2) \in \mathcal{F}_D^1 \times \mathcal{F}_D^1$ calculer $Ancre(P_1, P_2)$. Une implémentation de ce calcul est donnée dans la procédure *CalculeGraphe*, décrite dans l'algorithme 20.

Algorithme 20 CalculeGraphe

Entrée : $\mathcal{F}_D^1$ un ensemble fermé de pattern fréquents de profondeur 1
Sortie : $\mathcal{G}$ le graphe des accrochages de $\mathcal{F}_D^1$

$\mathcal{A} \leftarrow \emptyset$
Pour tout $P_1 \in \mathcal{F}_D^1$ **Faire**
  Pour tout $P_2 \in \mathcal{F}_D^1$ **Faire**
    Si $accrochables(P_1, P_2)$ **Alors**
      $AncreArc \leftarrow \emptyset$
      $\{na\} \leftarrow noeuds\_accrochage(P_1, P_2)$ // *1 seul nœud d'accrochage puisque patterns de profondeur 1*
      Pour tout $occ_1 \in P_1.loc$ **Faire**
        Pour tout $occ_2 \in P_2.loc$ **Faire**
          Si $\exists \mu \in occ_1.mappings$ tq $\mu(na) = occ_2.racine$ **Alors**
            $AncreArc \leftarrow AncreArc \cup \{(na, occ_1.racine, occ_2.racine)\}$
          Fin Si
        Fin Pour
      Fin Pour
      $\mathcal{A} \leftarrow \mathcal{A} \cup \{(P_2, P_1)$ d'étiquette $AncreArc\}$
    Fin Si
  Fin Pour
Fin Pour
Retourner $\mathcal{G} = (\mathcal{F}, \mathcal{A})$

Patterns racines

La première étape d'une itération de DRYADE est de trouver les patterns racine de $\mathcal{F}_D$.

La recherche des patterns racine est faite par la procédure *DonneRacines*, détaillée dans l'algorithme 21.

Propriété 11 *A partir d'un ensemble fermé de patterns fréquents $\mathcal{F}_D$ et de son graphe d'accrochages $(\mathcal{F}_D, \mathcal{A})$, la procédure DonneRacines calcule l'ensemble EPR des couples (PR, OR) tels que l'union des PR est l'ensemble des patterns racines de $\mathcal{F}_D$ et pour tout couple $(PR, OR) \in EPR$, OR est exactement l'ensemble des occurrences de racine de PR.*

Algorithme 21 *DonneRacines*

Entrée : Le graphe des accrochages $\mathcal{G} = (\mathcal{F}_D^i, \mathcal{A})$, pour une itération i donnée
Sortie : L'ensemble EPR de tous les couples (PR, OR) où PR est un pattern racine et OR sont ses occurrences de racine.

$EPR_{tmp} \leftarrow \emptyset$
Pour tout $P \in \mathcal{F}_D^i$ **Faire**
  $OR \leftarrow P.loc$
  Pour tout $P' \in succ(P)$ **Faire**
    Pour tout $(o', o) \in Ancre(P', P)$ **Faire**
      // $Ancre(P', P)$ est l'étiquette de l'arc (P', P) calculée par la procédure CalculeGraphe
      $OR \leftarrow OR - o$
    Fin Pour
  Fin Pour
  Si $\mid OR \mid \geq \varepsilon$ **Alors**
    $EPR_{tmp} \leftarrow EPR_{tmp} \cup \{(P, OR)\}$
  Fin Si
Fin Pour
// On recherche les classes d'équivalence racines du graphe quotient des accrochages
$EPR \leftarrow \emptyset$
Pour tout $(P, OR) \in EPR_{tmp}$ **Faire**
  Si $P.loc = OR$ **Alors**
    // Classe d'équivalence réduite à un singleton et racine du graphe quotient
    $EPR \leftarrow EPR \cup (P, OR)$
  Sinon
    // On recherche si la classe d'équivalence de P est un cycle racine du graphe quotient.
    Si $estDansCycleRacine(P, OR, \mathcal{G})$ **Alors**
      $EPR \leftarrow EPR \cup (P, OR)$
    Fin Si
  Fin Si
Fin Pour
Retourner EPR

Algorithme 22 *estDansCycleRacine*

Entrée : Un pattern P, ses ocurrences de racine OR, le graphe des accrochages $\mathcal{G} = (\mathcal{F}_D^i, \mathcal{A})$
Sortie : Renvoie VRAI si P est dans un cycle racine du graphe quotient associé à $\mathcal{G}$, et FAUX sinon.

// L'algorithme de cette fonction n'est pas donné ici. Il est basé sur un algorithme classique de détection de cycle, amélioré pour distinguer les cycles racines du graphes quotient associé à $\mathcal{G}$.

Preuve : L'algorithme itère sur tous les patterns P de $\mathcal{F}_D$, tous les patterns racines potentiels sont donc passés en revue.

Montrons d'abord que les occurrences de racine des patterns sont correctement calculées : On voit à la ligne 7 que pour calculer les occurrences de racine d'un pattern P, on retire de l'ensemble des occurrences de P toutes les occurrences qui ne satisfont pas la condition énoncée dans la définition 40. A la fin de la boucle l'ensemble OR trouvé est donc bien $OR(P)$. La condition de la ligne 10 assure que seuls les patterns vérifiant $\mid OR(P) \mid \geq \varepsilon$ peuvent être détectés comme patterns racines.

Montrons maintenant que les patterns résultats de *DonneRacines* sont bien les patterns des classes d'équivalence racine du graphe quotient des accrochages : On distingue deux cas de classes d'équivalence racine du graphe quotient des accrochages :

– Premier cas, une classe d'équivalence qui ne corresponde pas à un cycle. C'est donc une classe d'équivalence qui ne contient qu'un seul pattern P. Dans le graphe quotient des accrochages, cette classe n'a pas de père, cela signifie donc que le pattern P ne s'accroche sur aucun pattern, pour aucune de ses occurrences. On en déduit : $Locc(P, D) = OR(P)$. Ce cas est détecté par la ligne 17 de la procédure *DonneRacines*.

– Deuxième cas, une classe d'équivalence correspondant à un cycle, et qui donc contient plusieurs patterns $P_1, ..., P_n \quad (n > 1)$. Comme cette classe est racine pour le graphe quotient des accrochages, elle n'a pas de père. Cela signifie que les patterns de $P_1, ..., P_n$ ne s'accrochent sur aucun pattern autre qu'un pattern du cycle. Ceci est detecté correctement par la fonction *estDansCycleRacine*, dont nous n'avons pas donné le code ici pour ne pas alourdir inutilement ce chapitre.

□

Patterns à accrocher sur les patterns racines

Pour la suite, nous supposons que nous sommes à l'itération $i > 1$ de DRYADE.

Au début de l'itération i, nous disposons du résultat de l'itération précédente $\mathcal{F}_D^{i-1}$, et de EPR l'ensemble des patterns racines présents dans $\mathcal{F}_D^{i-1}$. Pour chaque pattern racine de EPR, l'algorithme doit découvrir les patterns de profondeur 1 à accrocher simultanément sur ce pattern, et les occurrences pour lesquelles cet accrochage multiple a lieu.

Pour cela, nous étendons d'abord notre notion d'accrochage ancré dans les données. Jusque là, nous avions restreint notre définition d'ancre à l'accrochage d'un pattern fréquent de profondeur 1 sur un autre pattern fréquent de profondeur quelconque. Nous donnons maintenant la définition de l'ancre pour l'accrochage simultané de plusieurs patterns de profondeur 1 sur un pattern racine de profondeur quelconque.

Définition 41 (Extension de la définition de l'ancre) *Soit $\mathcal{F}_D^i$ un ensemble fermé de patterns fréquents, et PR un pattern racine de $\mathcal{F}_D^i$. Soit OR l'ensemble des occurrences de racine de PR.*

Soit $EP_1 = \{(na_1, P_1), ..., (na_n, P_n)\} \subseteq N_{PR} \times \mathcal{F}_D^i$ un ensemble de couples comprenant chacun nœud d'accrochage sur PR et un pattern de profondeur 1 s'accrochant à PR sur ce nœud..

L'ancre de l'ensemble de couples EP_1 sur PR est définie de la façon suivante :

$Ancre(PR, EP_1) = \{(o_R, o_1, ..., o_n) \mid o_R \in OR, o_1 \in Locc(P_1, D), ..., o_n \in Locc(P_n, D)$ *tq* $(o_R, o_1) \in Ancre_{na_1}(PR, P_1) \wedge ... \wedge (o_R, o_n) \in Ancre_{na_n}(PR, P_n)\}$.

Nous pouvons ainsi définir la notion d'**ensemble d'accrochage multiple**, c'est-à-dire tous les patterns de profondeur 1 qui s'accrochent sur un pattern racine, avec leurs ancres et leurs occurrences d'accrochage.

Définition 42 (Ensemble et occurrence d'accrochage multiple) *Soit $\mathcal{F}_D^i$ un ensemble fermé de patterns fréquents, et PR un pattern racine de $\mathcal{F}_D^i$. Soit OR les occurrences de racine de PR.*

Soit $EP_1 = \{(na_1, P_1), ..., (na_n, P_n)\} \subseteq N_{PR} \times \mathcal{F}_D^i$ un ensemble de couples nœud d'accrochage sur PR/pattern fréquent de profondeur 1 s'accrochant sur ce nœud.

On note $EOAM(EP_1)$ l'ensemble des ***occurrences d'accrochage multiple*** *de EP_1 défini de la façon suivante :*

$$EOAM(EP_1) = \{o \in OR \mid \exists (o, o_1, ..., o_n) \in Ancre(PR, EP_1)\}$$

EP_1 est un ***ensemble d'accrochage multiple*** *ssi :*
- $\mid EOAM(EP_1) \mid \geq \varepsilon$
- *il n'existe aucun couple (na_j, P_j) et (na_k, P_k) dans EP_1 vérifiant $na_j = na_k$ et $P_j \sqsubset P_k$*

Définition 43 (Ensemble fermé des ensembles d'accrochage multiple) *Pour un pattern racine PR, l'ensemble $EAM(PR)$ est l'ensemble* ***fermé*** *de tous les ensembles d'accrochages multiples sur PR, c'est-à-dire que c'est le plus grand ensemble (au sens de l'inclusion ensembliste) d'ensemble d'accrochage multiple sur PR tel que : pour tout ensemble $EP_1 \in EAM(PR)$, il n'existe pas d'ensemble $EP_1' \in EAM(PR)$ vérifiant :*
- $EP_1' \subset EP_1$
- $EOAM(EP_1') = EOAM(EP_1)$

Considérons un pattern racine PR, et un ensemble d'accrochage multiple $EP_1 \in EAM(PR)$. Pour sélectionner les patterns de EP_1 correspondant au niveau de profondeur à construire à l'itération i, DRYADE recherche les patterns qui sont des patterns racines **au sein de** EP_1 (c'est-à-dire que l'on effectue une nouvelle étape de "stratification" des patterns, cette fois limitée aux patterns de EP_1).

Définition 44 (Ensemble d'accrochage multiple immédiat) *Soit PR un pattern racine, et $EP_1 \in EAM(PR)$ un ensemble d'accrochage multiple sur PR.*

L'ensemble $EP_{1R} = \{(na_1', P_1'), ..., (na_m', P_m')\} \subseteq EP_1$ est un ***ensemble d'accrochage multiple immédiat*** *si chaque pattern P de EP_{1R} est un pattern racine pour l'ensemble des patterns de EP_1.*

*L'**ancre d'accrochage immédiat** de EP_{1R} est l'ensemble $EAAI(EP_{1R}) \subseteq Ancre(PR, EP_{1R})$ tel que : $\forall (o_R, o_1, ..., o_m) \in EAAI(EP_{1R})$ et $\forall o_i \in \{o_1, ..., o_m\}$ o_i est une occurrence de racine pour P_i' lorsqu'il est accroché sur le nœud na_i'.*

L'ensemble des ***occurrences d'accrochage immédiat*** *de EP_{1R} est l'ensemble $EOAI(EP_{1R}) = \{o \in EOAM(EP_1) \mid \exists (o, o_1, ..., o_m) \in EAAI\}$, avec $|EOAI| \geq \varepsilon$.*

Notation : Dans les algorithmes, nous appellerons **contexte d'accrochage immédiat** un triplet $(EP_{1R}, EAAI(EP_{1R}), EOAI(EP_{1R}))$.

Pour chaque pattern racine PR, DRYADE recherche l'ensemble fermé de tous ses ensembles d'accrochage multiple immédiat.

Définition 45 (Ensemble fermé des ensembles d'accrochage multiple immédiat) *Soit PR un pattern racine de $\mathcal{F}_D$.*

L'ensemble ***fermé*** *de tous les ensembles d'accrochage multiple immédiat sur PR, noté $ECAI(PR)$ est le plus grand ensemble (au sens de l'inclusion ensembliste) d'ensembles d'accrochage multiple immédiat sur PR tel que : pour tout ensemble $EP_1 \in ECAI(PR)$, il n'existe pas d'ensemble $EP_1' \in ECAI(PR)$ vérifiant :*

- $EP_1 \subset EP'_1$
- $EOAI(EP'_1) = EOAI(EP_1)$

Exemple : Considérons le datatree D_Y. En observant son graphe des accrochages, figure V.17, on trouve comme pattern racine le pattern P_{11}, et comme candidats à l'accrochage les patterns P_{12}, P_{13} et P_{14}.

De fait, l'ensemble fermé des ensembles d'accrochage multiple sur P_{11} contient un seul élément : l'ensemble $EP_1 = \{(na_B^{11}, P_{12}), (na_C^{11}, P_{13}), (na_D^{11}, P_{14})\}$, pour les occurrences $EOAM(EP_1) = \{n_1, n_8\}$.

Au sein de EP_1, les patterns racines sont P_{12} et P_{13} (pour les trouver il suffit d'observer le graphe de la figure V.17 en supprimant P_{11}). On obtient donc un unique contexte d'accrochage immédiat :

$CAI = (\{(na_B^{11}, P_{12}), (na_C^{11}, P_{13})\}, \{(n_1, n_2, n_5), (n_8, n_{16}, n_{10})\}, \{n_1, n_8\})$

L'ensemble $ECAI(P_{11})$ est donc réduit au singleton $\{CAI\}$

Pour un pattern racine PR dont les occurrences de racine sont OR, la procédure $PatternsAAccrocher$, donnée dans l'algorithme 23, est chargée du calcul de $ECAI(PR)$.

Nous allons expliquer le fonctionnement de $PatternAAccrocher$ à l'aide d'un exemple. Pour cela, nous considérons le datatree D_Z représentée dans la figure V.18. Considérons un seuil de fréquence à $\varepsilon = 1$. L'ensemble $\mathcal{F}_1$ des patterns fréquents de profondeur 1 est représenté dans la figure V.19. Le graphe des accrochages pour ces patterns est donné dans la figure V.20. De manière évidente, le seul pattern racine est P_{11}, avec comme ensemble d'occurrences de racines $OR = \{n_1, n_8\}$.

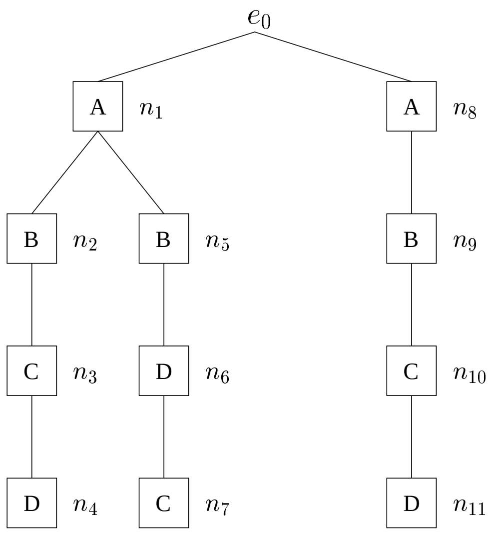


FIG. V.18 – D_Z

La procédure $PatternsAAccrocher$ commence par construire une matrice de transactions M pour les patterns accrochables sur P_{11}, voir tableau V.4.

Identifiant de transaction (occurrences)	(na_B^{11}, P_{12})	(na_C^{11}, P_{13})	(na_D^{11}, P_{14})
n_1	$\{n_2, n_5\}$	$\{n_3\}$	$\{n_6\}$
n_8	$\{n_9\}$	$\{n_{10}\}$	

TAB. V.4 – Matrice de transactions M des patterns à accrocher sur P_{11}

Cette matrice contient pour chaque occurrence $o \in OR$ et chaque pattern accrochable P_X sur un nœud d'accrochage na toutes les occurrences o' de P_X telles que $(o, o') \in Ancre_{na}(P_{11}, P_X)$. Les

Algorithme 23 *PatternsAAccrocher*

Entrée : Un pattern racine PR, ses occurrences de racine OR, $\mathcal{F}_D^i$ les patterns fréquents
Sortie : $ECAI(PR)$ tous les contextes d'accrochage immédiat sur PR.

$ECAI \leftarrow \emptyset$
$EAM \leftarrow \emptyset$
// Déterminer tous les ensembles d'accrochage multiple
Créer la matrice de transactions M avec transactions = occurrences de OR, items = patterns de profondeur 1 de $\mathcal{F}_D^i$ accrochables sur PR (pour un pattern, autant d'items que de nœuds d'accrochage) On a : $M[o \in OR, P_1 \text{ sur } na \in noeuds\_accrochage(PR, P_1)]$ = ensemble des nœuds de P_1 qui s'accrochent sur l'occurrence o de PR
$EAM \leftarrow CHARM(booleen(M), \varepsilon)$
$M' \leftarrow$ matrice vide
Pour tout $(EP_1, EOAM(EP_1)) \in EAM$ **Faire**
  Pour tout $(na, P) \in EP_1$ **Faire**
    // Test d'inclusion sur des patterns de profondeur 1, donc très facile à faire rapidement
    Si $\exists (na', P') \in EP_1$ tq $na = na'$ ET $P \sqsubset P'$ **Alors**
      $EP_1 \leftarrow EP_1 - \{(na, P)\}$
    Fin Si
  Fin Pour
  Pour tout $occPR \in EOAM(EP_1)$ **Faire**
    $TransactionsAncres \leftarrow ToutesAncres(occPR, M, 0)$ *// Voir algorithme 24*
    $TransactionsAncres \leftarrow RacineDansAncre(TransactionsAncres, occPR, PR, EP_1)$ *// Voir algorithme 25*
    Ajouter $TransactionAncres$ à M'
    $transacMapping(occPR) \leftarrow$ tous les identifiants de transaction de $TransactionsAncres$
  Fin Pour
Fin Pour
Supprimer de M' les transactions redondantes (mêmes patterns racines, mêmes ancres pour ces patterns racines)
$EP_{1R} \leftarrow CHARM(booleen(M'), \varepsilon)$
$ECAI \leftarrow ContextesImmediats(EP_{1R}, transacMapping)$ *// Voir algorithme 26*
// Garantir la fermeture sur les occurrences
Pour tout $CAI = (EP_1, EAAI, EOAI) \in ECAI$ **Faire**
  Si $\exists CAI' = (EP'_1, EAAI', EOAI') \in ECAI$ tq $EP_1 \subseteq EP'_1$ et $EOAI = EOAI'$ **Alors**
    $ECAI \leftarrow ECAI \setminus CAI$
  Fin Si
Fin Pour
Retourner $ECAI$

Algorithme 24 *ToutesAncres*

Entrée : M une matrice de transactions, $occPR$ l'identifiant d'une transaction de M, col un index de colonne dans M

Sortie : Toutes les transactions des ancres correspondant à $occPR$ dans M, à partir de la colonne col.

```
Resulat ← ∅
Si col ≤ nb colonnes de M Alors
  // Récupérer toutes les combinaisons que l'on peut construire à partir des items des colonnes suivantes
  tmpTrans ← ToutesAncres(occPR, M, col + 1)
  Si tmpTrans ≠ ∅ Alors
    Si M[occPR][col] ≠ ∅ Alors
      Pour tout occurrence o ∈ M[occPR][col] Faire
        Pour tout transac ∈ tmpTrans Faire
          // A chacune des combinaisons des colonnes suivantes ajouter l'occurrence courante de la
          colonne col
          Resultat ← Resultat ∪ {{o} ∪ transac}
        Fin Pour
      Fin Pour
    Sinon
      Resultat ← tmpTrans
    Fin Si
  Sinon
    Resultat ← M[occPR][col]
  Fin Si
Fin Si
Retourner Resultat
```

Algorithme 25 *RacineDansAncre*

Entrée : $TransactionsAncres$ un ensemble de transactions, $occPR$ l'occurrence de PR à laquelle correspondent ces transactions, PR un pattern racine, EP_1 un ensemble d'accrochage multiple sur PR

Sortie : Les transactions de $TransactionsAncres$ où on ne garde pour chaque transaction (donc ancre) que les patterns qui sont racines pour cette ancre.

```
Pour tout T ∈ TransactionsAncres Faire
  Pour tout itemP1 ∈ T Faire
    Si ∃itemP'1 ∈ T tq (itemP'1, itemP1) ∈ Ancre(P'1, P1) Alors
      T ← T − itemP1
    Fin Si
  Fin Pour
Fin Pour
Retourner TransactionsAncres
```

Algorithme 26 *ContextesImmediats*

Entrée : Un ensemble $EFIS$ d'itemsets fréquents du type $(EP_1, EnsSupport)$, où $EnsSupport$ est une liste d'ancres, $transacMapping$ fait correspondre à chaque occurrence de PR son ancre (on utilisera ici $transacMapping^{-1}$), M'

Sortie : Renvoie les contextes d'accrochage immédiat sur PR

```
EC ← ∅
Pour tout (EP1 = {(na1, P1), ..., (nan, Pn)}, EnsSupport) ∈ EFIS Faire
  Ancre ← ∅
  TID_occPR ← ∅
  Pour tout id ∈ EnsSupport Faire
    TID_occPR ← TID_occPR ∪ {transacMapping^-1(id)} // union SANS doublons
    Ancre ← Ancre ∪ {(transacMapping^-1(id), M'[id, (na1, P1)], ..., M'[id, (nan, Pn)])}
    // idem
  Fin Pour
  Si |TID_occPR| ≥ ε Alors
    EC ← EC ∪ {(EP1, Ancre, TID_occPR)}
  Fin Si
Fin Pour
Retourner EC
```

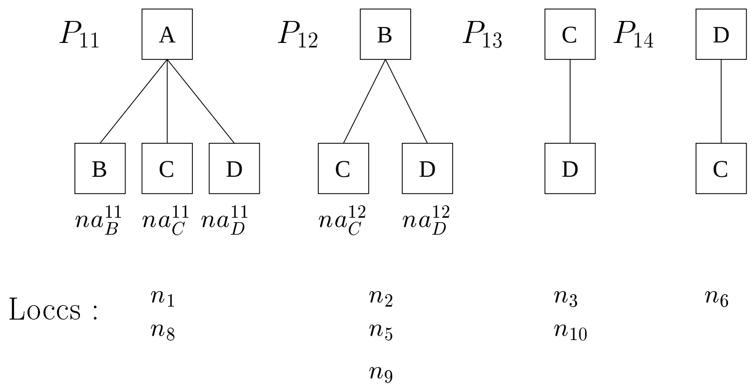


Fig. V.19 – Patterns fréquents fermés de profondeur 1 de D_Z

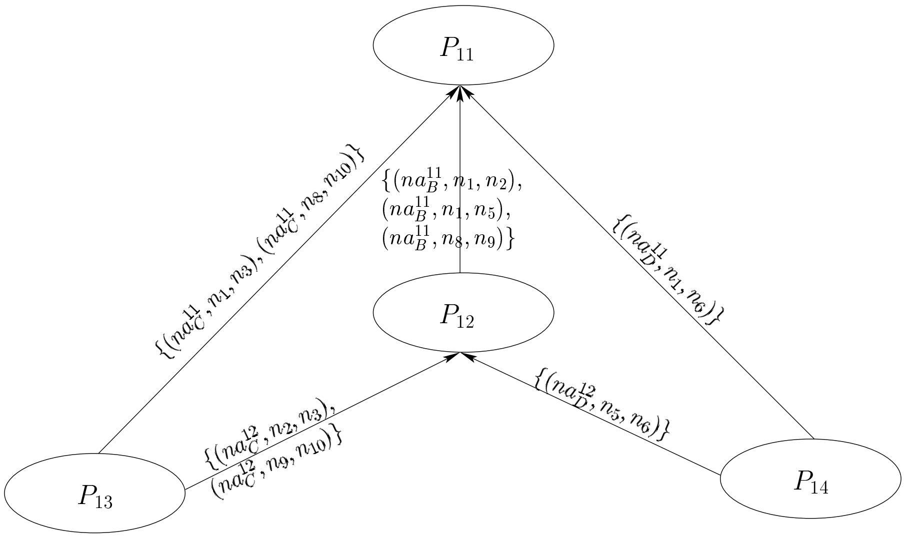


FIG. V.20 – Graphe des accrochages des patterns fréquents fermés de profondeur 1 de D_Z

attributs ne sont pas binaires : on applique donc CHARM sur $booleen(M)$, qui est une transformation de M en attributs binaires. La règle de transformation est simple : une case vide vaut 0, une case non vide vaut 1.

Nous obtenons ainsi des Frequent Itemsets fermés qui sont des ensembles d'accrochage multiple sur P_{11}, ainsi que les ensembles-support qui sont les occurrences du pattern racine où les patterns s'accrochent. Ici, ces Frequent Itemsets sont :

- $EP_1^1 = \{(na_B^{11}, P_{12}), (na_C^{11}, P_{13}), (na_D^{11}, P_{14})\}$, $EnsSupport^1 = \{n_1\}$
- $EP_1^2 = \{(na_B^{11}, P_{12}), (na_C^{11}, P_{13})\}$, $EnsSupport^2 = \{n_1, n_8\}$

L'étape suivante de l'algorithme consiste à déterminer parmi les ensembles de patterns fréquents découverts les ensembles de patterns immédiatement accrochables, et donc les contextes d'accrochage immédiats.

Pour cela, nous devons nous placer aux niveau des ancres des ensembles de patterns fréquents.

Considérons le premier Frequent Itemset, $EP_1^1 = \{(na_B^{11}, P_{12}), (na_C^{11}, P_{13}), (na_D^{11}, P_{14})\}$, $EnsSupport^1 = \{n_1\}$. En observant M, on voit que P_{12} accroché sur na_B^{11} a deux occurrences possibles. Il y a donc deux ancres possibles, c'est le rôle de la procédure récursive $ToutesAncres$ que de les rendre explicites. À partir d'une transaction de M, dont chaque item peut être un ensemble d'occurrences, cette procédure construit toutes les transactions possibles avec une seule occurrence par item. Cela revient à construire toutes les combinaisons d'occurrences en en choisissant une dans chaque ensemble d'occurrences correspondant à un item. Les nouvelles transactions obtenues sont données dans le tableau V.5.

Identifiant de transaction (ancres)	(na_B^{11}, P_{12})	(na_C^{11}, P_{13})	(na_D^{11}, P_{14})
n_1^1	n_2	n_3	n_6
n_1^2	n_5	n_3	n_6

TAB. V.5 – $TransactionsAncres$

Ensuite, la procédure $RacineDansAncre$ calcule pour chaque transaction, et donc pour chaque ancre, les patterns qui sont racines pour cette ancre. Les occurrences de ces patterns ne s'accrochent pas sur les occurrences des autres patterns de l'ancre. Les transactions ainsi modifiée sont données dans le tableau V.6.

Identifiant de transaction (ancres)	(na_B^{11}, P_{12})	(na_C^{11}, P_{13})	(na_D^{11}, P_{14})
n_1^1	n_2		n_6
n_1^2	n_5	n_3	

TAB. V.6 – *TransactionsAncres* après *RacinesDansAncre*

Ces transactions sont ajoutées à la matrice M'. Remarquons que de passer du niveau de granularité “occurrences” au niveau de granularité “ancre” était indispensable : ce que nous regroupions dans M à l'occurrence n_1 a dans M' été séparé en deux transactions différentes.

Les mêmes opérations sur le Frequent Itemset $EP_1^2 = \{(na_B^{11}, P_{12}), (na_C^{11}, P_{13})\}$, $EnsSupport^2 = \{n_1, n_8\}$ donnent la matrice M' du tableau V.7.

Identifiant de transaction (ancres)	(na_B^{11}, P_{12})	(na_C^{11}, P_{13})	(na_D^{11}, P_{14})
n_1^1	n_2		n_6
n_1^2	n_5	n_3	
n_1^3	n_2		
n_1^4	n_5	n_3	
n_8	n_9		

TAB. V.7 – M' avant suppression des doublons

On en supprime les transactions redondantes, ici n_1^4 (redondante avec n_1^2), pour obtenir la matrice finale du tableau V.8.

Identifiant de transaction (ancres)	(na_B^{11}, P_{12})	(na_C^{11}, P_{13})	(na_D^{11}, P_{14})
n_1^1	n_2		n_6
n_1^2	n_5	n_3	
n_1^3	n_2		
n_8	n_9		

TAB. V.8 – M' finale

L'exécution de CHARM sur M' nous donne :
- $EP_1^{1'} = \{(na_B^{11}, P_{12}), (na_D^{11}, P_{14})\}$, $EnsSupport^1 = \{n_1^1\}$
- $EP_1^{2'} = \{(na_B^{11}, P_{12}), (na_C^{11}, P_{13})\}$, $EnsSupport^2 = \{n_1^2\}$
- $EP_1^{3'} = \{(na_B^{11}, P_{12})\}$, $EnsSupport^3 = \{n_1^1, n_1^2, n_1^3, n_8\}$

Le résultat de *PatternsAAccrocher* est donc l'ensemble de contextes d'accrochage immédiat :
- $CAI_1 = (\{(na_B^{11}, P_{12}), (na_D^{11}, P_{14})\}, \{(n_1, n_2, n_6)\}, \{n_1\})$
- $CAI_2 = (\{(na_B^{11}, P_{12}), (na_C^{11}, P_{13})\}, \{(n_1, n_5, n_3)\}, \{n_1\})$
- $CAI_3 = (\{(na_B^{11}, P_{12})\}, \{(n_1, n_2), (n_1, n_5), (n_8, n_9)\}, \{n_1, n_8\})$

(Nous pouvons remarquer que dans CAI_3, l'ancre (n_1, n_2) a été produite deux fois, mais l'union sans doublons de la procédure *ContextesImmediats* n'en a gardé qu'une, comme il le fallait.)

Propriété 12 *Pour un pattern racine PR, l'algorithme PatternAAccrocher calcule bien $ECAI(PR)$.*

Preuve : La matrice de transactions M construite à la ligne 4 a comme colonnes les patterns de profondeur 1 de $\mathcal{F}_D^i$ qui s'accrochent sur PR ainsi que leur nœud d'accrochage, et comme lignes les occurrences de PR. Donc la j-ème case de la i-ème ligne indique les occurrences du pattern j qui s'accrochent sur l'occurrence i de PR pour un certain nœud d'accrochage.

L'algorithme CHARM de la ligne 5 opère sur une version booléenne de M, pour chaque occurrence de PR (ligne de M), on connait les patterns de profondeur 1 qui s'accrochent sur cette occurrence, ainsi que leur nœud d'accrochage. Donc ici CHARM calcule exactement l'ensemble fermé $EAM(PR)$ de la définition 43 (moyennant la suppression des patterns de EP_1 inclus les uns dans les autres effectuée plus tard, lignes 9 à 13).

Ensuite, pour chaque couple $EP_1 \in EAM(PR)$, associé à ses occurrences d'accrochage multiples $EOAM(EP_1)$, les lignes 8 à 12 de l'algorithme remplissent les lignes correspondantes de la matrice M'.

Les colonnes de M' sont toujours des patterns de profondeur 1 qui s'accrochent sur PR et leur nœud d'accrochage, mais les lignes de M' sont tous les éléments de $Ancre(RP, EP_1)$, pour tous les EP_1 de $EAM(PR)$.

Pour chaque $EP_1 = \{(na_1, P_1), ..., (na_n, P_n)\} \in EAM(PR)$, la fonction récursive $ToutesAncres$ remplit pour chaque occurrence o_R de $EOAM(EP_1)$ les lignes de M' correspondants aux éléments de $Ancre(PR, EP_1)$ qui ont o_R comme occurrence de PR. Ceci est effectué en étendant la ligne correspondant à l'occurrence o_R de M, qui contient pour chaque pattern de EP_1 toutes ses occurrences s'accrochant sur o_R, sur le nœud d'accrochage spécifié. Un élément de $Ancre(PR, EP_1)$ ayant o_R comme racine de PR correspond donc au choix d'une occurrence pour chacun des $(na_j, P_j), j \in [1, n]$ (ce choix est réalisé pour chaque (na_j, P_j) par la boucle de la ligne 7 de $ToutesAncre$). Il y a autant d'éléments de $Ancre(PR, EP_1)$ ayant o_R comme occurrence de PR qu'il y a de choix possibles pour les occurrences des $(na_j, P_j), j \in [1, n]$.

Pour chaque ligne de M' correspondant à un élément $(o_R, o_1, ..., o_n)$ de $Ancre(PR, EP_1)$, la procédure $RacineDansAncre$ supprime les occurrences o_j qui s'accrochent à d'autres occurrences $o_k \in \{o_1, ..., o_n\}$ (lignes 3 et 4), garantissant que les occurrences qui restent dans la ligne de M' sont bien des occurrences de racine.

Lors de la construction de M', deux ensembles d'accrochages multiples différents EP_1, EP_1' avec $EP_1 \subset EP_1$ peuvent produire les mêmes lignes après $RacineDansAncre$. La ligne 21 élimine tous les doublons de M'.

L'application de CHARM sur M' à la ligne 23 produit donc des ensemble fermés de patterns de profondeur 1 s'accrochant sur PR. Les occurrences de ces ensembles de ces patterns sont toutes des occurrences de racines, donc les patterns sont des patterns racines. Chaque ensemble de pattern est donc un ensemble d'accrochage multiple immédiat.

Pour faciliter la tâche des procédures ultérieures, on crée des structures appellées contextes d'accrochage immédiats. La création des contextes d'accrochage immédiat est réalisée par la fonction $ContextesImmediats$, qui élimine les ensembles d'accrochage multiple immédiat dont l'ensemble d'occurrences d'accrochage immédiat $EOAI$ est de taille inférieure à ε (lignes 9-10), et sinon se contente de mettre le résultat sous la forme attendue.

L'ensemble $ECAI$ obtenu vérifie la propriété de fermeture, mais par rapport aux ancres d'accrochage immédiat. La définition 45 de $ECAI(PR)$ réclame la fermeture par rapport aux occurrences d'accrochage immédiat. Les lignes 25 à 29 garantissent cette fermeture par rapport aux occurrences d'accrochage immédiat.

D'où l'ensemble $ECAI$ obtenu est bien $ECAI(PR)$, vérifiant la définition 45. □

Accrochage et suppression des feuilles redondantes

Les contextes d'accrochage immédiat précédemment découverts permettent de savoir quels patterns de profondeur 1 accrocher sur un pattern racine PR de $\mathcal{F}_D^i$, et sur quels nœuds d'accrochage. L'étape qui suit consiste donc à effectuer les accrochages correctement afin d'obtenir les patterns de $\mathcal{F}_D^{i+1}$.

L'accrochage en lui-même ne pose aucun problème, mais par contre il faut éliminer les feuilles redondantes provenant du pattern racine dans le pattern résultant de l'accrochage. Plusieurs combinaisons d'éliminations sont possibles, et produiront donc plusieurs patterns dans $\mathcal{F}_D^{i+1}$.

Nous devons être à même de détecter pour un pattern racine PR et un contexte d'accrochage immédiat CAI de ce pattern les feuilles redondantes.

Définition 46 (Feuilles redondantes pour une ancre) *Soit PR un pattern racine, soit $CAI = (EP_1, EAAI, EOAI)$ un contexte d'accrochage immédiat sur PR, avec $EP_1 = \{(na_1, P_1), ..., (na_n, P_n)\}$. Soient $f_{n+1}, ..., f_k$ $k > n$ les feuilles du pattern PR où ne s'accrochent pas un pattern de EP_1. Pour une ancre $(o_R, o_1, ..., o_N) \in EAAI$, et pour un mapping $\mu \in \mathcal{EM}_{o_R}(PR, D)$ tel que $\forall i \in [1, n]$ $\mu(na_i) = o_i$, une feuille $f_{red} \in \{f_{n+1}, ..., f_k\}$ est redondante s'il existe un pattern $P_j \in \{P_1, ..., P_n\}$, un mapping $\mu' \in \mathcal{EM}_{o_j}(P_j)$ et une feuille f' de P_j tels que $\mu(f_{red}) = \mu'(f')$.*

Nous introduisons la notion de **contexte de création de pattern**, qui permet de savoir exactement quels patterns créer. La découverte de ces contextes de création de patterns à partir des contextes d'accrochage immédiat précède la création des patterns par accrochage et suppression de feuilles redondantes.

Définition 47 (Contexte de création de pattern) *Soit PR un pattern racine, soit $CAI = (EP_1, EAAI, EOAI)$ un contexte d'accrochage immédiat sur PR, avec $EP_1 = \{(P_1, na_1), ..., (P_n, na_n)\}$. Soient $f_{n+1}, ..., f_k$ $k > n$ les feuilles du pattern PR où ne s'accrochent pas un pattern de EP_1.*

Un contexte de création de pattern $CCP = (EP_1, FG, EACP, EOCP)$ est constitué de :

- *l'ensemble EP_1 des patterns fréquents de profondeur 1 s'accrochant immédiatement sur PR, avec leurs nœuds d'accrochage ;*
- *l'ensemble $FG = \{f_{\neg red.1}, ..., f_{\neg red.m}\} \subseteq \{f_{n+1}, ..., f_k\}$ des feuilles de PR à ne pas supprimer lors de l'accrochage ;*
- *l'ensemble $EACP$ des ancres supportant l'accrochage immédiat des EP_1 sur PR où les feuilles de FG ne sont pas redondantes. Une ancre $ACP \in EACP$ est de la forme $ACP = (o_R, o_1, ..., o_n, o_{\neg red.1}, ..., o_{\neg red.m})$, où $(o_R, o_1, ..., o_n) \in EAAI$ et pour tout mapping $\mu \in \mathcal{EM}_{o_R}(PR, D)$ tel que $\forall i \in [1, n]$ $\mu(na_i) = o_i$ et $\forall i \in [1, m]$ $\mu(f_{\neg red.i}) = o_{\neg red.i}$, les feuilles $f_{\neg red.i}, i \in [1, m]$ ne sont pas redontantes ;*
- *l'ensemble $EOCP$ des occurrences de PR apparaissant dans $EACP$, avec $|EOCP| \geq \varepsilon$.*

Pour un pattern racine PR et un contexte d'accrochage immédiat CAI, nous voulons découvrir l'ensemble **fermé** $ECCP(PR, CAI)$ de tous les contextes de création de pattern pour PR et CAI.

Définition 48 *Soit PR un pattern racine et $CAI \in ECAI(PR)$ un contexte d'accrochage immédiat sur PR. $ECCP(PR, CAI)$ est l'ensemble fermé des contextes de création de pattern sur PR, CAI. C'est donc le plus grand ensembre tel qu'il n'existe pas deux contextes de création de pattern $CCP = (EP_1, FG, EACP, EOCP)$ et $CCP' = (EP_1, FG', EACP', EOCP')$ dans $ECCP(PR, CAI)$ tels que :*

- *$FG \subset FG'$*

– $EACP = EACP'_{|FG}$ *(on ne considère que les parties communes des ancres, c'est-à-dire les occurrences de* EP_1 *et de* FG*)*

Ainsi pour chaque CAI nous obtenons un ensemble de contextes de création de pattern $ECCP(PR, CAI)$. L'ensemble $\cup_{CAI \in ECAI} ECCP(PR, CAI)$ est l'ensemble qui va être utilisé pour effectuer les accrochages. Or, rien ne garantit que cet ensemble soit fermé par rapport aux patterns à accrocher et aux occurrences de PR.

Définition 49 *Soit* PR *un pattern racine, et* $ECAI(PR)$ *l'ensemble des contextes d'accrochage immédiat sur* PR*. On définit* $ECCP = Ferme(\cup_{CAI \in ECAI} ECCP(PR, CAI))$*, c'est-à-dire qu'il n'existe pas deux contextes de création de pattern* $CCP = (EP_1, FG, EACP, EOCP)$ *et* $CCP' = (EP'_1, FG', EACP', EOCP')$ *tels que :*
– $EP_1 \subset EP'_1$
– $EOCP = EOCP'$

Nous pouvons enfin créer les patterns fréquents par accrochage de patterns de profondeur 1 et suppression des feuilles redondantes, à l'aide des contextes de création de patterns de $ECCP$.

La procédure *AccrochageSuppr*, décrite dans l'algorithme 27, calcule l'ensemble $ECCP$ puis construit les patterns associés.

Exemple : Pour illustrer le fonctionnement de la procédure *AccrochageSuppr*, nous allons effectuer l'accrochage des contextes d'accrochage immédiat CAI_1, CAI_2, CAI_3 définis précédemment sur le pattern racine P_{11}. Nous rappelons le contenu de ces contextes d'accrochage immédiat :

$$CAI_1 = (\{(na_B^{11}, P_{12}), (na_D^{11}, P_{14})\}, \{(n_1, n_2, n_6)\}, \{n_1\})$$

$$CAI_2 = (\{(na_B^{11}, P_{12}), (na_C^{11}, P_{13})\}, \{(n_1, n_5, n_3)\}, \{n_1\})$$

$$CAI_3 = (\{(na_B^{11}, P_{12})\}, \{(n_1, n_2), (n_1, n_5), (n_8, n_9)\}, \{n_1, n_8\})$$

Nous nommerons f_B, f_C et f_D les feuilles de P_{11}.

Pour calculer $ECCP(P_{11}, CAI_1)$, la seule feuille qui puisse ne pas être redondante est f_C, car P_{12} s'accroche sur $f_B = na_B^{11}$ et P_{14} s'accroche sur $f_D = na_D^{11}$. La matrice M correspondante construite par $CreerCCP$ est donc :

Identifiant de transaction $= (P_{11}, (na_B^{11}, P_{12}), (na_D^{11}, P_{14}), f_C)$	f_C
(n_1, n_2, n_6, n_3)	0
(n_1, n_2, n_6, n_7)	0

La feuille f_C est redondante, il faut donc la supprimer. D'où $ECCP(P_{11}, CAI_1) = \{CCP_1\}$, avec $CCP_1 = (\{(na_B^{11}, P_{12}), (na_D^{11}, P_{14})\}, \emptyset, \{(n_1, n_2, n_6)\}, \{n_1\})$.

La matrice M pour l'accrochage de CAI_2 sur P_{11} construite par $CreerCCP$ est :

Identifiant de transaction $= (P_{11}, (na_B^{11}, P_{12}), (na_C^{11}, P_{13}), f_D)$	f_D
(n_1, n_5, n_3, n_4)	0
(n_1, n_5, n_3, n_6)	0

Ce cas est similaire au précédent, donc $ECCP(P_{11}, CAI_2) = \{CCP_2\}$, avec $CCP_2 = (\{(na_B^{11}, P_{12}), (na_C^{11}, P_{13})\}, \emptyset, \{(n_1, n_5, n_3)\}, \{n_1\})$. Pour l'accrochage du contexte CAI_3, la matrice M construite par $CreerCCP$ est la suivante :

Algorithme 27 *AccrochageSuppr*

Entrée : Un pattern racine PR, ses contextes d'accrochage immédiat $ECAI$

Sortie : EP l'ensemble fermé des patterns fermés fréquents résultant de l'accrochage des patterns de contextes de $ECAI$ sur PR, avec suppression des feuilles redondantes, $NewPattInfos$ qui donne pour chaque nouveau pattern les patterns dont il est issu.

$EP \leftarrow \emptyset$
$ECCP \leftarrow \emptyset$
$NewPattInfos \leftarrow \emptyset$
Pour tout $CAI = (EP_1, EAAI, EOAI) \in ECAI$ **Faire**
  $ECCP \leftarrow ECCP \cup CreerCCP(PR, CAI)$ *// Voir algorithme 28*
Fin Pour
// Garantir la fermeture sur les occurrences
Pour tout $CCP = (EP_1, FG, EACP, EOCP) \in ECCP$ **Faire**
  Si $\exists CCP' = (EP'_1, FG', EACP', EOCP') \in ECCP$ tq $EP_1 \subseteq EP'_1$ et $EOCP = EOCP'$ **Alors**
    $ECCP \leftarrow ECCP \setminus CCP$
  Fin Si
Fin Pour
Pour tout $CCP = (EP_1, FG, EACP, EOCP) \in ECCP$ **Faire**
  $P_{new} \leftarrow$ nouvel arbre basé sur $PR \oplus EP_1$
  Éliminer de P_{new} les feuilles provenant de PR qui ne sont pas dans FG
  $P_{new}.loc \leftarrow \emptyset$
  Pour tout $o \in EOCP$ **Faire**
    $Occ.racine \leftarrow o$
    $Occ.mappings \leftarrow \{$ mappings de PR correspondant aux ancres de $EACP\} \circ \{$ mappings de EP_1 correspondants aux ancres de $EACP\}$
    $P_{new}.loc \leftarrow P_{new}.loc \cup Occ$
  Fin Pour
  $EACP' \leftarrow EACP$ privé des références aux feuilles non redondantes (on revient au CAI)
  $NewPattInfos \leftarrow NewPattInfos \cup \{(P_{new}, PR, (EP_1, EACP', EOCP))\}$
  $EP \leftarrow EP \cup P_{new}$
Fin Pour
Retourner $(EP, NewPattInfo)$

Algorithme 28 *CreerCCP*

Entrée : Un pattern racine PR, un contexte d'accrochage immédiat $CAI = (EP_1, EAAI, EOAI)$ (avec $EP_1 = \{P_1, ..., P_n\}$). Feuilles de $PR = \{f_1, ..., f_n, f_{n+1}, ..., f_k\}$

Sortie : $ECCP$ l'ensemble des contextes de création de pattern pour PR et CAI.

$ECCP \leftarrow \emptyset$
// Création de la matrice pour découvrir les ensembles de feuilles non redondantes
$M \leftarrow$ matrice de transactions vide, colonnes = feuilles $f_{n+1}, ..., f_k$
Pour tout $A = (o_R, o_1, ..., o_n) \in EOAI$ **Faire**
    Pour tout $Occ \in PR.loc$ tq $Occ.racine \in EOAI$ **Faire**
        Pour tout $\mu \in Occ.mappings$ tq $\mu(f_1) = o_1$ et ... et $\mu(f_n) = o_n$ **Faire**
            Créer une transaction T, $T.tid = (o_R, o_1, ..., o_n, \mu(f_{n+1}), ..., \mu(f_k))$
            Pour tout Feuille $f \in \{f_{n+1}, ..., f_k\}$ **Faire**
                Si $\not\exists P_i \in EP_1, Occ' \in P_i.loc, \mu' \in Occ'.mappings, f' \in feuilles(P_i)$ avec $\mu'(racine(P_i)) = o_i$ tq $\mu(f) = \mu'(f')$ **Alors**
                    $T[f] \leftarrow 1$ *// pour cette occurrence f existe en tant que feuille exclusive de o_R, il ne faut donc pas la supprimer*
                Sinon
                    $T[f] \leftarrow 0$
                Fin Si
            Fin Pour
            Ajouter T à M
        Fin Pour
    Fin Pour
Fin Pour
$EFIS \leftarrow CHARM(M, \varepsilon)$
Éliminer de $EFIS$ tous les itemsets dont les ancres de l'ensemble-support correspondent à moins de ε occurrences de $EOAI$
// Création des contextes de création de patterns
Pour tout $(fis, EnsSupport) \in EFIS$ **Faire**
    $EOCP \leftarrow$ ensembles des occurrences de PR apparaissant dans les ancres de l'ensemble-support
    $ECCP \leftarrow ECCP \cup \{(EP_1, fis, EnsSupport, EOCP)\}$
Fin Pour
// Gestion du cas où toutes les feuilles sont supprimées
Si il n'existe par d'$EOCP$ tel que $EOCP = EOAI$ **Alors**
    $ECCP \leftarrow ECCP \cup \{(EP_1, \emptyset, EAAI, EOAI)\}$
Fin Si
Retourner $ECCP$

Identifiant de transaction $= (P_{11}, (na_B^{11}, P_{12}), f_C, f_D)$	f_C	f_D
(n_1, n_2, n_3, n_4)	0	0
(n_1, n_2, n_3, n_6)	0	1
(n_1, n_2, n_7, n_4)	1	0
(n_1, n_2, n_7, n_6)	1	1
(n_1, n_5, n_7, n_6)	0	0
(n_1, n_5, n_7, n_4)	0	1
(n_1, n_5, n_3, n_6)	1	0
(n_1, n_5, n_3, n_4)	1	1
$(n_8, n_9, n_{10}, n_{11})$	0	0

Le cas où toutes les feuilles sont supprimées est encore valide grâce à l'occurrence n_8. Mais cette fois il y a d'autres ensembles fermés fréquents, on trouve dans $ECCP(P_{11}, CAI_3)$:

- $CCP_3 = (\{(na_B^{11}, P_{12})\}, \{f_C\}, \{(n_1, n_2, n_7, n_4), (n_1, n_2, n_7, n_6), (n_1, n_5, n_3, n_6), (n_1, n_5, n_3, n_4)\}, \{n_1\})$
- $CCP_4 = (\{(na_B^{11}, P_{12})\}, \{f_D\}, \{(n_1, n_2, n_3, n_6), (n_1, n_2, n_7, n_6), (n_1, n_5, n_7, n_4), (n_1, n_5, n_3, n_4)\}, \{n_1\})$
- $CCP_5 = (\{(na_B^{11}, P_{12})\}, \{f_C, f_D\}, \{(n_1, n_2, n_7, n_6), (n_1, n_5, n_3, n_4)\}, \{n_1\})$
- $CCP_6 = (\{(na_B^{11}, P_{12})\}, \emptyset, \{(n_1, n_2), (n_1, n_5), (n_8, n_9)\}, \{n_1, n_8\})$

Toutefois, lorsque l'on va construire $ECCP = ECCP(P_{11}, CAI_1) \cup ECCP(P_{11}, CAI_2) \cup ECCP(P_{11}, CAI_3)$, les contextes de création de patterns CCP_3, CCP_4 et CCP_5 sont éliminés par la propriété de fermeture. On a donc $ECCP = \{CCP_1, CCP_2, CCP_6\}$.

La figure V.21 montre tous les patterns créés à partir de $ECCP$.

Définition 50 (Pattern fréquent par accrochage multiple immédiat) *Soit PR un pattern racine, et soit $EP_1 = \{(na_1, P_1), ..., (na_n, P_n)\}$ un ensemble d'accrochage multiple immédiat sur PR. L'ensemble $EPatt(PR, EP_1)$ des patterns fréquents résultant de l'accrochage multiple immédiat de EP_1 sur PR est l'ensemble fermé des patterns P vérifiant :*

- $P \sqsubseteq ((PR \oplus P_1) \oplus ...) \oplus P_n$
- $| Locc(P, D) | \geq \varepsilon$
- *P ne contient aucune feuille redondante.*
- *il n'existe pas de pattern $P' \in EPatt(PR, EP_1)$ tel que :*
- $P' \sqsubset P$
- $Locc(P', D) = Locc(P', projete(P, D)) (= Locc(P, D)$ *ici*$)$

Définition 51 (Ensemble fermé des patterns dérivés d'un pattern racine) *Soit PR un pattern racine. $EPatt(PR)$ est l'**ensemble fermé des patterns dérivés** de PR :*

$EPatt(PR)$ est le plus gros ensemble tel que $EPatt(PR) \subseteq \bigcup_{EP_1 \in ECAI(PR)} EPatt(PR, EP_1)$ et pour tout pattern $P \in EPatt(PR)$ il n'existe pas $P' \in EPatt(PR)$ tel que :

- $P' \sqsubset P$
- $Locc(P', D) = Locc(P', projete(P, D)) (= Locc(P, D)$ *ici*$)$

Propriété 13 *À partir d'un pattern racine PR et de l'ensemble $ECAI(PR)$ de ses contextes d'accrochage immédiat, la procédure AccrochageSuppr calcule $EPatt(PR)$.*

Preuve : Pour chaque $CAI = (EP_1, EAAI, EOAI) \in ECAI(PR)$, *AccrochageSuppr* appelle la procédure $CreerCCP$. Le rôle de cette procédure est de créer $ECCP(PR, CAI)$, montrons qu'elle le fait effectivement.

Des lignes 3 à 19, la procédure $CreerCCP$ crée une matrice M. Cette matrice va être utilisée pour déterminer les combinaisons de feuilles non redondantes (feuilles à ne pas supprimer). Chaque

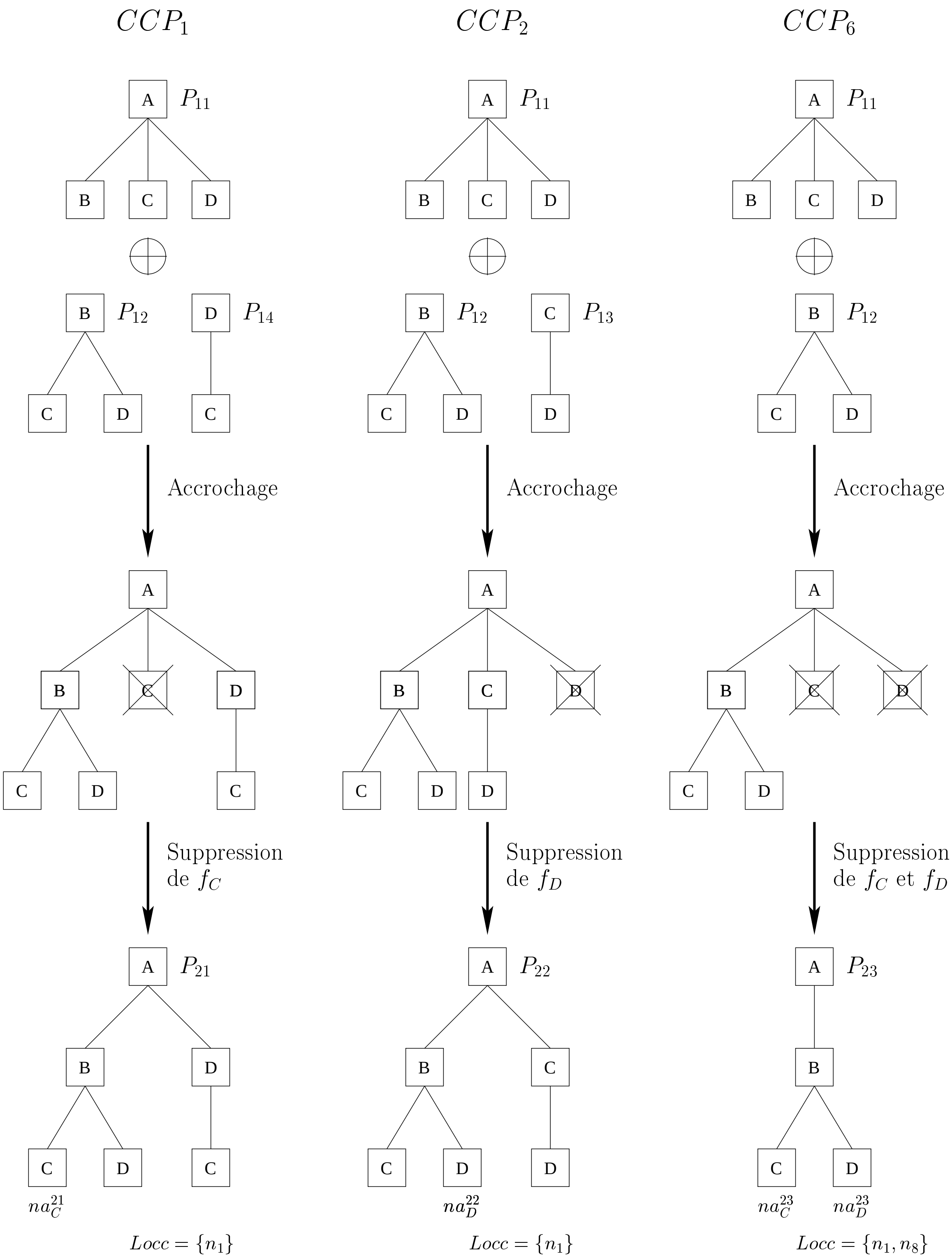


FIG. V.21 – Patterns créés à partir des contextes de création de patterns de $ECCP$

transaction de la matrice est construite à partir d'un mapping qui respecte un élément de $EAAI$ (test de la ligne 6), dont on considère les occurrences pour les feuilles de PR auxquelles ne s'accrochent pas des patterns de EP_1 (feuilles $\{f_{n+1}, ..., f_k\}$). Pour chacune de ces occurrences, le test de la ligne 9 vérifie qu'elle n'est pas redondante. Ce test correspond exactement à la définition 46.

Nous en déduisons que l'ensemble fermé $EFIS$ produit à la ligne 19 correspond aux ancres et aux feuilles non redondantes de $ECCP(PR, CAI)$, sans tenir compte de la contrainte sur le nombre d'occurrences de $EOAI$ contenues dans les ancres. La ligne 20 assure le respect de cette contrainte, et les lignes 22 à 25 se chargent de la mise en forme. Pour obtenir exactement l'ensemble $ECCP(PR, CAI)$, il manque la prise en compte du cas où $FG = \emptyset$, jamais effectuée par l'algorithme CHARM.

En effet, pour faire un ensemble fermé comme $ECCP(PR, CAI)$, CHARM ne prend pas en compte le cas où $FG = \emptyset$. Mais ici ce cas nous intéresse, car c'est le cas où toutes les feuilles sont supprimées. Nous le prenons donc aussi en compte. L'ensemble vide est inclus dans tous les autres ensembles, mais le support du pattern où toutes les feuilles sont supprimées est égal à l'ensemble $EAAI$: toutes les ancres supportent au moins ce cas, car l'ensemble vide est minimal par l'inclusion de patterns. On en déduit que le cas $FG = \emptyset$ ne sera pas éliminé s'il existe au moins une ancre $A = (o_R, o_1, ..., o_n) \in EAAI$ qui n'apparaît dans aucun $EACP$ de l'ensemble des contextes de création de pattern. Pour cela, on peut montrer qu'il faut que l'occurrence o_R de A n'apparaîsse dans aucun des $EACP$ de l'ensemble des contextes de création de pattern. Il suffit de remarquer que dans le cas contraire, s'il existe un contexte de création de pattern dont le $EACP$ contient une ancre du type $(o_R, o'_1, ..., o'_n)$ qui a des feuilles non redondantes, alors soit ses feuilles non redondantes ne sont pas des descendants des $o_1, ..., o_n$, et dans ce cas le pattern d'ancre A pourra aussi les utiliser comme feuilles non redondantes, soit ses feuilles non redondantes sont des descendants des nœuds $o_1, ..., o_n$, et donc symétriquement A pourra utiliser les descendants de $o'_1, ..., o'_n$ comme feuilles non redondantes.

Nous en déduisons que si une occurrence de $EOAI$ n'apparaît dans aucun des $EACP$ connus, alors le contexte de création de pattern $(EP_1, \emptyset, EAAI, EOAI)$ se trouve dans $ECCP(PR, CAI)$.

Les lignes 27 à 29 l'ajoutent donc ce cas si au moins une occurrence de $EOAI$ n'apparaît jamais dans les $EOCP$.

L'ensemble $ECCP$ retourné correspond donc exactement à l'ensemble $ECCP(PR, CAI)$ de la définition 48.

La procédure *AccrochageSuppr* fait l'union des $ECCP(PR, CAI)$ pour tout $CAI \in ECAI$, puis des lignes 7 à 12 garantit que l'ensemble $ECCP$ obtenu est bien fermé par rapport aux patterns à accrocher et aux occurrences. Donc l'ensemble $ECCP$ obtenu après la ligne 12 est bien l'ensemble de la définition 49.

Ensuite, pour chaque contexte de création de pattern $CCP = \{EP_1, FG, EACP, EOCP\} \in ECCP$, l'algorithme construit un pattern P tel que :

- $P = PR \oplus EP_1$ (ligne 14)
- P ne garde que les feuilles de PR indiquées dans FG, donc ne contient aucune feuille redondante (ligne 15)
- Les occurrences de P sont les occurrences de $EOCP$ (boucle de la ligne 17 et ligne 18)
- Les mappings de P correspondent à l'ancre $EACP$ (ligne 19)

Montrons que l'ensemble EP des patterns obtenus est bien $EPatt(PR)$ de la définition 51.

Pour un ensemble d'accrochage multiple immédiat donné EP_1, et plus précisément pour son contexte d'accrochage multiple $CAI = (EP_1, EAAI(EP_1), EOAI(EP_1))$, on a découvert l'ensemble $ECCP(PR, CAI)$. Chaque élément $CCP \in ECCP(PR, CAI)$ permet de construire un pattern P, comme indiqué précédemment. Chaque pattern P ainsi construit vérifie les conditions de la définition de $EPatt(PR, EP_1)$ (définition 50). La fermeture de l'ensemble des patterns construits vient de la fermeture de $ECCP(PR, CAI)$: les seules différences entre les patterns créés sont les ensembles de feuilles supprimées. Comme $ECCP(PR, CAI)$ est fermé par rapport aux feuilles à supprimer, alors à partir de $ECCP(PR, CAI)$ on construit $EPatt(PR, CAI)$.

On en déduit que les patterns créés à partir de $\cup_{CAI \in ECAI} ECCP(PR, CAI)$ forment $\cup_{CAI \in ECAI} EPatt(PR, CAI)$.

Les patterns construits par *AccrochageSuppr* le sont à partir de $ECCP = Ferme(\cup_{CAI \in ECAI} ECCP(PR, CAI))$.

Tout pattern créé $P \in EP$ est donc dans $\cup_{CAI \in ECAI} EPatt(PR, CAI)$.

Montrons que l'on a la fermeture. Supposons par l'absurde que ce ne soit pas le cas. Alors il existe P_1 et P_2 dans EP tels que :
- $P_1 \sqsubset P_2$
- $Locc(P_1, D) = Locc(P_2, D)$

P_1 et P_2 ne peuvent pas provenir de l'accrochage du même ensemble d'accrochage multiple immédiat EP_1 sur PR, car cela contredirait la fermeture de $EPatt(PR, EP_1)$, or nous avons vu qu'elle était garantie par la fermeture de $ECCP(PR, EP_1)$.

Donc P_1 provient d'un ensemble d'accrochage multiple immédiat EP_1 et P_2 provient d'un ensemble d'accrochage multiple immédiat EP'_1 distinct.

Comme $P_1 \sqsubset P_2$, on en déduit $EP_1 \subsetneq EP'_1$.

P_1 et P_2 proviennent des contextes de création de pattern $CCP = (EP_1, FG, EACP, EOCP)$ et $CCP' = (EP'_1, FG', EACP', EOCP')$. Les occurrences de P_1 et P_2 correspondent respectivement à $EOCP$ et $EOCP'$, d'où $EOCP = EOCP'$. On en déduit donc que l'ensemble des contextes de création de patterns $ECCP$ n'est pas fermé, d'où une contradiction.

Donc $EP = EPatt(PR)$. □

Mise à jour du graphe des accrochages

AccrochageSuppr ajoute des patterns à $\mathcal{F}_D^i$, ce qui crée l'ensemble $\mathcal{F}_D^{i+1}$. Toutefois, aucun arc n'est rajouté dans le graphe des accrochages vers ces patterns. De plus, la fermeture de $\mathcal{F}_D^{i+1}$ peut être compromise par l'ajout des nouveaux patterns. La procédure $MiseAJour$, donnée dans l'algorithme 29, se charge de remettre à jour le graphe des accrochages et d'assurer la fermeture de $\mathcal{F}_D^{i+1}$, afin de préparer l'itération suivante.

Propriété 14 *Après l'exécution de MiseAJour, le graphe des accrochages contient tous les arcs vers les patterns ajoutés par AccrochageSuppr, correctement étiquetés.*

Preuve : La procédure $MiseAJour$ reçoit en paramètre une structure NPI qui contient l'ensemble des patterns créés par *AccrochageSuppr*, et pour chacun de ces patterns P le pattern racine PR dont il provient, ainsi que son contexte d'accrochage immédiat $CAI = (EP_1, EAAI, EOAI)$.

Or pour un pattern P résultant de l'accrochage de EP_1 sur PR, les seuls patterns de $\mathcal{F}_D^{i+1}$ qui peuvent s'accrocher sur lui sont les patterns qui s'accrochent sur les patterns de EP_1. Les deux boucles des lignes 5 et 6 se chargent d'examiner ces patterns. Soit P_X un pattern qui s'accroche sur $P_j \in EP_1$. Alors on a : $Ancre(P, P_X) = \{(o, o_x) \mid (o, ..., o_j, ...) \in EAAI \wedge (o_j, o_x) \in Ancre(P_j, P_X)\}$. La construction de l'ensemble $NewAncre$, lignes 8 à 12, respecte bien ces contraintes, donc $NewAncre = Ancre(P, P_X)$. Un arc de P_X vers P n'est ajouté que si $Ancre(P, P_X)$ n'est pas vide. La définition du graphe des accrochages (définition 36) est bien respectée, donc la propriété est vraie. □

Propriété 15 *Après l'exécution de MiseAJour, $\mathcal{F}_D^{i+1}$ est fermé.*

Preuve : Ceci est immédiat, les calculs effectués des lignes 31 à 35 éliminent tous les patterns qui ne respectent pas la propriété de fermeture de $\mathcal{F}_D^{i+1}$. □

Consommation des éléments d'ancres utilisés : Afin de simplifier les itérations suivantes, $MiseAJour$ consomme tous les éléments d'ancres utilisés dans l'itération courante pour construire des patterns fréquents, des lignes 19 à 27. Ces éléments ne serviront en effet plus jamais dans les itérations ultérieures, par contre il ne faut pas qu'ils soient réutilisés pour produire indéfiniment les mêmes patterns.

Exemple : Continuons notre exemple sur le datatree D_Z. Les patterns P_{11} et P_{12} sont tous deux éliminés de $\mathcal{F}_D^{i+1}$, car leurs ensembles d'occurrences sur D_Z sont inclus dans leurs ensembles d'occurrences sur $projete(P_{23}, D_Z)$. Le nouveau graphe des accrochages est donc celui de la figure V.22.

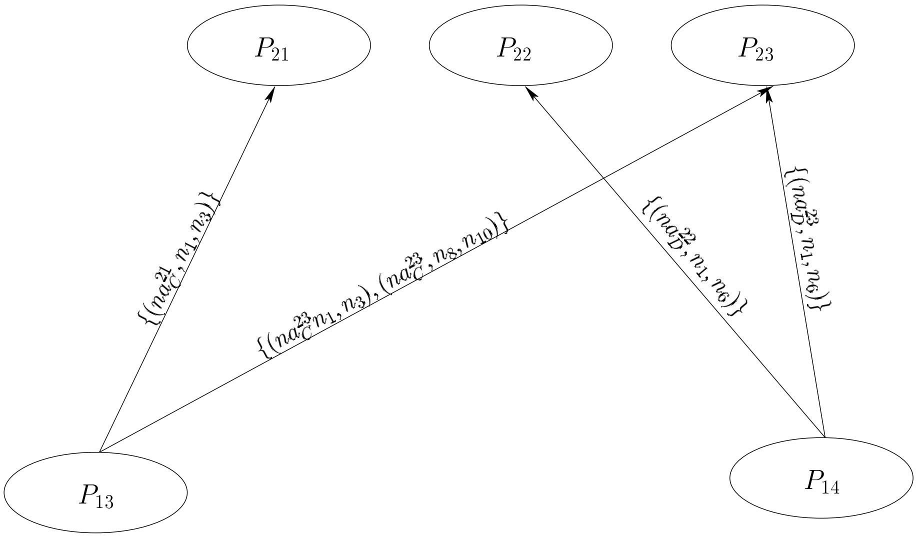


Fig. V.22 – Nouveau graphe des accrochages pour D_Z

4.4 Correction et complétude de Dryade

Nous allons montrer dans cette section que l'algorithme Dryade termine, et qu'il est correct et complet.

Lemme 2 (Terminaison) *Pour tout datatree d'entrée D et seuil ε, l'algorithme* Dryade *termine. Lorsque* Dryade *termine, il n'y a plus aucun arc dans le graphe $\mathcal{G}$ des accrochages.*

Preuve : L'algorithme Dryade termine s'il existe une itération k telle que pour tout pattern racine PR de $\mathcal{F}_D^k$, l'ensemble $ECAI$ calculé à partir de PR est vide. Pour que cet ensemble soit vide, il faut qu'on ne puisse faire aucun accrochage multiple immédiat sur PR. Or s'il existe au moins un pattern P qui s'accroche sur PR, soit l'accrochage est immédiat, soit il n'est pas immédiat et dans ce cas il existe un pattern P' tel que P s'accroche sur P' et P' s'accroche sur PR. Donc P' est en accrochage immédiat avec PR.

On en déduit que pour que Dryade termine, il faut qu'il n'y aie plus aucun arc dans le graphe $\mathcal{G}$ des accrochages : plus aucun pattern de $\mathcal{F}_D^k$ ne s'accroche sur un autre pattern de $\mathcal{F}_D^k$. Or à chaque itération, la procédure $MiseAJour$ consomme un nombre non nul d'arcs de $\mathcal{G}$.

Algorithme 29 *MiseAJour*

Entrée : $\mathcal{F}_D^{i+1}$ l'ensemble de patterns fréquents découverts, $\mathcal{G}$ le graphe des accrochages, NPI ensemble d'informations sur tous les patterns créés dans l'itération courante, au format (Pattern créé, Pattern racine d'origine, Contexte d'accrochage immédiat).

Sortie : G le graphe des accrochages est mis à jour, $\mathcal{F}_D^{i+1}$ est bien fermé

Pour tout $(EP, PR, CAI = \{EP_1, EAAI, EOAI\}) \in NPI$ **Faire**
  // Tous les nouveaux patterns de EP viennent de l'accrochage des patterns de EP_1 sur PR
  Pour tout $P \in EP$ **Faire**
    // On ajoute des arcs vers le nouveau pattern
    Pour tout $P_1 \in EP_1$ **Faire**
      Pour tout $P_X \in pred(P_1)$ **Faire**
        $NewAncre \leftarrow \emptyset$
        Pour tout $(o_1, o_2) \in Ancre(P_1, P_X)$ **Faire**
          Si $\exists (o_R, o_1) \in EAAI$ **Alors**
            $NewAncre \leftarrow NewAncre \cup \{(o_1, o_2)\}$
          Fin Si
        Fin Pour
        Si $NewAncre \neq \emptyset$ **Alors**
          Créer un arc dans $\mathcal{G}$ de P_X vers P, d'étiquette $NewAncre$
        Fin Si
      Fin Pour
    Fin Pour
    // Mise à jour des étiquettes pour les ancres utilisées dans cette itération de Dryade
    $CCP = (EP_1, FG, EACP, EOCP) \leftarrow$ contexte de création de pattern pour P
    Pour tout arc (PR, P_j), $P_j \in EP_1$ **Faire**
      Pour tout $(o_R, o_1, ..., o_n, o_{n+1}, ..., o_k) \in EACP$ **Faire**
        Supprimer (o_R, o_j) de l'étiquette de (PR, P_j)
      Fin Pour
      Si l'étiquette est vide **Alors**
        Supprimer l'arc (PR, P_j)
      Fin Si
    Fin Pour
  Fin Pour
Fin Pour
// Garantir la fermeture de $\mathcal{F}_D^{i+1}$
Pour tout $P \in \mathcal{F}_D^{i+1}$ **Faire**
  Si $\exists P' \in \mathcal{F}_D^{i+1}$ tq $P \sqsubset P'$ ET $Locc(P, D) = Locc(P, projete(P', D))$ **Alors**
    $\mathcal{F}_D^{i+1} \leftarrow \mathcal{F}_D^{i+1} - P$
  Fin Si
Fin Pour

Soit i une itération de DRYADE. Soit P un pattern de $\mathcal{F}_D^i$. Appelons *distance de racine* le plus grand nombre d'arc séparant P d'un pattern racine de $\mathcal{F}_D^i$ dans $\mathcal{G}$. Alors à chaque itération, les arcs consommés par $MiseAJour$ diminuent de 1 la distance de racine de P (si P n'est pas éliminé du graphe). En effet, soit PR le pattern racine pour lequel la distance de racine de P est maximale. Il existe donc une chaîne maximale de patterns dans le graphe : $P \to P_1 \to .... \to P_n \to PR$. Par définition, dans cette chaîne P_n est en accrochage immédiat sur PR. La procédure $AccrochageSuppr$ va donc construire un pattern P' résultant (entre autres) de l'accrochage de P_n sur PR. La procédure $MiseAJour$ va ensuite supprimer les accrochages entre P_n et PR. Des accrochages vers P' vont aussi être rajoutés par $MiseAJour$, les différentes chaînes pouvant être produites sont :

- $P \to P_1 \to ... \to P_n$ (si P_n n'est pas éliminé)
- $P \to P_1 \to ... \to P_{n-1} \to PR$ (si PR n'est pas éliminé)
- $P \to P_1 \to ... \to P_{n-1} \to P'$

Toutes ces chaînes ont une longueur inférieure de 1 à la chaîne $P \to P_1 \to .... \to P_n \to PR$. On en déduit que la distance de racine de P diminue effectivement de 1 à chaque itération.

Donc pour tout pattern de $\mathcal{F}_D^i$, il existe une itération où soit il est éliminé, soit sa distance de racine est nulle, ce qui signifie qu'il est un pattern racine lui-même.

Si P est racine, alors pour ses occurrences de racine il ne s'accroche sur aucun pattern. Pour ses autres occurrences il s'accroche sur des patterns qui sont dans la même classe d'équivalence de cycle que lui, donc qui sont racines aussi : ces accrochage là vont donc aussi être supprimés dans les itérations ultérieures.

On en déduit qu'il existe une étape à laquelle $\mathcal{G}$ n'a plus d'arcs, et donc DRYADE termine. □

Lemme 3 *L'ensemble fermé $\mathcal{F}_1$ des patterns fréquents de profondeur 1 est l'ensemble fermé des applatissements des patterns de l'ensemble fermé $\mathcal{F}$ des patterns fréquents et des applatissements des sous-arbres des patterns de $\mathcal{F}$.*

Preuve : Sens $\supset$: Pour $Q \in \mathcal{F}$, nous avons vu que son applatissement est dans $\mathcal{F}_1$ (propriété 8). Soit T un sous-arbre d'un pattern $Q \in \mathcal{F}$. On en déduit que $P = plat(T)$ est fréquent. Si $P \notin \mathcal{F}_1$, c'est qu'il a été éliminé par la propriété de fermeture. Il existe donc $P' \in \mathcal{F}_1$ tel que $P \sqsubset P'$ et $Locc(P, D) = Locc(P', D)$. On en déduit $Locc(P', D) = Locc(T, D)$ puisque P a les mêmes occurrences que T. Comme $P \sqsubset P'$, il existe au moins une étiquette e qui est dans P' mais pas dans P, et pour les occurrences de P' et donc de T cette étiquette est toujours descendante de la racine de P' (et donc de T). On en déduit que l'arbre T' résultant de l'ajout à T d'un fils d'étiquette e à la racine est fréquent, avec les mêmes occurrences que T. Comme $T \sqsubset T'$, cela contredit la fermeture de $\mathcal{F}$. Donc $plat(T) \in \mathcal{F}_1$.

Sens $\subset$: Soit $P \in \mathcal{F}_1$. Montrons qu'il existe un pattern $Q \in \mathcal{F}$ tel que soit $P = plat(Q)$, soit $P = plat(T)$ où T est un sous-arbre de Q. Le premier cas est un cas particulier du second, où $T = Q$, nous ne montrerons donc que ce cas. Supposons par l'absurde qu'il n'existe pas de pattern $Q \in \mathcal{F}$ tel que P est un applatissement d'un sous-arbre de Q. Comme le pattern P est fréquent, on doit retrouver ses occurrences dans les patterns de $\mathcal{F}$. Comme P n'est pas l'applatissement d'un sous-arbre d'un pattern de $\mathcal{F}$, alors cela signifie qu'il existe un pattern $Q \in \mathcal{F}$ qui a un sous-arbre T' tel que $Locc(T', D) = Locc(P, D)$, et T' a au moins une étiquette de plus que P (tout sous-arbre dont P pourrait être l'applatissement est donc inclus dans T'). On en déduit que $P \sqsubset plat(T')$. Cela contredit la fermeture de $\mathcal{F}_1$. Donc P est bien l'applatissement d'un sous-arbre d'un pattern de $\mathcal{F}$. □

Définition 52 (Troncage d'un pattern) *Soit P un pattern de profondeur m, et soit $m' < m$. Le troncage à profondeur m' de P est le pattern $P_{|m'}$ dont les nœuds sont les nœuds de profondeur* *inférieure ou égale à m' de P et dont les arcs sont les arcs entre ces nœuds dans P.*

Définition 53 (Pattern $m-1$-générateur) *Soit D une base de donnée d'arbres et $\mathcal{F} = Ferme(\mathcal{P})$ l'ensemble fermé des patterns fréquents de D. Soit P un pattern fréquent dans D de profondeur m. Soit $\{Q_1, ..., Q_n\}$ un ensemble de patterns de $\mathcal{F}$.*

*On dit que P est $m-1$-**générateur** pour $\{Q_1, ..., Q_n\}$ si $\forall i \in [1, n]$:*
- *$P \sqsubseteq Q_i$*
- *$Locc(Q_i, D) \subseteq Locc(P, D)$*
- *les troncages de P et Q_i à profondeur $m-1$ sont isomorphes par un isomorphisme μ.*
- *tout sous-arbre de P dont la racine r est à la profondeur $m-1$ est l'applatissement du sous-arbre de Q_i dont la racine est $\mu(r)$ (par définition $\mu(r)$ est aussi à la profondeur $m-1$).*

Exemple : Supposons que tous les patterns de la figure V.23 ont le même ensemble d'occurrences. Alors le pattern P est 1-générateur pour les patterns Q_1 et Q_2.

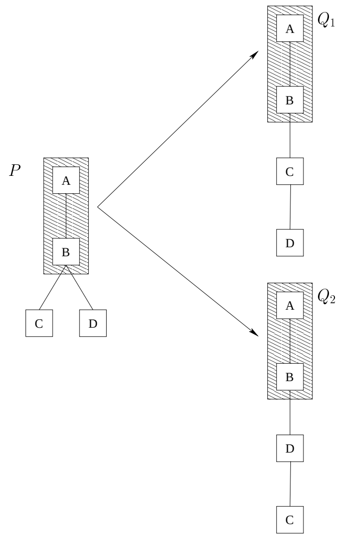


FIG. V.23 – Exemple de pattern 1-générateur

Lemme 4 (Caractérisation des résultats intermédiaires de DRYADE) *À chaque itération i, l'algorithme DRYADE produit un ensemble fermé $\mathcal{F}_D^i$ de patterns fréquents.*

Cet ensemble se décompose en trois sous-ensembles disjoints : $\mathcal{F}_D^i = F_{\mathcal{F}}^i \cup F_R^i \cup F_1^i$, où :
- *$F_{\mathcal{F}}^i$ est l'ensemble des patterns qui ne sont impliqués dans aucun accrochage à l'étape i.*
- *F_R^i est l'ensemble des patterns racines de $\mathcal{F}_D^i$ qui n'appartiennent pas à $F_{\mathcal{F}}^i$.*
- *F_1^i est l'ensemble des patterns de profondeur 1 qui s'accrochent entre eux ou sur les patterns de F_R^i.*

Ces ensembles vérifient les propriétés suivantes :

i. *Pour tout pattern $Q \in \mathcal{F}$, il existe un unique pattern $P \in \mathcal{F}_D^i$ de profondeur m qui est $m-1$-générateur pour Q, tel qu'il n'existe pas de patterns $m'-1$-générateurs pour Q dans $\mathcal{F}_D$, avec $m' > m$. On dira que P est le plus gros générateur de Q, noté $P = gen_i(Q)$. Si $P \in F_{\mathcal{F}}^i$, alors $P = Q$ (les patterns de $F_{\mathcal{F}}^i$ sont les patterns de $\mathcal{F}$ déjà découverts par DRYADE à l'étape i).*

ii. *Pour tout pattern racine $P \in F_R^i$ il existe au moins un pattern $Q \in \mathcal{F}$ tel que $P = gen(Q)$, et $Locc(Q, D) \subseteq OR(P)$.*

iii. *Pour tout pattern $Q \in \mathcal{F}$, soit m la profondeur de son plus gros générateur. Pour chaque sous-arbre T de Q dont la racine a une profondeur supérieure ou égale à m, $plat(T) \in F_1^i \cup F_R^i$.*

iv. *Pour chaque pattern $P_1 \in F_1^i$ il existe un pattern $Q \in \mathcal{F}$ dont le plus gros générateur est de profondeur m et qui a un sous-arbre T dont la racine est à une profondeur supérieure ou égale à m tel que $P_1 = plat(T)$.*

Preuve :

Soit i une itération de DRYADE**. Montrons que $\mathcal{F}_D^i$ se décompose en $F_{\mathcal{F}}^i$, F_R^i et F_1^i :**

Considérons $\mathcal{GQ}_{\mathcal{F}_D^i}$ le graphe quotient des accrochages de $\mathcal{F}_D^i$.

– $F_{\mathcal{F}}^i$ est l'ensemble des patterns de $\mathcal{F}_D^i$ dont qui ne sont connecté à aucun autres patterns par accrochages.

– F_R^i est l'ensemble des patterns dont les classes d'équivalence de cycles sont racines de $\mathcal{GQ}_{\mathcal{F}_D^i}$.

– F_F^i est l'ensemble des patterns de $\mathcal{F}_D^i$ qui ne sont ni dans $F_{\mathcal{F}}^i$ ni dans F_R^i.

Ces ensembles sont bien disjoints, et par construction $\mathcal{F}_D^i = F_{\mathcal{F}}^i \cup F_R^i \cup F_1^i$.

Montrons maintenant la vérification des propriétés i, ii, iii et iv pour toute itération i de DRYADE

Nous allons prouver par récurrence sur les itérations de DRYADE que pour chaque itération i de DRYADE que les propriétés **i**, **ii**, **iii** et **iv** sont vérifiées.

Cas de base : $i = 1$

Nous supposerons que la première itération de DRYADE est la construction de l'ensemble de patterns fréquents fermés de profondeur 1.

D'après la propriété 10, on a $\mathcal{F}_D^1 = \mathcal{F}_1 = Ferme(\mathcal{P}_1)$.

Montrons la vérification de la propriété i :

Soit $Q \in \mathcal{F}$. D'après la propriété 8, il existe $P \in \mathcal{F}_1$, $P = plat(Q)$.

De plus, P et Q ont les mêmes racines, donc P est 0-générateur pour Q.

D'après la propriété 8, P est unique. De plus, comme tous les patterns de $\mathcal{F}_D^1$ sont de profondeur 1, il ne peut pas y avoir de patterns m-générateurs pour Q, avec $m > 0$.

Supposons que P soit un pattern de $F_{\mathcal{F}}^1$, c'est-à-dire que P ne s'accroche sur aucun pattern de $\mathcal{F}_D^1$, et aucun pattern de $\mathcal{F}_D^1$ ne s'accroche sur P. Donc Q n'a aucun sous-arbres. Sinon d'après le lemme 3 il existerait dans $\mathcal{F}_D^1$ des patterns étant les applatissements de ces sous-arbres, et ces patterns s'accrocheraient sur P. Q est donc un pattern de profondeur 1. Par construction de P, on en déduit $P = Q$.

Donc la propriété **i** est vérifiée.

Montrons la vérification de la propriété ii :

Soit $P \in F_R^1$. Nous devons montrer que P est 0-générateur pour au moins un pattern $Q \in \mathcal{F}$. Pour cela, d'après le lemme 3, nous devons montrer que P n'est pas l'applatissement d'un sous-arbre d'un pattern de $\mathcal{F}$ (il est donc uniquement l'applatissement d'un pattern $Q \in \mathcal{F}$).

Pour l'ensemble $OR(P)$ de ses occurrences de racine, P ne s'accroche sur aucun pattern. On en déduit que pour ces occurrences, P ne peut pas être l'applatissement d'un sous-arbre d'un pattern de $\mathcal{F}$, sinon il y aurait des accrochages. Donc pour ses occurrences de racine, P est l'applatissement d'un pattern $Q \in \mathcal{F}$. Montrons de plus que $Locc(Q, D) \subseteq OR(P)$. Supposons par l'absurde que ce ne soit pas le cas : alors pour certaines de ses occurrences, Q est un sous-arbre d'un pattern Q'. Dans ce cas, il y a un arc dans le graphe des accrochages entre P et $P' = plat(Q')$. Pour que P soit malgré tout racine, il faudrait donc que P' soit dans la même classe d'équivalence de cycle que P. Les occurrences de P' sur lesquelles s'accrochent P ne peuvent pas être prolongées pour faire le cycle et revenir à P, sinon on aurait $Q \sqsubset Q' \sqsubset ... \sqsubset Q$, ce qui est impossible. Il existe donc un pattern Q'' tel que P' est générateur pour Q'', les occurrences de Q'' sont différentes de celles que Q', et elles s'accrochent (en passant par les autres patterns du cycle) aux occurrences de P. On en déduit $Q'' \sqsubset Q \sqsubset Q'$. Donc les occurrences de Q' sont aussi des occurrences de Q'', d'où une nouvelle contradiction. P n'est donc pas racine dans ce cas.

On en déduit que la propriété **ii** est vérifiée.

Montrons la vérification des propriétés iii et iv :

D'après **i**, pour tout pattern $Q \in \mathcal{F}$ son plus gros générateur existe et est 0-générateur. Les sous-arbres de Q qui nous intéressent dans cette propriété sont donc tous les sous-arbres de Q sauf Q lui-même.

Montrons d'abord que pour tout pattern $Q \in \mathcal{F}$, les patterns correspondants à l'applatissement de ses sous-arbres sont dans $F_1^1 \cup F_R^1$.

- Soit $Q \in \mathcal{F}$. D'après le lemme 3, pour tout sous-arbre T de Q il existe un pattern $P \in \mathcal{F}_D^1$ qui est un applatissement de T. Comme T est un sous-arbre de Q, pour ses occurrences correspondant aux occurrences de T, P s'accroche sur $plat(Q) \in \mathcal{F}_D^1$. Puisque P a des accrochages, alors P n'est pas dans $F_{\mathcal{F}}^1$. Donc $P \in F_R^1 \cup F_1^1$.

La propriété **iii** est donc vérifiée.

Montrons ensuite que tout pattern $P \in F_1^1$ est l'applatissement d'un sous-arbre d'un pattern de $\mathcal{F}$.

- Soit $P \in F_1^1$. D'après le lemme 3, P est l'applatissement de patterns de $\mathcal{F}$ et/ou de sous-arbres de patterns de $\mathcal{F}$. Supposons par l'absurde qu'il n'existe pas T tel que $P = plat(T)$ et T est un sous-arbre d'un pattern $Q \in \mathcal{F}$. Alors P n'est l'applatissement que de patterns de $\mathcal{F}$ qui ne sont jamais sous-arbres d'autres patterns de $\mathcal{F}$. On en déduit que P n'a aucun accrochage avec d'autres patterns de $\mathcal{F}_D^1$. Ce qui contredit $P \in F_1^1$.

La propriété **iv** est donc vérifiée.

Ainsi, le cas de base est vérifié.

Cas général :

Supposons la propriété vraie à l'itération i, c'est à dire que l'on a $\mathcal{F}_D^i = F_{\mathcal{F}}^i \cup F_R^i \cup F_1^i$, vérifiant **i**, **ii**, **iii** et **iv**. Montrons que la propriété est toujours vraie à l'itération $i+1$.

Montrons que la propriété i est vérifiée

Soit $Q \in \mathcal{F}$. Par hypothèse de récurrence il existe $P \in \mathcal{F}_D^i$ tel que $P = gen_i(Q)$. On pose $m = profondeur(P)$. Trois cas se présentent :

- $P \in F_{\mathcal{F}}^i$. Dans ce cas, toujours par hypothèse de récurrence $P = Q$. On ne peut donc pas trouver de plus gros générateur pour Q que P, de plus comme P ne participe à aucun accrochage il reste inchangé à l'itération $i+1$. On en déduit $P = gen_{i+1}(Q)$.
- $P \in F_R^i$. Soient $P_1, ..., P_n$ les patterns obtenus par accrochage et suppression des feuilles redondantes à partir de P. Montrons qu'il existe $j \in [1, n]$ tel que $P_j = gen_{i+1}(Q)$.

  Par définition, P est $m-1$-générateur pour Q. Soient $T_1, ..., T_k$ tous les sous-arbres de Q dont la racine est à une profondeur supérieure à $m-1$. Par hypothèse de récurrence, propriété **iii**, les applatissements de tous ces sous-arbres sont dans $F_R^i \cup F_1^i$. Puisque Q est fréquent, alors l'accrochage multiple des applatissements de $T_1, ..., T_k$ est fréquent aussi. Donc dans $EAM(P)$, on trouve $EP_1 = \{(na_1, plat(T_1)), ..., (na_k, plat(T_k))\}$, et l'ensemble des occurrences d'accrochage multiple de EP_1 contient au moins toutes les occurrences de Q. EP_1 est bien fermé car on ne peut trouver dans $EAM(P)$ d'ensemble $EP_1' = \{(na_1, plat(T_1)), ..., (na_k, plat(T_k)), (na', P')\}$ pour les occurrences de Q. En effet dans ce cas P' ne pouvant pas être l'applatissement d'un sous-arbre

de Q (ils sont déjà tous représentés), il doit donc être inclus dans l'applatissement de l'un de ces sous-arbres. Ce n'est pas possible par construction de $EAM(P)$.
Considérons maintenant l'ensemble $\{T'_1, ..., T'_l\} \subseteq \{T_1, ..., T_k\}$ des sous-arbres de Q dont la racine est exactement à la profondeur m. Toutes les racines des autres sous-arbres de profondeur inférieure sont des descendantes des racines de $\{T'_1, ..., T'_l\}$. On en déduit que $plat(T'_1), ..., plat(T'_l)$ sont les seuls patterns qui ne s'accrochent sur aucun pattern de $plat(T_1), ..., plat(T_k)$. Ils satisfont donc la définition d'accrochage immédiat. Donc il existe dans $ECAI(PR)$ un ensemble d'accrochage immédiat $\{(na'_1, plat(T'_1)), ..., (na'_l, plat(T'_l))\}$, dont les occurrences contiennent les occurrences de Q. Les racines de $T'_1, ..., T'_l$ sont les nœuds internes de profondeur m de Q.
Soient $f_1, ..., f_p$ les feuilles de profondeur m de Q. Par construction, ces feuilles sont aussi des feuilles de P. À ces feuilles correspondent des nœuds dans les données, que l'on ne retrouve évidemment dans aucun des sous-arbre $T'_1, ..., T'_l$. Donc ces nœuds ne se retrouvent pas plus dans les nœuds correspondants aux feuilles de $plat(T'_1), ..., plat(T'_l)$. Ils satisfont donc la définition de feuille non redondante. On en déduit que l'on va trouver dans $ECCP(PR)$ un contexte de création de pattern $CCP = (\{(na'_1, plat(T'_1)), ..., (na'_l, plat(T'_l))\}, \{f_1, ..., f_p\}, EACP, EOCP)$, dont les occurrences contiennent les occurrences de Q.
Le pattern P_j que l'on va construire à partir de CCP sera donc un pattern m-générateur pour Q.
Pour que $P_j = gen_{i+1}(Q)$, il faut montrer que :

- P_j n'est pas éliminé lors de la mise à jour.
  - Supposons par l'absurde que P_j soit éliminé lors de la mise à jour. Cela signifie qu'il n'est pas fermé, donc qu'il existe $P' \in \mathcal{F}_D$ tel que $P_j \sqsubset P'$ et $Locc(P_j, D) = Locc(P_j, projete(P, D))$. De plus nous savons que les occurrences de Q sont incluses dans celles de P_j, puisque P_j est m-générateur pour Q.
    Supposons d'abord que $racine(P_j) \neq racine(P')$. Alors pour toutes les occurrences de Q, la racine de Q a comme ancêtre la racine de P'. Cela contredit la fermeture de $\mathcal{F}$.
    Supposons maintenant que $racine(P_j) = racine(P')$. P' ne peut pas provenir du pattern P à l'origine de P_j, puisque pour P on a construit $EPatt(P)$, qui est fermé. Donc soit P' a des nœuds de profondeur inférieure à m en plus des nœuds de P, soit P' à des nœuds de profondeur supérieure ou égale à m en plus des nœuds de P, et qui ont des étiquettes qui ne sont pas dans les étiquettes de P. Dans les deux cas, ces nœuds en plus existent pour toutes les occurrences de Q mais ne peuvent pas être dans Q, ce qui contredit la fermeture de $\mathcal{F}$.
    Donc P_j n'est pas éliminé lors de la fermeture.
- P_j est bien l'unique plus gros m-générateur pour Q.
  - Nous savons par hypothèse de récurrence que P est l'unique plus gros $m-1$-générateur de Q. Il existe par construction un unique ensemble d'accrochage multiple de EAM qui contienne exactement les applatissements des sous-arbres de Q. De même, par construction il existe un unique ensemble d'accrochage multiple immédiat dans $ECAI(P)$ qui contienne les applatissements des sous-arbres de profondeur m de Q. Et enfin il existe par construction un unique contexte de construction de pattern de $ECCP(P)$ qui contienne les applatissements des sous-arbres de profondeur m de Q et les feuilles de profondeur m de Q. D'où l'unicité de P_j.
    P_j est bien maximal : sinon, cela signifierait qu'il existe un autre pattern racine P' qui a produit un pattern au moins $m+1$-générateur pour Q. Or comme par accrochage et suppression des feuilles redondantes à partir d'un pattern k-générateur on peut au mieux arriver à un pattern $k+1$-générateur, alors P' serait lui-même m-générateur pour Q, ce qui contredit le fait que $P = gen_i(Q)$.

– $P \in F_1^i$. Nous devons montrer que P n'est pas éliminé par $MiseAJour$, et qu'il reste donc le plus gros générateur de Q à l'itération $i+1$. Si P n'est pas en accrochage immédiat avec un pattern racine $PR \in F_R^i$, alors il n'est pas examiné par $MiseAJour$, et reste donc tel quel. Sinon, P est supprimé par $MiseAJour$ s'il existe un pattern P' produit à partir de PR et P, non éliminé lui-même, tel que toutes les occurrences de P se déduisent des occurrences de P'. Puisque P' n'est pas éliminé, d'après la démonstration précédente il existe $Q' \in \mathcal{F}$ tel que $P' = gen_{i+1}(Q')$. On a nécéssairement $Q \sqsubset Q'$, donc par fermeture de $\mathcal{F}$ on a : $Locc(Q, D) \supseteq Locc(Q, projete(Q', D))$, c'est-à-dire qu'il existe des occurrences de Q qui ne sont pas induites par l'inclusion de Q dans Q'. Les occurrences de P correspondantes ne sont donc pas inclues dans celles de P', donc P n'est pas éliminé.
On en déduit que P reste bien le plus gros générateur de Q : $P = gen_{i+1}(Q)$.

La propriété **i** est donc vérifiée dans le cas général.

Montrons que la propriété ii est vérifiée

Nous devons montrer que pour tout pattern racine $P \in F_R^{i+1}$, il existe un pattern $Q \in \mathcal{F}$ tel que $P = gen_{i+1}(Q)$. Les patterns de F_R^{i+1} ont deux origines possibles :

1. Soit P a été construit par accrochage et suppression des feuilles redondantes à partir d'un pattern $PR \in F_R^i$.

2. Soit P est un pattern de F_1^i qui est devenu racine après la suppression de certains de ses accrochages par $MiseAJour$.

Nous allons montrer que dans ces deux cas il existe $Q \in \mathcal{F}$ tel que $P = gen_{i+1}(Q)$.

Soit $P \in F_R^{i+1}$.

1. P a été construit par accrochage et suppression de feuilles à partir de $PR \in F_R^i$. Donc on peut associer à P un contexte de création de pattern $CCP \in ECCP(PR)$, auquel on peut associer un ensemble d'accrochage multiple immédiat $EP_1 \in ECAI(PR)$, auquel on peut associer enfin un ensemble d'accrochage multiple $AM \in EAM(PR)$. Soient $P_1, ..., P_n$ les patterns de AM, montrons qu'il existe $Q \in \mathcal{F}$ contenant PR tel que $P_1, ..., P_n$ sont tous des applatissements de sous-arbres de Q au niveau des feuilles de PR.

Supposons par l'absurde qu'il n'existe aucun Q contenant PR dont les applatissements des sous-arbres au niveau des feuilles de PR soient $P_1, ..., P_n$.

Pourtant l'accrochage des patterns $P_1, ..., P_n$ sur PR est fréquent et fermé, les relations d'ancestralité correspondantes doivent donc apparaître dans au moins un pattern $Q \in \mathcal{F}$.

- Soit $Q \in \mathcal{F}$ dont les applatissements des sous-arbres s'accrochant sur les feuilles de PR sont $P_1, ., P_i', .., P_n$, avec $P_i \sqsubset P_i'$ et P_i, P_i' s'accrochent sur la même feuille de PR. Ce cas ne peut pas arriver, car par construction les patterns $P_1, ..., P_n$ sont maximaux au sens de l'inclusion d'arbre pour l'accrochage sur une feuille donnée de PR.
- Soit $Q \in \mathcal{F}$ dont les applatissements des sous-arbres s'accrochant sur les feuilles de PR sont $\{P_1', ..., P_{n'}'\} \supset \{P_1, ..., P_n\}$. Comme précédemment, ce cas ne peut pas arriver : $P_1', ..., P_{n'}'$ s'accrochent sur les nœuds d'accrochages de PR, cela supposerait que la découverte de l'ensemble fermé $EAM(PR)$ est incorrecte, or nous avons prouvé que ce n'est pas le cas.
- Soit $Q \in \mathcal{F}$ dont les applatissements des sous-arbres s'accrochant sur les feuilles de PR sont $P_1, ., P_i', .., P_n$, avec $P_i' \sqsubset P_i$. Ce cas ne peut pas arriver, car ici P_i a au moins un nœud de plus que P_i', et ce nœud est fréquent. Or Q n'a pas ce nœud, cela contredirait la fermeture de $\mathcal{F}$.
- Soit $Q \in \mathcal{F}$ dont les applatissements des sous-arbres s'accrochant sur les feuilles de PR sont $\{P_1', ..., P_{n'}'\} \subset \{P_1, ..., P_n\}$. Ce cas non plus ne peut pas arriver, pour la même rason que précédemment : en effet l'accrochage de tout $P_i \in \{P_1, ..., P_n\} \backslash \{P_1', ..., P_{n'}'\}$ est fréquent,

mais pas dans Q, il y a donc une contradiction avec la fermeture de $\mathcal{F}$.

Il existe donc au moins un pattern $Q \in \mathcal{F}$ contenant PR dont les applatissements des sous-arbres au niveau des feuilles de PR sont les patterns $P_1, ..., P_n$ de l'ensemble AM. Deux cas sont alors possibles : soit $PR = gen_i(Q)$, soit il existe $PR' \in F^i_R \cup F^i_{\mathcal{F}}$ tel que $PR \sqsubset PR' = gen_i(Q)$.

Nous devons ensuite montrer qu'il existe au moins un pattern $Q \in \mathcal{F}$ dont les applatissements des sous-arbres qui s'accrochent immédiatement sous les feuilles de PR, et qui ne sont inclus dans aucun autre sous-arbre de $P_1, ..., P_n$, sont les patterns $P'_1, ..., P'_{n'}$ de $EP_1 \in ECAI(PR)$, puis nous devons monter qu'il existe un pattern $Q \in \mathcal{F}$ pour lequel les feuilles de PR qui restent des feuilles dans Q sont les feuilles non redondantes trouvées dans le contexte de création de pattern $CCP \in ECCP(PR)$.

Nous ne détaillons pas les démonstrations de ces deux propriétés car elles suivent exactement le même raisonnement que la précédente, en se focalisant non plus sur $P_1, ..., P_n$ mais sur $P'_1, ..., P'_{n'}$ puis sur les feuilles non redondantes.

On en déduit qu'il existe au moins un pattern $Q \in \mathcal{F}$ contenant PR dont les applatissements des sous-arbres s'accrochant immédiatement sur les feuilles de PR sont les patterns de EP_1, et donc les feuilles communes aux feuilles de PR sont les feuilles non redondantes trouvées dans les contexte de création de pattern CCP.

Si $PR = gen_i(Q)$, alors P est m-générateur pour Q par construction. En s'appuyant sur la preuve de la propriété **i**, on obtient $P = gen_{i+1}(Q)$.

Sinon, il existe $PR' \in F^i_R \cup F^i_{\mathcal{F}}$ tel que $PR \sqsubset PR' = gen_i(Q)$, et les occurrences de P sont induites par les occurrences de PR'. Dans ce cas P doit être éliminé par la mise à jour, or comme P n'a pas été éliminé nous sommes donc dans la situation précédente : $PR = gen_i(Q)$.

2. P est un pattern de F^i_1 qui est devenu racine à la suite de la suppression de ses accrochages par $MiseAJour$. Nous avons vu dans la démonstration de la propriété **i** que s'il existe $Q \in \mathcal{F}$ tel que $P = gen_i(Q)$, alors P n'est pas éliminé. Montrons que dans le cas contraire, P est éliminé par $MiseAJour$. Puisque les accrochages de P ont été supprimés par $MiseAJour$, alors P était en accrochage immédiat avec un pattern racine $PR \in F^i_R$. Supposons par l'absurde que P n'est pas éliminé par $MiseAJour$. Cela signifie que pour tout pattern $P' \in \mathcal{F}^{i+1}_D$ tel que $P \sqsubset P'$, certaines occurrences de P ne sont pas induites par son inclusion dans P'. Comme les patterns de $\mathcal{F}$ sont inclus dans les patterns de $\mathcal{F}^{i+1}_D$, alors de même pour tout pattern $Q \in \mathcal{F}$ tel que $P \sqsubset Q$, certaines occurrences de P ne peuvent être déduites de son inclusion dans Q. Cela signifie que P est lui-même un pattern de $\mathcal{F}$. Cela n'est pas possible puisque nous avons ici supposé que P n'était générateur d'aucun pattern de $\mathcal{F}$. Donc si P n'est le plus gros générateur d'aucun pattern de $\mathcal{F}$, alors P est éliminé.

Puisque P n'est pas éliminé, alors il existe $Q \in \mathcal{F}$ tel que $P = gen_i(Q)$. Cela ne change pas au cours de l'itération de DRYADE donc $P = gen_{i+1}(Q)$.

La propriété **ii** est donc vérifiée.

Montrons que les propriétés iii et iv sont vérifiées

Les propriétés **iii** et **iv** sont vraies au rang i par hypothèse de récurrence. Donc pour tout $Q \in \mathcal{F}$, avec $P = gen_i(Q)$ de profondeur m, les applatissements des sous-arbres de Q dont la racine est à une profondeur supérieure ou égale à m sont dans $F^i_R \cup F^i_1$. Et pour tout $P \in F^i_1$, on peut montrer qu'il est l'applatissement d'un sous-arbre dont la racine est à une profondeur supérieure ou égale à m d'un pattern $Q \in \mathcal{F}$ dont le plus gros générateur est de profondeur m.

Montrons d'abord que pour tout pattern $Q \in \mathcal{F}$ dont le plus gros générateur à l'itération $i+1$ est de profondeur $m+1$, les applatissements de ses sous-arbres dont la racine a une profondeur supérieure ou égale à $m+1$ sont dans $F^{i+1}_1 \cup F^{i+1}_R$.

Pour les patterns $Q \in \mathcal{F}$ dont le plus gros générateur ne change pas à l'itération $i+1$, nous devons montrer qu'aucun des applatissements des sous-arbres de Q n'est éliminé. Supposons par l'absurde que ce soit le cas : soit P l'applatissement d'un sous-arbre de Q dont la racine dans Q est à une profondeur supérieure ou égale à m, supposons que P soit éliminé. Cela signifie que P était en accrochage immédiat avec un pattern PR, générateur pour des patterns de $\mathcal{F}$ autres que Q, et que toutes les occurrences de P peuvent se déduire des occurrences d'un pattern P' résultant de l'accrochage et de la suppression des feuilles redondantes de P sur PR. Cela signifierait que toutes les occurrences de Q (incluses dans celles de P) peuvent se déduire des occurrences de P', d'où une contradiction par rapport à la fermeture de $\mathcal{F}$.

Pour les patterns $Q \in \mathcal{F}$ dont le plus gros générateur change à l'itération $i+1$, cela signifie que le nouveau générateur P a été construit par accrochage et suppression des feuilles redondantes à partir de l'ancien générateur PR, comme vu dans les démonstrations précédentes. Dans ces cas, les seuls applatissements de sous-arbres de Q qui peuvent être supprimés par $MiseAJour$ sont les patterns en accrochage immédiat sur PR, correspondant aux sous-arbres dont la racine est à la profondeur m dans Q. Les applatissements des sous-arbres dont la racine est à une profondeur supérieure ou égale à $m+1$ ne sont pas éliminés.

La propriété **iii** est donc vérifiée.

Montrons maintenant que pour tout pattern $P \in F_1^{i+1}$, on peut trouver un pattern $Q \in \mathcal{F}$ dont le plus gros générateur est de profondeur $m+1$, et dont P est l'applatissement d'un sous-arbre dont la racine a une profondeur supérieure ou égale à $m+1$ dans Q.

Par construction, si $P \in F_1^{i+1}$ alors $P \in F_1^i$. Par hypothèse de récurrence, il existe donc $Q \in \mathcal{F}$ dont le plus gros générateur PR est de profondeur m, et dont P est l'applatissement d'un sous-arbre dont la racine a une profondeur supérieure ou égale à m dans Q.

Si PR n'est pas racine, le plus gros générateur de Q ne change pas d'une itération sur l'autre. En effet, on ne peut pas avoir $PR \in F_{\mathcal{F}}^i$, car dans ce cas on aurait $PR = Q$, et P serait réduit à un nœud, ce qui ne nous intéresse pas. Donc $PR \in F_1^i$. Comme il n'est pas racine, il n'y a aucun accrochages d'effectués, Q ne change donc pas de plus gros générateur. Donc P est toujours l'applatissement d'un sous-arbre de Q dont la racine est d'une profondeur supérieure ou égale à m la profondeur du plus gros générateur de Q.

Si PR est racine, alors nous avons vu dans les démonstrations des propriétés **i** et **ii** qu'à l'itération $i+1$, on construit à partir de PR un pattern $P' = gen_{i+1}(Q)$, P' de profondeur $m+1$. Si P était uniquement l'applatissement d'un sous-arbre dont la racine est de profondeur m de Q, alors P aurait été utilisé pour construire P', et il aurait été éliminé car ses occurrences auraient pu être déduites de celles de P'. Sinon, il serait devenu racine (voir démonstration du lemme 2). Puisque P n'a pas été éliminé et qu'il n'est pas racine, P est donc l'applatissement d'un sous-arbre de profondeur supérieure ou égale à $m+1$ de Q.

La propriété **iv** est donc vérifiée.

La propriété générale est donc vraie par récurrence. □

Lemme 5 *Soit $Q \in \mathcal{F}$. S'il existe une itération i de* Dryade *pour laquelle $gen_i(Q)$ est un pattern racine, alors il existe une itération de* Dryade *$i' > i$ où Q est découvert par l'algorithme, c'est-à-dire $Q \in F_{\mathcal{F}}^{i'}$.*

Preuve : Soit $P = gen_i(Q)$, de profondeur m, P est donc $m-1$-générateur pour Q. Montrons que $i' = i + profondeur(Q) - m$.

Nous savons que P est tel que Q et P ont le même troncage à la profondeur m. Nous avons vu dans la démonstration du lemme précédent qu'à l'itération $i+1$, $EPatt(P)$ contiendra un pattern P'

qui sera m-générateur pour Q. En appliquant ce procédé récursivement, il faut $profondeur(Q) - m$ itérations à partir de l'itération i pour découvrir le pattern Q. □

Lemme 6 *Soit $P \in \mathcal{F}_1 = \mathcal{F}_D^1$. S'il existe $Q \in \mathcal{F}$ tel que $P = gen_1(Q)$, alors il existe une itération i de* Dryade *à laquelle le pattern P devient racine, c'est-à-dire $P \in F_R^i$. Sinon, il existe une itération i' à laquelle P est éliminé de $\mathcal{F}_D^{i'}$.*

Preuve : Nous avons vu dans la preuve du lemme 2 que pour tout pattern de $\mathcal{F}_D^1$, après plusieurs itérations de Dryade, soit il devenait un pattern racine, soit il était éliminé.

De plus, nous avons vu das la preuve du lemme 4 (cas général, propriétés **i** et **ii**), que tout pattern de F_1^i éliminé n'était pas un plus gros générateur pour un pattern de $\mathcal{F}$.

Les patterns qui deviennent racines sont donc bien ceux qui sont les plus gros générateurs d'un pattern de $\mathcal{F}$. □

THÉORÈME 2 *L'algorithme* Dryade *est correct et complet.*

Preuve :

Montrer que Dryade est correct et complet revient à montrer qu'il existe une itération k pour laquelle $F_D^k = \mathcal{F}$.

Complétude

Montrons qu'il existe un itération k pour laquelle $F_{\mathcal{F}}^k = \mathcal{F}$.

Soit $Q \in \mathcal{F}$, considérons $P \in \mathcal{F}_D^1$ son plus gros générateur.

– si $P \in F_{\mathcal{F}}^1$, alors Q est déjà découvert par Dryade.

– si $P \in F_R^1$, alors d'après le lemme 5, il existe une itération à laquelle Q sera découvert par Dryade.

– si $P \in F_1^1$, alors d'après le lemme 6, il existe une itération à laquelle P va devenir racine, et donc d'après le cas précédent Q sera découvert par Dryade à une itération ultérieure.

Donc pour tout $Q \in \mathcal{F}$ il existe une itération k_Q où Q est découvert par Dryade, on en déduit la complétude.

Correction

Considérons l'itération k où Dryade termine, on a $F_{\mathcal{F}}^k = \mathcal{F}$. Montrons que $F_R^k = F_1^k = \emptyset$.

Comme Dryade a terminé il ne reste plus d'accrochages dans le graphe d'après le lemme 2. Donc s'il reste des patterns, ce sont soit des patterns de $F_{\mathcal{F}}^k$, soit des patterns de F_R^k : $F_1^k = \emptyset$.

Or d'après la propriété **ii** du lemme 4, s'il reste un pattern racine il est générateur pour un pattern de $\mathcal{F}$. A l'itération k cela n'est pas possible, donc $F_R^k = \emptyset$. D'où la correction.

□

Chapitre VI

Étude expérimentale

L'objet de ce chapitre est d'étudier expérimentalement les performances de l'algorithme Dryade. Pour cette étude, nous avons réalisé une implémentation de Dryade. Nous commencerons par exposer les caractéristiques de cette implémentation, puis nous détaillerons le protocole expérimental. Ensuite nous donnerons les résultats des expériences et nous les commenterons.

1 Implémentation

À partir de l'algorithme Dryade exposé dans la deuxième partie du chapitre V, nous avons réalisé une implémentation en C++. Cette implémentation totalise 15000 lignes de code, et utilise intensivement les structures de données fournies par la *Standard Templates Library (STL)*.

Lors de la réalisation de notre implémentation, nous avons dû, le plus souvent pour des impératifs de temps, effectuer des choix qui nous écartent quelque peu de l'algorithme présenté dans le chapitre V. Nous allons expliquer ici ces choix et les restrictions associées.

1.1 Eclat au lieu de Charm

Nous avons vu dans le chapitre V que pour réaliser Dryade, nous avions besoin d'utiliser un algorithme vertical de découverte d'itemsets fréquents fermés, comme Charm. M.Zaki a aimablement mis à notre disposition son implémentation de l'algorithme Charm. Pour une efficacité maximale, l'idéal était de pouvoir intégrer directement le code de M. Zaki dans notre programme. Toutefois, le code de Charm est très complexe et s'appuie sur plusieurs programmes préparant les données en amont. Compte tenu des délais, nous n'avons pas pu analyser tous ces programmes et les regrouper dans une classe C++ utilisable dans notre implémentation.

Nous avons donc utilisé comme algorithme vertical de découverte d'itemsets fréquents fermés l'algorithme Eclat, dont l'implémentation en C nous a été fournie par C. Borgelt, que nous remercions ici. Il nous a été très facile d'intégrer le code d'Eclat dans une classe C++ utilisable par notre implémentation. Eclat découvrant **tous** les itemsets fréquents, nous avons ajouté à sa suite un filtre qui ne renvoie que les itemsets fréquents fermés.

Ce choix a pour seule conséquence d'augmenter les temps de notre programme. Nous ne mesurons pas le temps de passage dans le filtre, mais utiliser Eclat à la place de Charm nous fait perdre un ordre de grandeur en vitesse d'après les études de M. Zaki ([Zaki et al. 1997], [Zaki et Gouda 2003]).

Les temps que nous donnerons pour notre algorithme distingueront donc le temps relatif à la recherche d'itemsets fréquents fermés, pour lequel ECLAT fournit une borne supérieure, du temps propre à DRYADE.

1.2 Pas de stockage des mappings

Notre implémentation de DRYADE est fondée sur une implémentation antérieure de DRYAL, qui nous a permis de tester la faisabilité de notre approche.

Les structures de données utilisées dans l'implémentation de DRYAL ne conservent pas, pour un pattern fréquent, tous ses mappings dans les données. Cela n'est pas nécessaire dans le cas de DRYAL : il suffit de garder les éléments du mapping relatifs à la racine et aux feuilles du pattern pour réaliser les accrochages.

Pour accélérer le développement de notre implémentation de DRYADE, nous avons gardé les mêmes structures de données que dans notre implémentation de DRYAL, c'est-à-dire qu'un mapping d'un pattern n'est représenté que par l'image de la racine et l'image des feuilles.

Cela se traduit par deux limitations de notre code par rapport à l'algorithme DRYADE décrit dans le chapitre V :

1. **Test de fermeture** Puisque nous ne connaissons pas précisément les mappings, le test qui garantit la fermeture à la fin de la procédure $MiseAJour$ (algorithme 29) ne peut pas être effectué. Concrètement, cela signifie que des patterns redondants risquent d'être générés. L'influence de cette redondance sur le temps d'exécution de notre algorithme peut être discutée de la façon suivante. D'un côté, le test qui garantit la fermeture à la fin de la procédure $MiseAJour$ n'est pas effectué, ce qui fait gagner du temps. D'un autre côté, les patterns redondants éventuellement générés devront être traités dans les itérations suivantes. Il est clair que le surcoût de la redondance dépasse le coût du test de fermeture lorsque la taille et la profondeur des solutions augmentent, ceci correspondant à une augmentation exponentielle du nombre de patterns redondants.

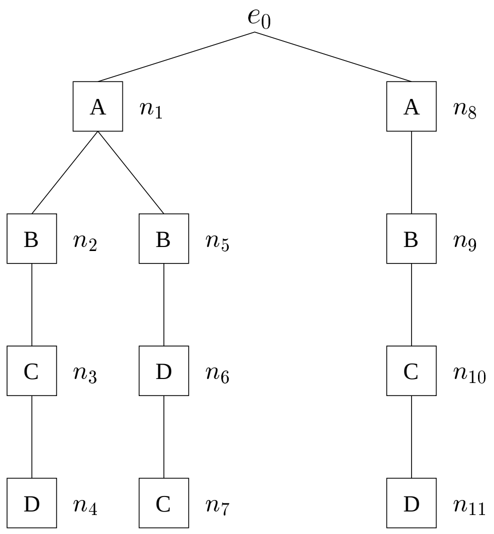


FIG. VI.1 – D_Z

Bien que le test de fermeture à la fin de la procédure $MiseAJour$ ne soit pas réalisé, la plupart des patterns trouvés par DRYADE sont bien des patterns fermés, car les résultats

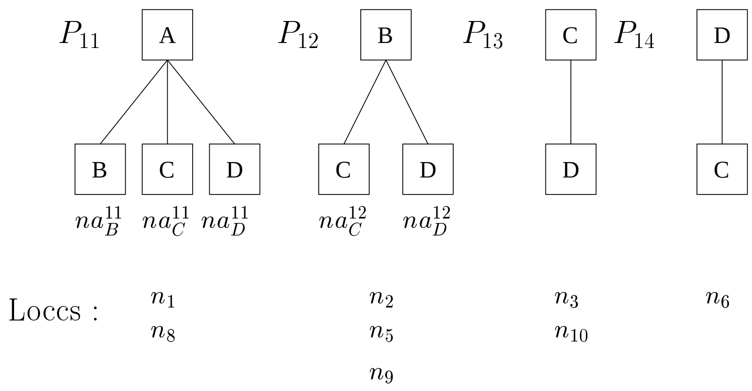


FIG. VI.2 – Patterns fréquents fermés de profondeur 1 de D_Z

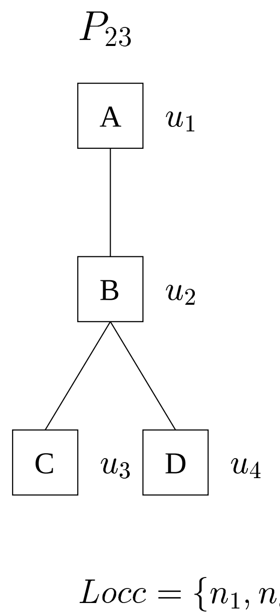


FIG. VI.3 – Pattern P_{23} obtenu par accrochage et suppression des feuilles redondantes des algorithmes de découverte d'itemsets fréquents sont eux fermés. Ceci est garanti par un traitement des itemsets fréquents que nous fournit ECLAT.

2. **Forme des arbres** Dans certains cas particuliers, le fait d'avoir une structure de données qui ne représente pas l'intégralité des mappings ne permet pas de garantir un traitement correct des accrochages. Ces cas correspondent aux arbres qui ont au moins deux nœuds u et v de même étiquette, qui ne descendent pas l'un de l'autre, et tels qu'il existe des descendants de u et de v qui ont la même étiquette.

   La figure VI.1 montre une base de données d'arbres D_Z contenant un arbre de ce type, il s'agit de l'arbre dont le nœud racine est n_1. En effet, les nœuds n_2 et n_5 de cet arbre sont tous deux étiquetés par B, et ont tous deux des descendants étiquetés par C et D.

   Le problème survient lors de l'accrochage et de la suppression des feuilles de P_{12} sur P_{11} (figure VI.2), qui donne le pattern P_{23} (figure VI.3). Il y a trois mappings entre les nœuds de P_{23} et les nœuds de D_Z, représentés dans la table ci-dessous :

Nœud de P_{23}	Mapping 1	Mapping 2	Mapping 3
u_1	n_1	n_1	n_8
u_2	n_2	n_5	n_9
u_3	n_3	n_7	n_{10}
u_4	n_4	n_6	n_{11}

   Dans le code, les mappings 1 et 2 ne peuvent pas être distingués : ils correspondent à une unique structure de données qui associe à u_1 le nœud n_1, à u_3 les nœuds n_3 et n_7, et à u_4 les nœuds n_4 et n_6. Cette représentation suppose de façon implicite que dans D_Z les nœuds n_2 et n_5 soient confondus, ce qui n'est pas le cas. Cela conduira à des accrochages incorrects : par exemple, l'accrochage simultané de P_{13} et P_{14} sur P_{23}.

   Nos expérimentations ne considèrent donc que des cas où un traitement correct des accrochages est garanti. Cela signifie que dans nos expérimentations, les arbres d'entrée vérifient toujours la contrainte suivante, que nous appelerons **CNTR**.

   Contrainte : Un arbre vérifie la contrainte **CNTR** si pour tout couple de nœuds u et v de même étiquette, soit u est ancêtre de v, soit aucun descendant de u n'a la même étiquette qu'un descendant de v.

Ainsi notre implémentation s'écarte des algorithmes du chapitre V en deux points : d'une part, le test de mise à jour est incomplet, ce qui pénalise le temps de calcul lorsque le problème admet des solutions de grande taille. D'autre part, nous supposons que les arbres considérés vérifient la contrainte **CNTR**.

2 Protocole expérimental

Le but de nos expérimentations est d'étudier les performances et le comportement de DRYADE sur des données arborescentes. Nous comparerons également DRYADE au seul algorithme qui actuellement puisse trouver les mêmes résultats que lui, à savoir WARMR. Bien sûr cette comparaison n'est pas très juste, car WARMR est un algorithme qui peut trouver des solutions à des problèmes beaucoup plus généraux que ceux de DRYADE, et il n'utilise pas la propriété de fermeture. Toutefois, c'est le seul algorithme auquel nous puissions nous comparer, nous montrerons donc le gain de performance que notre approche spécifique peut apporter par rapport à une approche générale.

À l'heure actuelle et à notre connaissance, les corpus de données arborescentes en libre accès ne sont pas nombreux. De plus, dans le cas de documents XML, nous sommes souvent confronté à un problème de synonymie au niveau des balises, problème dont nous avons déjà parlé dans l'introduction et qui se situe en amont de Dryade.

La validation expérimentale s'appuiera ainsi, d'une part sur un ensemble de problèmes aléatoires (sections 3.1 et 3.2), et d'autre part sur une application réelle (section 3.3).

Le cadre considéré pour la validation de notre approche sur des problèmes artificiels est tout d'abord détaillé.

Les données artificielles sont générées selon une distribution uniforme, pour laquelle nous avons défini 4 paramètres d'ordre :

- N : le nombre d'arbres dans le corpus
- E : L'ensemble d'étiquettes utilisables (sans perte de généralité, les étiquettes sont des entiers de 0 à $|E|-1$)
- B : le facteur de branchement moyen pour un arbre du corpus
- P : la profondeur moyenne des feuilles pour un arbre du corpus

La génération du corpus est assurée par la procédure *GenereCorpus*, détaillée dans l'algorithme 30.

Algorithme 30 *GenereCorpus*

Entrée : E l'ensemble des étiquettes, B le facteur de branchement moyen, P la profondeur moyenne, N le nombre d'arbres

Sortie : Un corpus de N arbres à étiquettes dans E satisfaisant tous la contrainte **CNTR**, tel que chaque arbre a un facteur de branchement moyen de B et une profondeur moyenne de P.

$Corpus \leftarrow \emptyset$
Pour $i = 1$ à N **Faire**
  $nbEssais \leftarrow 0$
  Répéter
    Répéter
      $arbre \leftarrow GenereArbre(P, E, 2B, P)$ // *voir algorithme 31*
    Jusqu'à ce que $profondeurMoyenne(arbre) = P$ et $branchementMoyen(arbre) = B$
    $nbEssais \leftarrow nbEssais + 1$
  Jusqu'à ce que ($arbre$ vérifie **CNTR**) ou $nbEssais > MAX\_ESSAIS$
  Si $nbEssais > MAX\_ESSAIS$ **Alors**
    Quitter en renvoyant une erreur : la génération de ce corpus est trop contrainte.
  Sinon
    $Corpus \leftarrow Corpus \cup \{arbre\}$
  Fin Si
Fin Pour
Retourner $Corpus$

Le principe est de générer des arbres aléatoires grâce à la procédure *GenereArbre* (algorithme 31), puis de ne retenir dans le corpus que ceux qui d'une part vérifient les contraintes de branchement moyen et de profondeur moyenne désirées, et d'autre part satisfont la contrainte **CNTR**. Cette méthode est coûteuse en temps de calcul, mais elle a l'avantage d'éviter d'introduire un biais important lors de la génération de chaque arbre, comme on peut le voir en examinant la procédure *GenereArbre*.

Le seul biais par rapport à une génération uniforme (ligne 1 de l'algorithme 31) est motivé par le désir d'obtenir des solutions non-triviales, c'est-à-dire de profondeur strictement supérieure à 1, pour de faibles valeurs de P. Les essais préliminaires montrent en effet que pour P petit (inférieur

Algorithme 31 *GenereArbre*

Entrée : p la profondeur courante, E l'ensemble des étiquettes, $bmax$ le facteur de branchement maximal, $pmax$ la profondeur maximale

Sortie : Un arbre aléatoire à étiquettes dans E de profondeur maximale $pmax$, de facteur de branchement moyen maximal $bmax$

$e \leftarrow$ tirage aléatoire d'une étiquette dans $E[p]$
$n \leftarrow$ nouveau nœud d'étiquette e
Si $p > 0$ **Alors**
  $b \leftarrow$ tirage aléatoire d'un facteur de branchement entre 0 et $bmax$
  Si $(b = 0)$ et $(p = pmax)$ **Alors**
    $b \leftarrow 1$ // *Évite d'avoir des arbres réduits à un seul nœud*
  Fin Si
  Pour $i = 0$ à b **Faire**
    $n_i \leftarrow GenereArbre(p-1, E, bmax, pmax)$
    Ajouter n_i comme fils à n
  Fin Pour
Fin Si
Return n

à 3 ou 4), il y a peu de patterns fréquents de profondeur supérieure à 1 dans les données. Mais augmenter P est extrêmement coûteux, car cela fait augmenter exponentiellement le nombre de relations d'ancestralité à considérer. Le biais introduit pour remédier à ce défaut considère une partition de E en $P+1$ sous-ensembles disjoints de même taille $E[0], ..., E[P]$. Dans le cas où E n'est pas divisible par $P+1$, tous les ensembles recoivent $round(|E|/(P+1))$ étiquettes, sauf $E[0]$ qui reçoit les étiquettes restantes.

Pour chaque niveau de profondeur p les étiquettes des nœuds de ce niveau sont tirées de manière uniforme dans le sous-ensemble $E[p]$. Le biais que nous introduisons permet ainsi de trouver des patterns fréquents de profondeur supérieure ou égale à 2 dès que P vaut 3 ou 4.

La majeure partie des expérimentations a été réalisée sur des Pentium IV 2.8 GHz avec 512 Mo de mémoire vive, sous Linux. Certaines expériences ont été réalisées avec un AMD Athlon 1 GHz avec 1 Go de mémoire vive, nous préciserons lorsque ce sera le cas.

3 Résultats

Dans une première section, nous allons comparer Warmr et Dryade. Puis nous étudierons la complexité effective de Dryade dans une deuxième section, avant de terminer par une étude sur des données réelles.

3.1 Comparaison avec Warmr

Pour comparer Dryade avec Warmr, nous traduisons chaque corpus d'arbres générés aléatoirement dans un format acceptable par Warmr. Pour cela, chaque arbre est remplacé par son encodage relationnel. Nous utilisons l'encodage relationnel Rel^+ donné dans la définition 26 du chapitre IV. Warmr a besoin de connaître le nombre maximal d'atomes des solutions qu'il recherche : nous avons établi cette limite au nombre d'arêtes père-fils moyen d'un arbre du corpus.

Nous avons produit trois corpus pour le test :

Corpus 1	1000 arbres, $P = 3$, $B = 1$, $\|E\| \in [4..56]$
Corpus 2	1000 arbres, $P = 3$, $B = 2$, $\|E\| \in [16..56]$
Corpus 3	1000 arbres, $P = 3$, $B = 3$, $\|E\| \in [24..56]$

Notons que les valeurs de $|E|$ ne commencent pas à un, car quand il y a trop peu d'étiquettes il est impossible de générer des arbres vérifiant la contrainte **CNTR**.

Le seuil de fréquence absolu est de 15. Pour chaque corpus caractérisé par P, B et un nombre d'étiquettes $|E|$ choisi, nous effectuons 8 tirages aléatoires différents. Le temps d'exécution des programmes WARMR et DRYADE est limité à une heure pour chaque exécution.

Sur les corpus 2 et 3, nous ne pouvons présenter aucun résultat pour WARMR, car suivant le nombre d'étiquettes soit WARMR dépasse le temps imparti à son exécution, soit il consomme toute la mémoire et se fait arrêter par le système. Dans les mêmes conditions DRYADE a toujours terminé son exécution correctement.

Les résultats pour le corpus 1 sont présentés sur la figure VI.4.

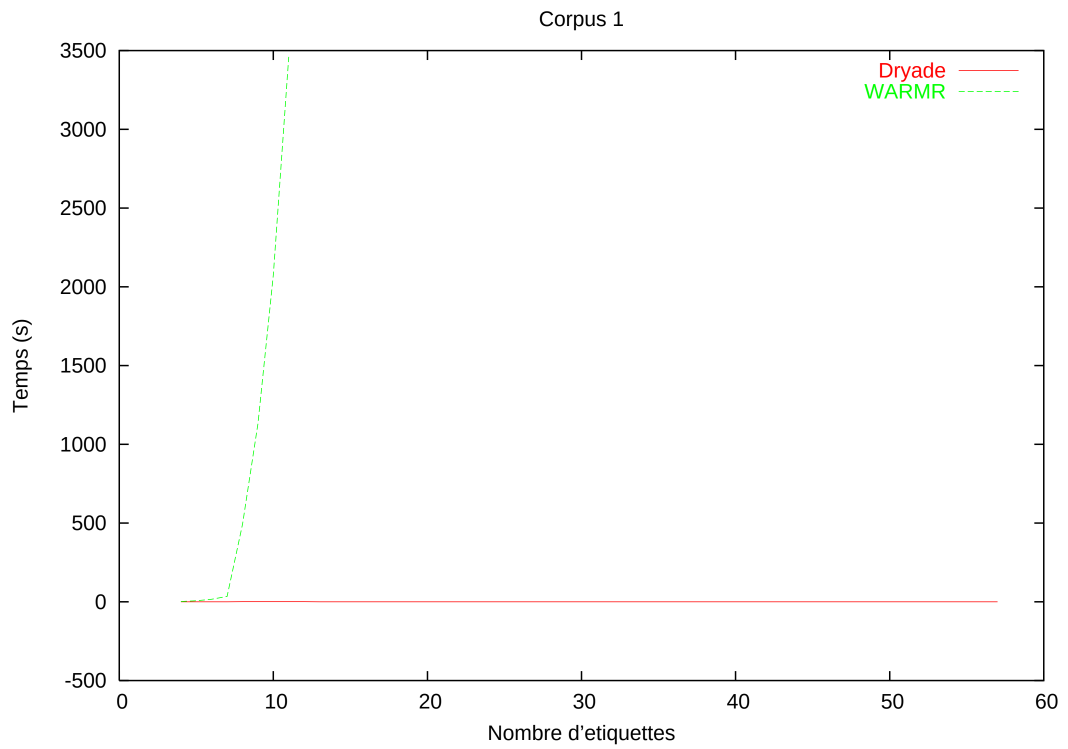


FIG. VI.4 – Différence de temps d'exécution entre WARMR et DRYADE

Nous pouvons constater que le temps d'exécution de DRYADE est inférieur à celui de WARMR de plusieurs ordres de grandeur. En effet, le temps maximal mis par DRYADE pour trouver les patterns fréquents est de 1.14s, lorsqu'il y a 8 étiquettes dans E. Dans les mêmes conditions, WARMR a besoin de 493.19s pour trouver les arbres fréquents. Dès que le nombre d'étiquettes dans E atteint 12, WARMR n'arrive pas à répondre dans le temps et la quantité de mémoire impartis.

Comme nous l'avons vu plus tôt, cette comparaison n'est pas très juste. En effet, alors que DRYADE se contente de chercher les patterns fréquents fermés, WARMR cherche **tous** les patterns fréquents. Cela lui demande donc beaucoup plus de calcul, et on peut une nouvelle fois constater l'efficacité d'une approche utilisant la propriété de fermeture.

De plus, même des patterns fréquents simples sont représentés par des formules logiques qui peuvent être assez longues. Considérons l'arbre A de la figure VI.5.

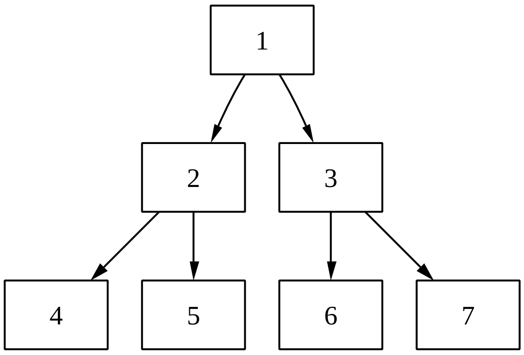


Fig. VI.5 – Arbre binaire complet de profondeur 2

Considérons maintenant un corpus très simple ne contant que cet arbre, et un seuil de fréquence absolu de 1. L'unique arbre fréquent à trouver est donc A lui-même. Sur une machine cadencée à 1 GHz, Dryade trouve A en moins d'un centième de secondes. Sur la même machine, Warmr a besoin de 1508.6 secondes pour retrouver A.

Non seulement Warmr est handicapé par le fait qu'il ne cherche pas les patterns fréquents fermés, mais de plus la formule qui encode les relations de parenté de A est longue de 6 atomes (c'est le nombre d'arêtes de A). Retrouver des formules fréquentes contenant de nombreux atomes est difficile pour Warmr, comme [Inokuchi et al. 2000] l'a montré. Dans le cas particulier de données structurées sous forme de graphe considéré dans cet article (molécules chimiques), l'approche spécifique aux graphes développée permet de trouver des motifs plus complexes que ceux trouvés par Warmr.

De même, notre approche spécifiquement développée pour prendre en compte des données arborescentes, s'avère significativement plus rapide que Warmr sur ces données.

3.2 Étude de performance

3.2.1 Passage à l'échelle

Nous allons d'abord étudier le comportement de Dryade en fonction du nombre d'arbres donnés en entrée. Nous considérons une instance du corpus 2, c'est-à-dire des arbres de profondeur moyenne 3 et de facteur de branchement moyen 2. Le nombre d'étiquettes sera fixé à 20, car c'est un cas où Dryade doit trouver des solutions de profondeur supérieure à 1. Le seuil de fréquence relatif est de 15/1000. Nous avons fait varier le nombre d'arbres entre 1000 et 10000. La figure VI.6 présente les temps d'exécution de chaque étape de Dryade (courbes logarithmiques).

Au delà de 8000 arbres, Dryade n'a pas pu terminer son exécution à cause d'une saturation de la mémoire.

La courbe la plus basse correspond au temps de découverte des patterns fréquents fermés de profondeur 1 par Eclat. Cette courbe en échelle logarithmique est clairement sublinéaire, ce qui était attendu. En effet il est connu qu'Eclat est linéaire en fonction du nombre de transactions, c'est-à-dire du nombre de nœuds dans les données. Or ici seul le nombre d'arbre change, ce sont des arbres ayant le même nombre de nœuds en moyenne, et le seuil de fréquence est relatif. Donc Eclat est bien linéaire en fonction du nombre d'arbres d'entrée.

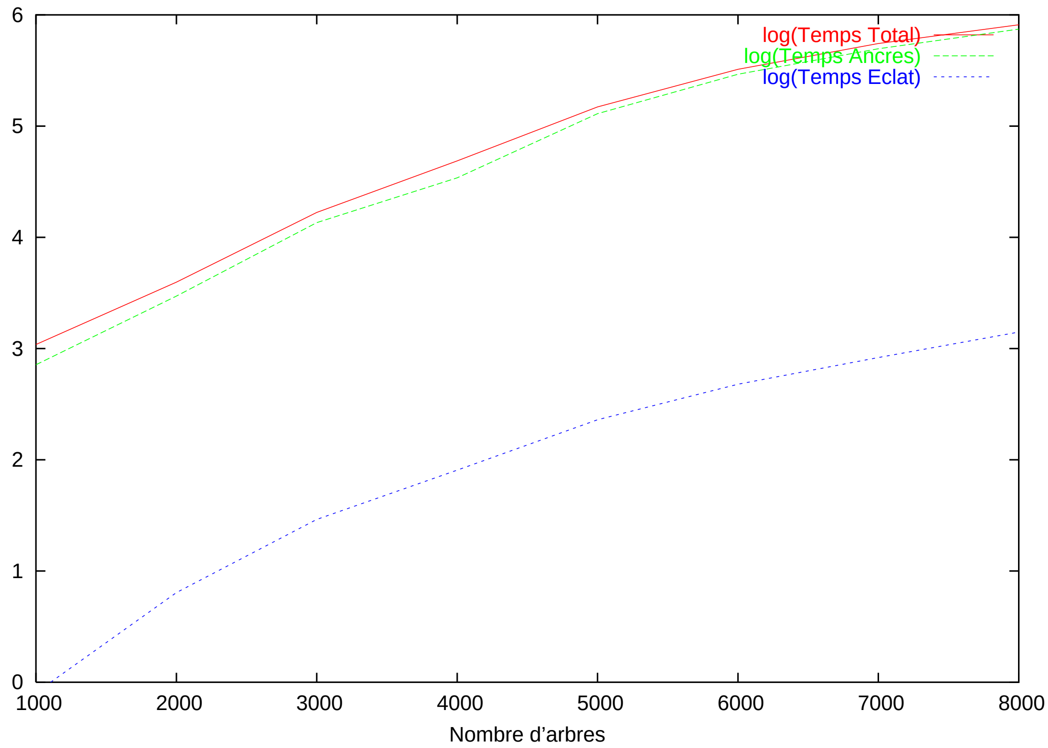


Fig. VI.6 – Expériences de passage à l'échelle

La courbe suivante correspond au temps après la découverte des ancres d'accrochage entre les patterns fréquents de profondeur 1, son aspect logarithmique est sublinéaire aussi. Cette phase est donc polynômiale en fonction du nombre d'arbres donnés en entrée.

Enfin, la dernière courbe représente le temps total de l'algorithme, son aspect logarithmique sublinéaire nous permet d'affirmer que Dryade est en moyenne polynômial en fonction du nombre de données d'entrée.

Ce résultat nous permet d'espérer qu'avec une gestion plus efficace de la mémoire (représentations condensées [Jeudy et Boulicaut 2002] ou compressées par exemple), Dryade serait capable de traiter des quantités de données encore plus volumineuses.

3.2.2 Analyse détaillée des 3 phases de Dryade

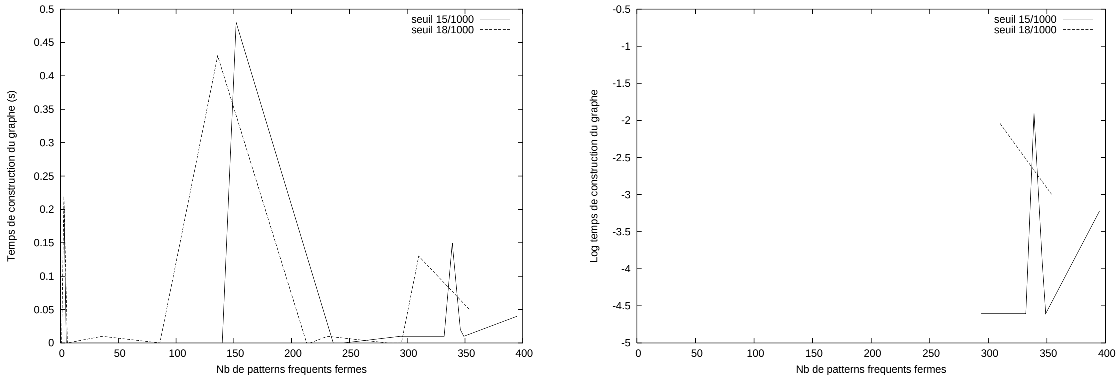
L'algorithme Dryade procède en trois phases : tout d'abord, il calcule les patterns fréquents fermés de profondeur 1 (avec Eclat dans notre code), ensuite il calcule le graphe des accrochages entre ces patterns, puis il effectue les accrochages.

Calcul des patterns fréquents fermés de profondeur 1 :

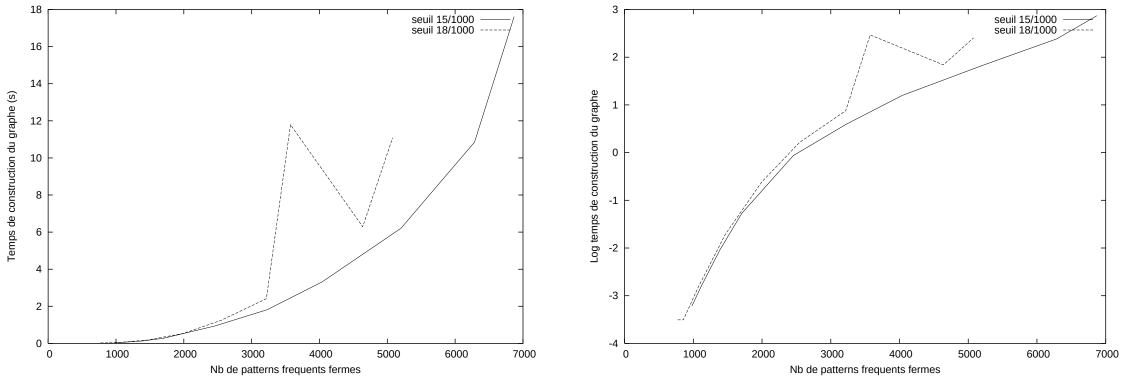
La première phase reposant sur des algorithmes bien connus, nous n'étudierons pas sa complexité. Une étude de la complexité des algorithmes de type Apriori se trouve dans le chapitre III.

Calcul du graphe des accrochages :

Le temps de calcul du graphe des accrochages est en relation avec le nombre de patterns fréquents fermés de profondeur 1 découverts à la première étape. Pour mettre en évidence cette relation, nous donnons dans les tables VI.1, VI.2 et VI.3 les courbes représentant en abscisse le nombre de patterns fréquents de profondeur 1 découverts, et en ordonnée le temps nécessaire à la construction du graphe



TAB. VI.1 – Corpus 1



TAB. VI.2 – Corpus 2

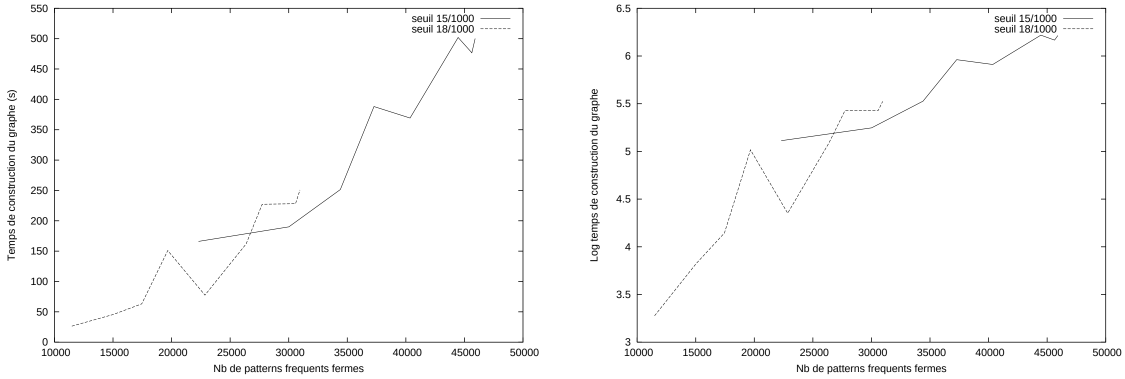
d'accrochage (courbe de gauche), et le logarithme de ce temps (courbe de droite).

Les courbes logarithmes des corpus 2 et 3 (facteurs de branchements moyens de 2 et 3 respectivement) sont sublinéaires. Nous pouvons en déduire que le temps pour le calcul du graphe des accrochages est dans ces cas polynômial en fonction du nombre de patterns de profondeur 1. En observant la procédure *CalculeGraphe* (algorithme 20 page 97), ce comportement s'explique facilement : le premier test, quadratique en fonction du nombre de patterns fréquents de profondeur 1, teste la possibilité d'un accrochage, et ce test échoue dans la grande majorité des cas.

Toutefois, la courbe correspondant au corpus 1 (facteur de branchement moyen de 1) exhibe un comportement différent, que nous allons analyser.

La courbe VI.7 montre l'évolution du nombre de patterns fréquents de profondeur 1 en fonction du nombre d'étiquettes. Une modélisation analytique est proposée en annexe à la fin de ce chapitre, que nous allons résumer ici. Le point clé est que le nombre de patterns fréquents de profondeur 1 présente, en fonction du nombre d'étiquettes, deux pics. L'un correspond à une majorité de patterns "longs" (plus de deux fils), l'autre à une majorité de patterns "courts" (1 seul fils). Pour mieux comprendre l'évolution du temps de calcul du graphe des accrochages dans ce cas, nous proposons une autre représentation dans le paragraphe suivant.

Nous formulons l'hypothèse suivante : dans le cas d'un facteur de branchement moyen égal à 1, il y a beaucoup de structures fréquentes de profondeur supérieure à 1, surtout quand il y a peu d'étiquettes dans les arbres du corpus. Par conséquent, il y a beaucoup de calculs d'ancres à effectuer dans *CalculeGraphe*, ce qui n'était pas le cas pour des corpus avec des facteurs de branchement



TAB. VI.3 – Corpus 3

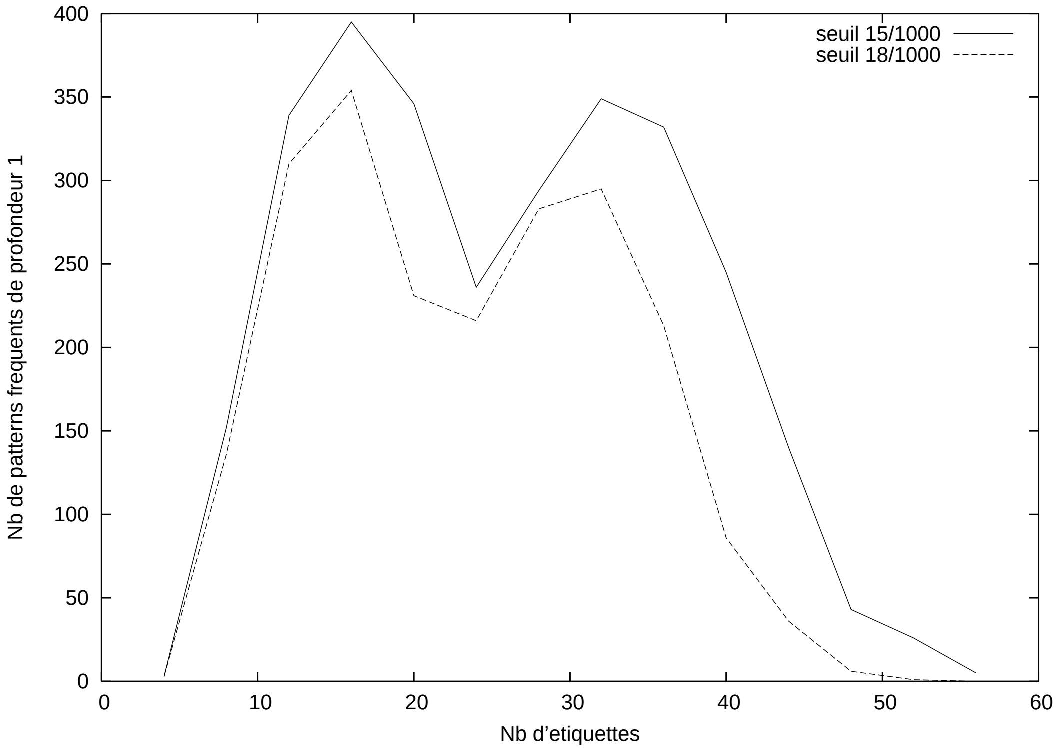


FIG. VI.7 – Corpus 1

plus élevés. Pour vérifier cette hypothèse, nous allons tracer une courbe avec en abscisse le rapport entre le nombre de structures fréquentes découvertes en tout par Dryade, et le nombre de patterns fréquents de profondeur 1 de départ. Plus il y a de mappings, plus il y aura d'accrochages, et plus ce rapport sera élevé. On s'attend à ce que le temps consommé pour la découverte du graphe des accrochages augmente avec ce rapport. Comme le montre la figure VI.8, c'est effectivement ce qui se produit : dès que l'on sort des cas où tous les patterns fréquents fermés finaux sont des patterns de profondeur 1, le temps de construction du graphe des accrochages croit linéairement avec le rapport du nombre de patterns fréquents trouvés par le nombre de patterns fréquents de profondeur 1.

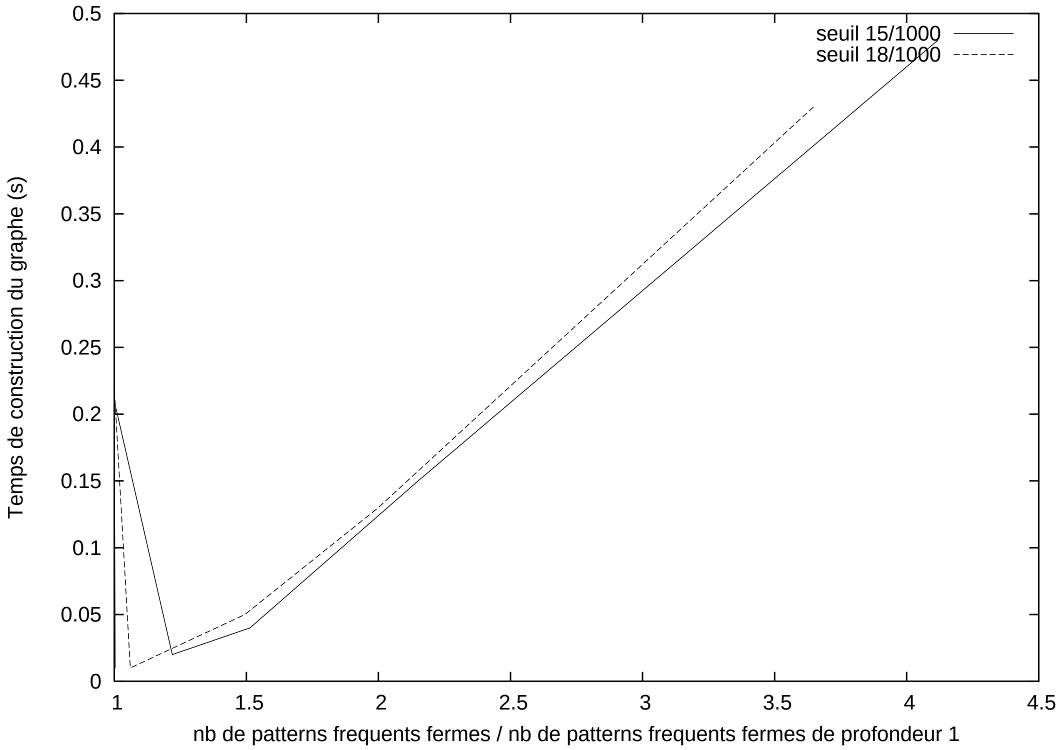


Fig. VI.8 – Corpus 1

Calcul des accrochages :

Le nombre d'itérations d'accrochage dépendant de la profondeur des patterns fréquents fermés à trouver, nous avons tracé sur les courbes VI.9, VI.10 et VI.11 le temps nécessaire aux accrochages en fonction de la profondeur moyenne des patterns fréquents présents dans le corpus, pour les seuils de fréquence absolus 15 et 18. Notons que dans la plupart des courbes, les derniers points pour le seuil 15 n'ont pas pu être obtenus : comme toujours, pour un seuil faible le problème devient plus difficile, et le résoudre peut dépasser le temps imparti.

Toutes ces courbes ont un aspect logarithmique sublinéaire, nous pouvons en déduire que la complexité de la phase d'accrochage de Dryade est polynômiale en fonction de la profondeur moyenne des patterns fréquents fermés à découvrir.

Notons que le décrochement qui apparaît pour la profondeur maximale de la courbe de la figure VI.9 est dû au fait que les solutions profondes ne sont obtenues qu'avec un faible nombre d'étiquettes. Ainsi, le point extrême de la figure VI.9 est obtenu pour $|E| = 4$. Avec le biais de génération que nous imposons, les 1000 arbres sont identiques, donc il y a un seul pattern fréquent à découvrir, ce qui facilite le travail de Dryade.

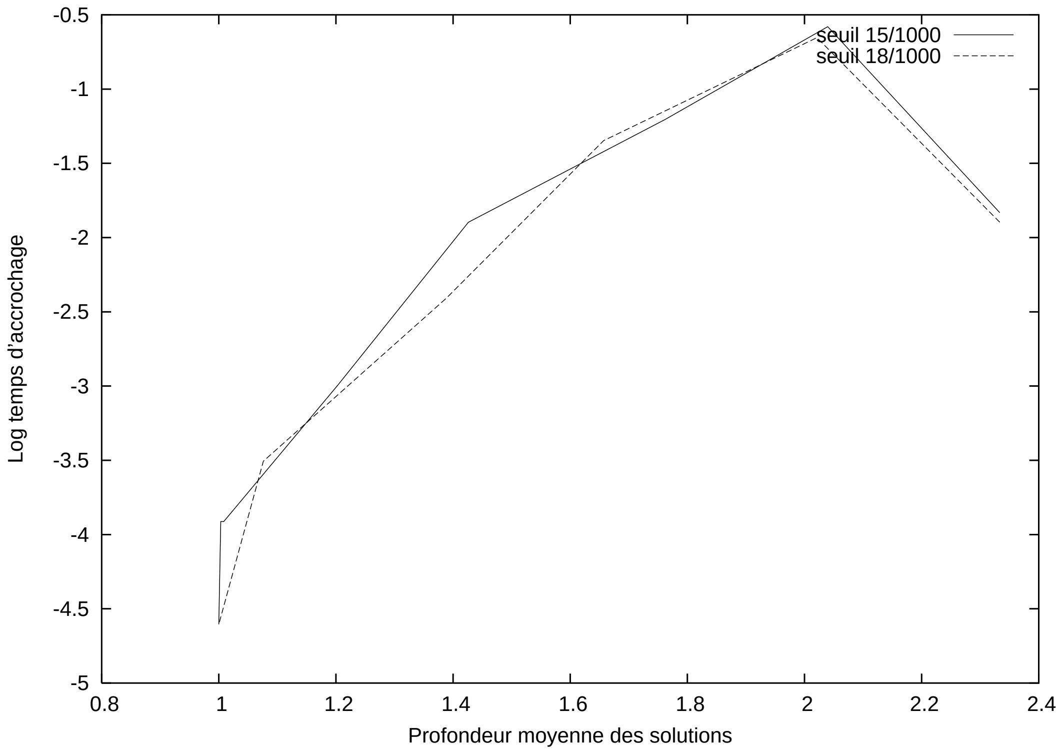


Fig. VI.9 – Corpus 1

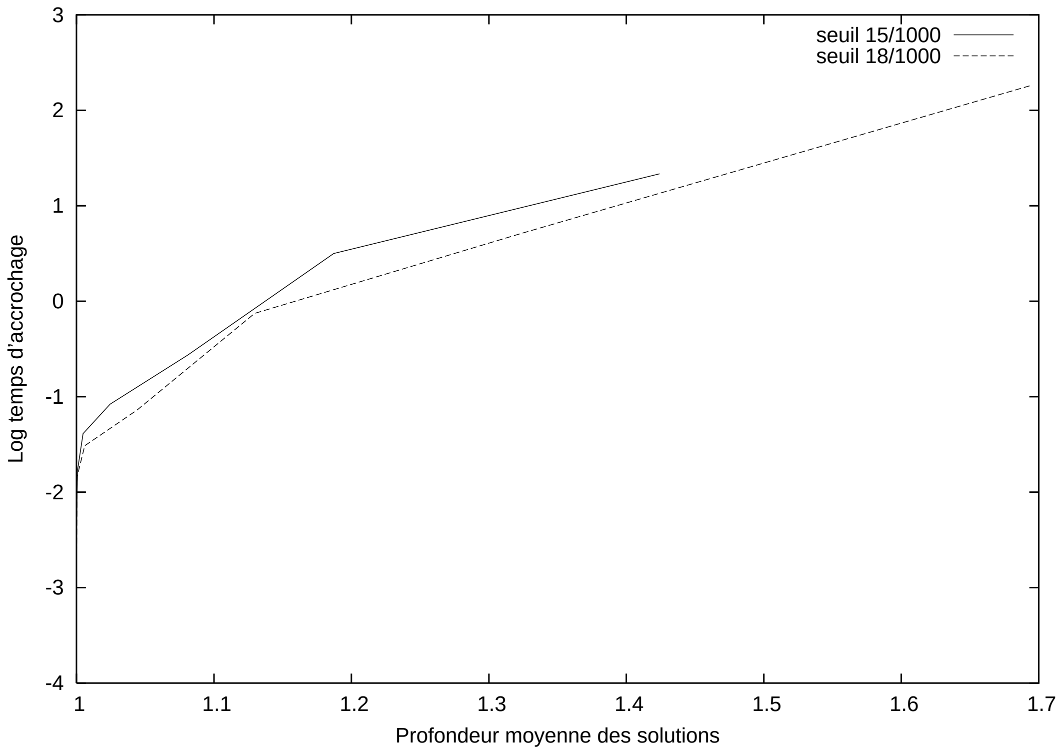


Fig. VI.10 – Corpus 2

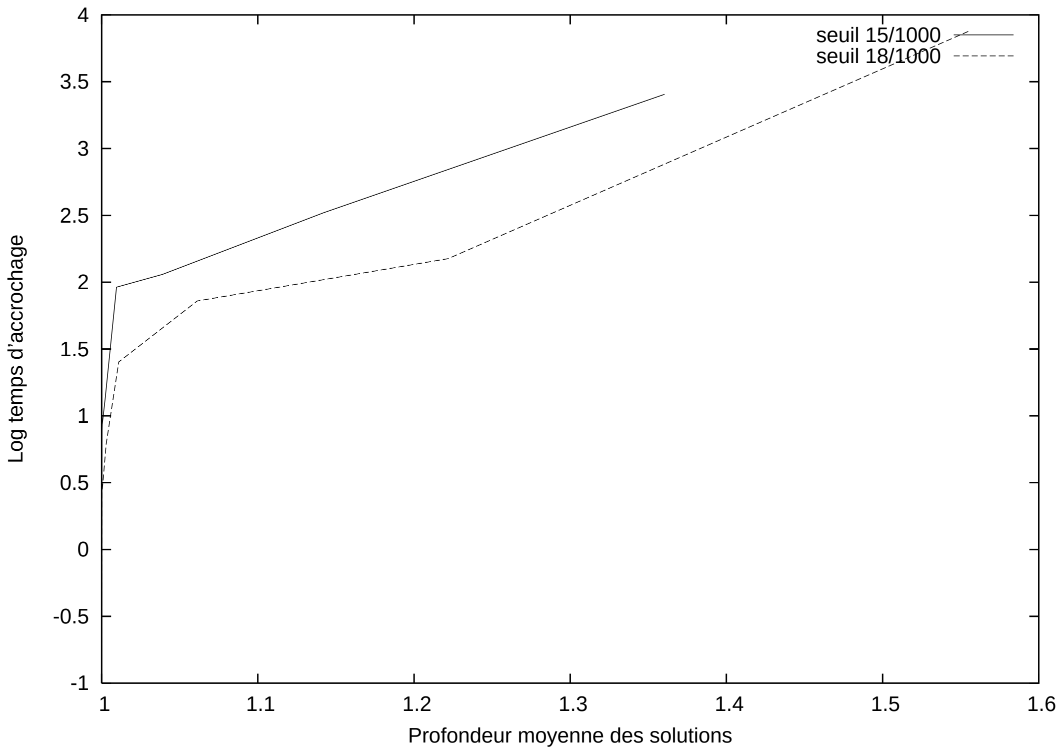


Fig. VI.11 – Corpus 3

3.3 Étude sur des données réelles

La société Xylème a gracieusement mis à notre disposition un corpus de documents XML réels. Ce corpus comprend 3667 documents XML représentant des dépêches sportives de l'Agence France Presse (AFP). 3396 de ces documents (soit 92,6 %) vérifient la contrainte **CNTR** et sont donc utilisables pour nos expérimentations. Nous pouvons ici relever que notre contrainte est vérifiée dans un corpus réel, ce qui signifie qu'elle peut s'avérer réaliste dans certains cas. Les arbres du corpus ont une profondeur moyenne comprise entre 2 et 3, et un facteur de branchement moyen compris entre 4 et 5. Le nombre d'étiquettes différentes est 32. Ce cas est donc plus complexe que nos expériences précédentes, à cause d'un facteur de branchemement plus élevé.

Ces documents suivent tous les mêmes règles d'écriture, et 90 % d'entre eux ont strictement la même structure. Le but de Dryade était donc de retrouver cette structure commune à la majorité des documents. Pour cela, nous avons choisi un seuil de fréquence relatif de 3000/3396.

Sur une machine cadencée à 1 GHz, il a fallu environ 19 heures à Dryade pour trouver les arbres fréquents. Ce temps se décompose de la manière suivante :

Découverte des patterns fréquents de profondeur 1	67081 s (18.6 h)
Découverte des ancres d'accrochages de ces patterns	1592 s (0.4 h)
Itérations d'accrochage	890 s (0.25 h)
Total	69563 s (19.3 h)

Le temps total est donc dominé par l'étape initiale de découverte des patterns fréquents fermés de profondeur 1, qui se résume à autant d'applications de l'algorithme Eclat qu'il y a d'étiquettes. Ce temps très important peut s'expliquer par le fait que souvent, les arbres du corpus ont un facteur de branchement élevé au niveau des profondeurs les plus grandes. Certains nœuds peuvent avoir une dizaine de feuilles par exemple. La saturation par la relation d'ancestralité entre ces feuilles et tous leurs ancêtres va produire beaucoup de paires de labels ancêtre/descendant, qui représentent les items de l'algorithme Eclat. Or Eclat est exponentiel en fonction du nombre d'items.

De plus, notre implémentation paye ici le fait d'utiliser Eclat et non Charm, qui serait beaucoup plus rapide car extrairait directement les itemsets fréquents fermés.

Après cette étape, le calcul du graphe des accrochages et la construction des patterns fréquents sont comparativement très rapides, bien que ce soient les phases les plus complexes de Dryade. Cette situation diffère donc de ce que nous avions observé dans nos expériences sur des données aléatoires.

Nous montrons dans la figure VI.12 le plus gros arbre fréquent découvert par Dryade. Ce plus gros arbre correspond bien à la structure commune des documents XML de la collection. Nous pouvons constater que cette structure est complexe, elle comprend 28 arêtes père-fils.

4 Conclusion

Dans cette étude expérimentale, nous avons vu que Dryade était beaucoup plus efficace que l'approche généraliste Warmr. De plus, nous avons étudié la complexité effective de Dryade, qui est polynômiale en temps en fonction du nombre d'arbres d'entrée, et qui dépend polynômialement du nombre de patterns de profondeur 1 pour le calcul du graphe des accrochages, et de la profondeur des solutions à trouver pour l'exécution des accrochages.

Dans un cas réel difficile, Dryade a pu retrouver la structure commune à 3000 documents XML, et c'est le calcul des patterns fréquents de profondeur 1 qui a occupé la plus grande partie du temps de calcul.

Annexe

Dans cette annexe, nous allons expliquer le comportement de la courbe du nombre de patterns fréquents de profondeur 1, dans un corpus d'arbres de profondeur 3 et dont le facteur de branchement moyen est 1. Pour simplifier, nous supposerons que les arbres ont un facteur de branchement égal à 1, ils sont donc comme dans la figure VI.13. Rappelons que nous avons un biais de génération, et que l'ensemble E des étiquettes est donc divisé en 4 sous-ensembles disjoints de taille égale, et que chaque nœud d'un arbre prend ses étiquettes dans un sous-ensemble différent de E. Le choix de ce sous-ensemble dépend de la profondeur du nœud.

Les patterns fréquents de profondeur 1 ont trois formes possibles dans ce type de corpus : à 3 feuilles, à 2 feuilles et à 1 feuille, comme le montre la figure VI.13.

Nous allons donc calculer l'espérance des patterns fréquents de chacune de ces formes pour un ensemble de labels donnés, le total des espérances des trois formes nous donnera l'espérance de tous les patterns fréquents de profondeur 1. Notre variable sera toujours x, le nombre d'étiquettes.

Pour obtenir l'espérance des patterns fréquents à 3 fils, nous commençons par calculer la probabilité $p_3(x)$ qu'un pattern fréquent à trois fils donné soit inclus dans un arbre A du corpus.

Un pattern fréquent P à trois fils sera inclus dans un arbre du corpus si son nœud racine a été tiré dans le premier sous-ensembre d'étiquettes, et que tous ses fils ont été tirés dans des sous-ensembles différents. Ainsi il est possible d'établir un mapping entre P et un arbre A du corpus. Donc :

$$p_3(x) = (4/x)^4$$

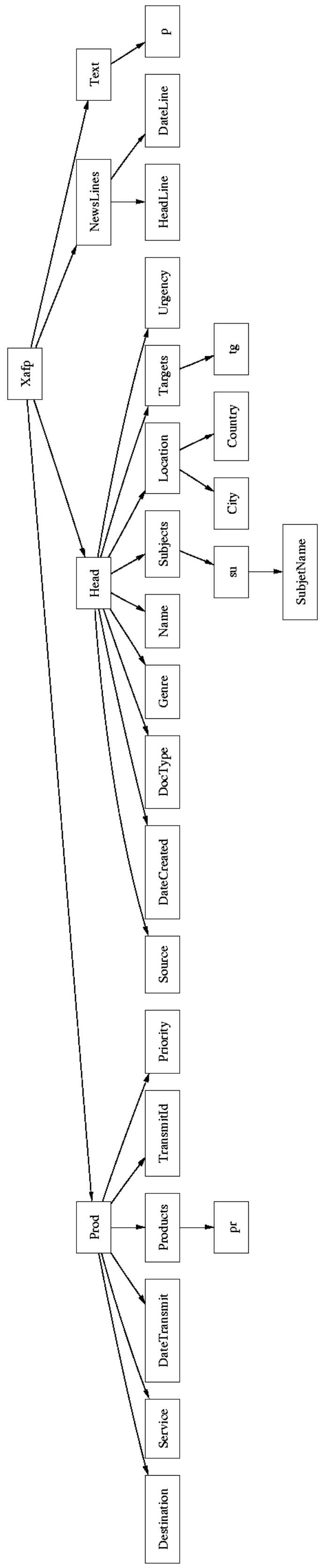


FIG. VI.12 – Plus gros arbre fréquent

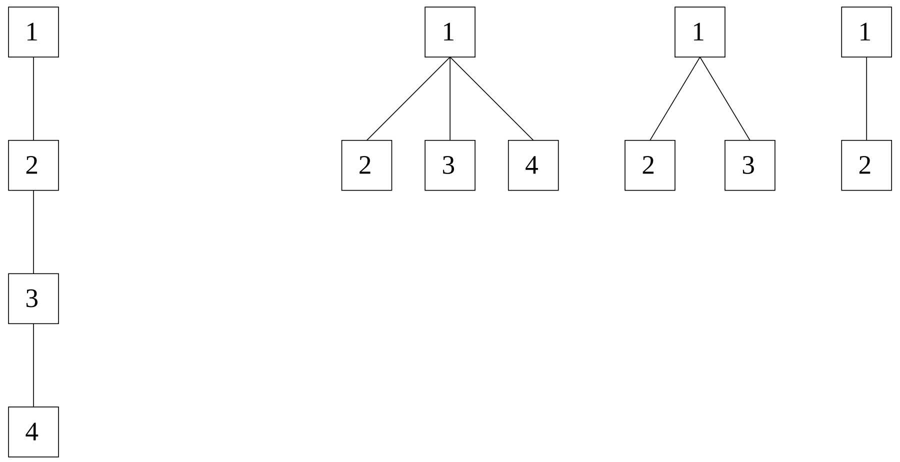


FIG. VI.13 – À gauche, un arbre du corpus, à droite, des patterns fréquents de différentes formes

Pour que le pattern P soit fréquent, il doit apparaître dans au moins ε arbres du corpus. La probabilité qu'un pattern apparaîsse dans exactement k arbres du corpus est calculée selon la loi de Bernouilli :

$$\mathcal{B}(k, p_3(x)) = C_N^k \times p_3(x)^k \times (1 - p_3(x))^{(N-k)}$$

où N est le nombre total d'arbres dans le corpus. Donc pour connaître la probabilité que P soit fréquent, il faut additionner la probabilité que P apparaisse ε fois, celle qu'il apparaisse $\varepsilon + 1$ fois et ainsi de suite.

$$pfrequent_3(x) = \sum_{k=\varepsilon}^{N} \mathcal{B}(k, p_3(x))$$

L'espérance des patterns fréquents de profondeur 1 à trois fils est le produit de la probabilité précédente par le nombre de patterns fréquents de profondeur 1 à trois fils, d'où :

$$E_3(x) = pfrequent_3(x) \times (\frac{|E|}{4})^4$$

Nous procédons de la même manière pour les patterns fréquents de profondeur 1 à deux fils et à 1 fils.

On a de façon évidente :

$$p_2(x) = (4/x)^3$$

donc :

$$pfrequent_2(x) = \sum_{k=\varepsilon}^{N} \mathcal{B}(k, p_2(x))$$

et

$$p_1(x) = (4/x)^2$$

donc :

$$pfrequent_1(x) = \sum_{k=\varepsilon}^{N} \mathcal{B}(k, p_1(x))$$

Pour trouver le nombre de patterns à 2 fils, remarquons qu'il faut d'abord choisir un nœud de l'arbre cible sur lequel il ne sera pas fait de mappings (4 choix), puis il faut instancier les étiquettes des nœuds du pattern. Donc :

$$E_2(x) = pfrequent_2(x) \times 4 \times (\frac{|E|}{4})^3$$

De même, pour le nombre de patterns fréquents de profondeur 1 à 1 fils il faut choisir 2 nœuds de l'arbre parmi 4, puis instancier les étiquettes du pattern :

$$E_1(x) = pfrequent_1(x) \times C_4^2 \times (\frac{|E|}{4})^2$$

La figure VI.14 représente $E_1(x)+E_2(x)+E_3(x)$, nous remarquons qu'il a bien les deux pics que constatés dans nos expériences, qui correspondent pour le pic de droite à une majorité de patterns fréquents de profondeur 1 à 1 fils, et pour le pic de gauche à une majorité de patterns fréquents de profondeur 1 à 2 ou 3 fils.

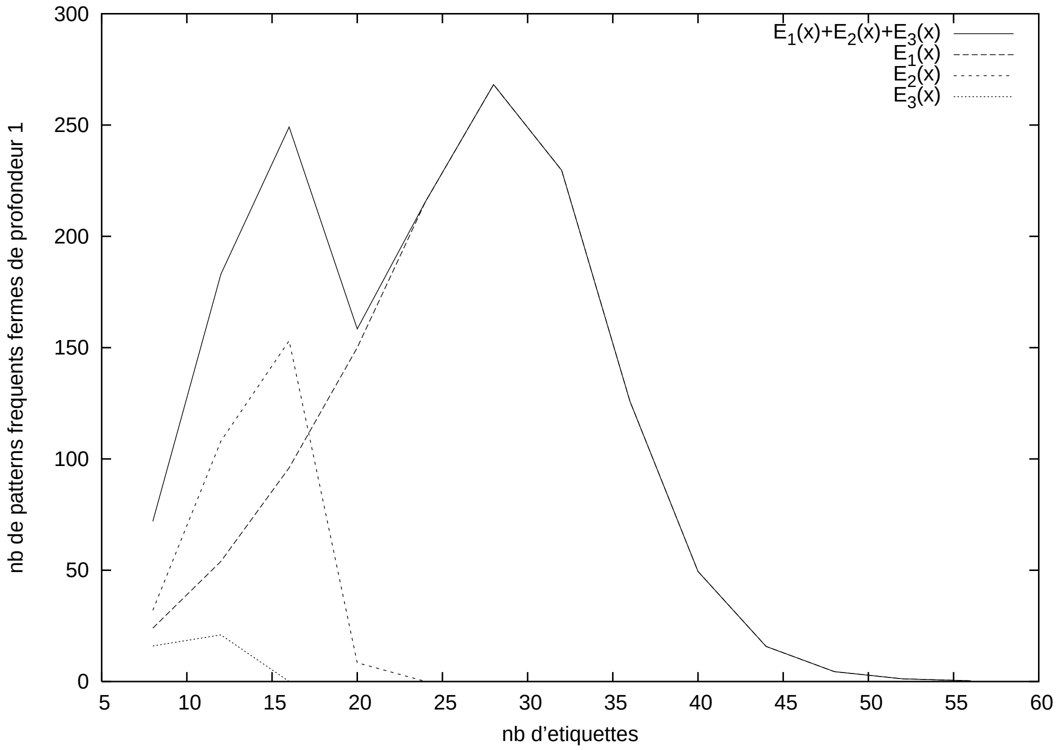


FIG. VI.14 – $E_1(x) + E_2(x) + E_3(x)$

Chapitre VII

Conclusion

Tout au long de ce mémoire, nous nous sommes intéressés au problème de la découverte d'arbres fréquents. Bien qu'il existe déjà des algorithmes généralistes comme Warmr répondant au problème, nous avons préféré nous intéresser aux méthodes spécialisées, en raison de leur plus grande efficacité. Nous avons d'abord passé en revue dans le chapitre III les différentes méthodes de recherche d'arbre fréquents existantes. Nous avons vu que les méthodes proposées résolvent des problèmes différents, car elles utilisent des définitions de l'inclusion d'arbres différentes. La méthode proposée par Asai et al. utilise une définition d'inclusion préservant la relation de parenté, tandis que la méthode proposée par Zaki est plus générale et n'impose que de préserver l'ancestralité.

Toutefois, ces deux méthodes imposent que dans les arbres fréquents trouvés, les fils ou les descendants aient le même ordre que dans les arbres des données. Le problème de la recherche de motifs fréquents ne respectant pas l'ordre des fils ou descendants n'avait jusque là pas été considéré par une approche spécifique. Cela peut entre autres s'expliquer par la difficulté du test d'inclusion d'arbres que ce problème suppose : nous avons vu dans le chapitre II que le test d'inclusion d'arbres dans le cas non-ordonné est NP-complet.

Nous avons présenté plusieurs solutions pour la recherche d'arbres fréquents, dans le cas non-ordonné. La première de ces solutions est l'algorithme approché TreeFinder, présenté dans le chapitre IV. Cet algorithme exploite grâce à des recodages adaptés des méthodes et algorithmes bien connus, à savoir la découverte d'itemsets fréquents dans le cas transactionel, et le calcul de *Least General Generalization*. Toute amélioration sur ces techniques lui bénéficie donc directement. Nous avons vu que l'algorithme TreeFinder est correct, mais que dans le cas général il n'est pas complet. Notre étude expérimentale nous a montré qu'en pratique, tant que le taux de recouvrement entre les paires d'étiquettes ancêtre/descendant des arbres fréquents à découvrir n'est pas trop élevé, la complétude des résultats fournis par TreeFinder est acceptable. C'est un autre point important de TreeFinder et de son étude expérimentale : nous avons pu montrer que notre algorithme approché pouvait dans un certain nombre de cas fournir des résultats satisfaisants. Dans le cadre d'applications réelles où la vitesse prime sur la qualité de l'approximation, un algorithme comme TreeFinder peut donc être exploité avec succès, à condition que les données à traiter n'entraînent pas de perte de complétude trop importante, c'est-à-dire que le taux de recouvrement entre les étiquettes ancêtres/descendants des arbres fréquents ne soit pas trop élevé.

Par la suite, nous avons présenté dans le chapitre V les algorithmes Dryal et Dryade. Ces algorithmes calculent des patterns fréquents, c'est-à-dire des arbres dont un nœud ne peut pas avoir deux fils de même étiquette. L'article [Delobel et al. 2003] a montré que cette restriction sur la forme des arbres n'empêchait pas de trouver des structures arborescentes intéressantes pour effectuer des interrogations de documents XML. Les algorithmes Dryal et Dryade exploitent la nature arborescente des données dans une approche novatrice en construisant au fur et à mesure l'ensemble des solutions, par accrochage et raffinements successifs de patterns fréquents de profondeur 1. Les calculs de fréquences des candidats exploitent intensivement des approches verticales de recherche d'itemsets fréquents, et échappent ainsi à de coûteux tests d'inclusion d'arbres. Nous avons détaillé les algorithmes Dryal et Dryade, et prouvé leur correction et complétude dans le cas général. Nous avons ensuite effectué une étude expérimentale sur notre implémentation de Dryade, dans le chapitre VI. Cette étude a tout d'abord montré que Dryade était plus efficace que l'approche généraliste Warmr, avec un gain de plusieurs ordres de grandeur en terme de temps de calcul et de place mémoire. Ce résultat justifie *a posteriori* l'approche suivie, exploitant les spécificités des structures arborescentes. Par ailleurs, l'amélioration significative des performances est liée à l'utilisation de la contrainte de fermeture, inspirée de Charm et Closed. Une étude empirique de la complexité de Dryade a été présentée, et a montré les bonnes capacités de passage à l'échelle de Dryade. Enfin, l'expérimentation sur des données réelles complexes a montré que Dryade est à même d'être utilisé dans des applications réelles.

Notre contribution principale, matérialisée par les algorithmes Dryal et Dryade, se décompose en deux points.

En premier lieu, les algorithmes Dryal et Dryade reposent sur l'usage intensif d'approches verticales de recherche d'itemsets fermés fréquents. Ces approches assurent non seulement la phase d'initialisation de Dryade, mais interviennent aussi dans l'algorithme au niveau du choix des accrochages à effectuer et de la suppression des feuilles redondantes. L'intérêt de réutiliser les acquis de ces approches est double : tout d'abord, c'est un cadre bien connu, ce qui peut faciliter la compréhension et l'amélioration future de nos algorithmes. Ensuite, ces approches sont le sujet de nombreux travaux, donc toute amélioration qui leur est apportée peut être utilisée immédiatement dans Dryal et Dryade. Ce qui signifie qu'en améliorant la recherche de motifs fréquents propositionnels, on améliore automatiquement la recherche de patterns fréquents. Ce point nous semble particulièrement important.

En second lieu, nous avons défini pour l'algorithme Dryade la notion de pattern fréquent fermé. Nous avons ainsi étendu la notion d'itemset fréquent fermé aux patterns fréquents. Nous avons pu constater dans nos expérimentations que la restriction aux patterns fréquents fermés permettait un gain de performance très important. Nous espérons que l'utilisation de la contrainte de fermeture pourra bénéficier à l'ensemble des recherches en fouille de données structurées, notre approche illustrant sur des bases à la fois théoriques et pratiques l'utilité de telles contraintes.

Perspectives

Le travail présenté débouche sur plusieurs perspectives de recherche. À court terme, une amélioration de notre implémentation actuelle de Dryade nous semble indispensable, afin de lever les restrictions vues dans le chapitre VI. Il sera intéressant de mesurer le gain de performance apportée par cette nouvelle implémentation. Nous comptons aussi étudier diverses optimisations de Dryade, notamment au niveau de la mise à jour du graphe des accrochages. Une utilisation judicieuse de certaines propriétés de la fermeture peut permettre d'anticiper l'élimination de certains arcs inutiles, évitant ainsi de générer certains patterns qui seront forcément redondants, et donc éliminés par la suite.

À moyen terme, il est clair que Dryade est destiné à être intégré dans un processus de fouille de données réel. Le type d'application envisagé serait de partir d'une collection hétérogène de documents XML, dans une telle collection, les requêtes se posent via un langage de requête XML comme XQuery. Mais à cause de l'hétérogénéité de la collection, il peut être très difficile pour l'utilisateur de savoir quelles balises utiliser pour sa requête. Ici DRYADE peut se révèler utile : une exécution de DRYADE sur la collection de documents XML fournira des patterns fréquents, dont on est certain qu'ils apparaissent au moins ε fois dans la collection. Ces patterns peuvent servir de requêtes-type à l'utilisateur, et l'informer sur ce qu'il peut trouver dans la collection. Une présentation des patterns fréquents en formulaires est envisageable pour faciliter leur usage par l'utilisateur (figure VII.1).

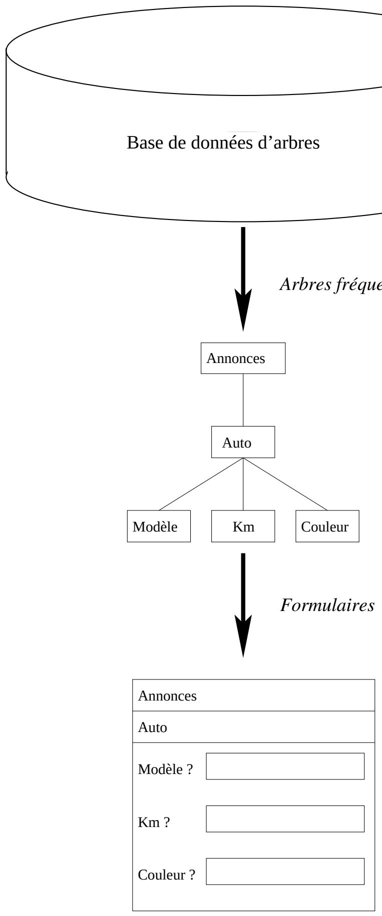


FIG. VII.1 – Un exemple de construction de formulaires

Dans cette figure, un pattern fréquent concernant des annonces de voitures d'occasion est découvert. Présenté sous forme de formulaire, l'utilisateur peut se contenter de remplir les valeurs qui l'intéressent, et la requête sera posée via XQuery.

L'étude d'un problème réel permettra à notre approche de se concentrer sur les problèmes significatifs et de proposer des extensions favorisant le compromis exploitabilité/utilité pertinent.

À long terme, plusieurs perspectives sont ouvertes. Comme nous le mentionnons ci-dessus, une première perspective est d'étendre la découverte de motifs fréquents fermés aux séquences et aux graphes. Cela pourra permettre de gagner en efficacité sur ces problèmes, et de s'attaquer à de nouveaux types d'applications. Par exemple pour les données de type graphe, il existe de nombreux travaux sur les bases de données de molécules chimiques ([Inokuchi et al. 2000], [Maloberti et Suzuki 2003]), et les analyser avec une approche de motifs fréquents fermés serait certainement très intéressant.

Une autre perspective est de relâcher la contrainte sur la forme des arbres découverts par DRYADE, afin qu'il ne découvre plus des patterns fréquents, mais des arbres fréquents, pouvant avoir plusieurs fils de même étiquette. Dans ces conditions le coût des approches de type APRIORI augmente beaucoup : il faut donc découvrir comment rester aussi efficace que possible malgré l'explosion combinatoire.

Une dernière perspective plus théorique est de définir, comme nous l'avons fait pour les patterns fréquents fermés, la notion de pattern fréquent maximal, et de modifier DRYADE pour qu'il calcule de tels patterns. Une perspective passionnante serait alors de découvrir le lien entre les itemsets fréquents / fréquents fermés / fréquents maximaux et les patterns fréquents / fréquents fermés / fréquents maximaux. Cette investigation serait intéressante à mener tant au point de vue des convergences entre algorithmes, que des propriétés théoriques. De plus, une comparaison expérimentale des différences de performances entre algorithmes sur des données propositionnelles classiques, fermés et maximaux et les algorithmes correspondants sur des données structurées serait capitale. On pourrait ainsi apprendre si pour des motifs structurés, il est encore plus intéressant de calculer des motifs fermés ou maximaux que dans les approches propositionnelles. Un but à encore plus long terme serait alors de proposer un cadre général sur la recherche de motifs fréquents, quel que soit le degré de structure des motifs recherchés. Ceci est une instanciation pour les données structurées de type arbres et graphes, avec les contraintes de fermeture et de maximalité, du cadre général défini dans [Mannila et Toivonen 1997].

Table des figures

Liste des tableaux

Liste des Algorithmes

Bibliographie

[Agrawal et al. 1993] RAKESH AGRAWAL AND TOMASZ IMIELINSKI AND ARUN N. SWAMI (1993). « Mining Association Rules between Sets of Items in Large Databases ». Dans P. Buneman et S. Jajodia, editors, *Proceedings of the 1993 ACM SIGMOD International Conference on Management of Data*, pages 207–216, Washington, D.C. **cité page 29**

[Agrawal et Srikant 1994] RAKESH AGRAWAL AND RAMAKRISHNAN SRIKANT (1994). « Fast Algorithms for Mining Association Rules ». Dans *Proceedings of the 20th VLDB Conference*, Santiago, Chile. **cité pages 4,29,34,36**

[Asai et al. 2002] TATSUYA ASAI AND KENJI ABE AND SHINJI KAWASOE AND HIROKI ARIMURA AND HIROSHI SAKAMOTO AND SETSUO ARIKAWA (2002). « Efficient Substructure Discovery from Large Semi-structured Data ». Dans *In Proc. of the Second SIAM International Conference on Data Mining (SDM2002), Arlington, VA*, pages 158–174. **cité pages 8,9,10,41**

[Brin et al. 1997] SERGEY BRIN AND RAJEEV MOTWANI AND CRAIG SILVERSTEIN (1997). « Beyond market baskets : generalizing association rules to correlations ». Dans *In Proc. 1997 Int. Conf. of Management of Data, Tucson, Arizona*, pages 265–276. **cité pages 5,34**

[Carme et al.] J. CARME AND R. GILLERON AND A. LEMAY AND A. TERLUTTE AND M. TOMMASI. « Residual Finite Tree Automata ». Dans *Proceedings of the Seventh Int. Conf. Developments in Language Theory DLT'03*, volume Lecture Notes in Computer Science 2710, pages 171 – 182. **cité page 54**

[Cong et al. 2002] GAO CONG AND LAN YI AND BING LIU AND KE WANG (2002). « Discovering Frequent Substructures from Hierarchical Semi-Structured Data ». Dans *In Proc. of the Second SIAM International Conference on Data Mining (SDM2002), Arlington, VA*. **cité page 53**

[Dehaspe 1998] DEHASPE, LUC (1998). « *Frequent pattern discovery in first-order logic* ». Phd, Department of Computer Science, K.U.Leuven, Leuven, Belgium. 193+xiii pagesURL = http ://www.cs.kuleuven.ac.be/cgi-bin-dtai/publ_info.pl ?id=20990. **cité page 6**

[Dehaspe et Toivonen 1999] DEHASPE, LUC AND TOIVONEN, H. (1999). « Discovery of frequent Datalog patterns ». *Data Mining and Knowledge Discovery*, 3(1) :7–36. URL = http ://www.cs.kuleuven.ac.be/cgi-bin-dtai/publ_info.pl ?id=20765. **cité page 6**

[Delobel et al. 2003] CLAUDE DELOBEL AND CHANTAL REYNAUD AND MARIE-CHRISTINE ROUSSET AND JEAN-PIERRE SIROT AND DAN VODISLAV (2003). « Semantic Integration in Xyleme : a Uniform Tree-based Approach ». *Journal on Data and Knowledge Engineering*, 44(2) :pp 267–298. **cité pages 13,145**

[Gold 1967] E. GOLD (1967). « Language identification in the limit ». *Information and Control*, 10 :447–474. **cité page 54**

[Houtsma et Swami 1995] Maurice A. W. Houtsma and Arun N. Swami (1995). « Set-Oriented Mining for Association Rules in Relational Databases ». Dans P. S. Yu et A. L. P. Chen, editors, *Proceedings of the Eleventh International Conference on Data Engineering, March 6-10, 1995, Taipei, Taiwan*, pages 25–33. IEEE Computer Society. **cité page 29**

[Inokuchi et al. 2000] Akihiro Inokuchi and Takashi Washio and Hiroshi Motoda (2000). « An Apriori-Based Algorithm for Mining Frequent Substructures from Graph Data ». Dans *Principles of Data Mining and Knowledge Discovery*, pages 13–23. **cité pages 8,134,147**

[Jeudy et Boulicaut 2002] Baptiste Jeudy and Jean-François Boulicaut (2002). « Using condensed representations for interactive association rule mining ». Dans *Proceedings of the 6th European conferences on Principles and practice of Knowledge Discovery in Databases ECML/PKDD 2002, Helsinki*, pages 225–236. **cité page 135**

[Jr. 1998] Roberto J. Bayardo Jr. (1998). « Efficiently Mining Long Patterns from Databases ». Dans L. M. Haas et A. Tiwary, editors, *SIGMOD 1998, Proceedings ACM SIGMOD International Conference on Management of Data, Juin 2-4, 1998, Seattle, Washington, USA*, pages 85–93. ACM Press. **cité page 35**

[Kilpeläinen 1992] Pekka Kilpeläinen (1992). « *Tree Matching Problems with Applications to Structured Text Databases* ». PhD thesis, Department of Computer Science, University of Helsinki. Rapport A-1992-6. **cité pages 11,15,17,18**

[Kosala et al.] R. Kosala and J.V. den Bussche and H. Blockeel. « Information extraction from web documents based on local unranked tree automaton inference ». Dans *Proceedings de IJCAI 2003*. **cité page 54**

[Lin et Kedem 1998] Dao-I Lin and Zvi M. Kedem (1998). « Pincer Search : A New Algorithm for Discovering the Maximum Frequent Set ». *Lecture Notes in Computer Science*, 1377 :105– ? ? **cité page 35**

[Lloyd 1986] J. W. Lloyd (1986). « *Foundations of Logic Programming* ». Springer-Verlag New York, Incorporated. **cité page 6**

[Maloberti et Suzuki 2003] Jérôme Maloberti and Einoshin Suzuki (2003). « Improving Efficiency of Frequent Query Discovery by Eliminating Non-relevant Candidates ». Dans *Discovery Science 2003*, pages 220–232. **cité page 147**

[Mannila et Toivonen 1997] H. Mannila and H. Toivonen (1997). « Levelwise search and borders of theories in knowledge discovery ». *Data Mining and Knowledge Discovery*, 1(3) :241–258. **cité page 148**

[Papakonstantinou et Vianu] Yannis Papakonstantinou and Victor Vianu. « DTD inference for views of XML data ». Dans *Proceedings of SIGMOD 2000*. **cité page 54**

[Pasquier et al. 1999] Nicolas Pasquier and Yves Bastide and Rafik Taouil and Lotfi Lakhal (1999). « Discovering Frequent Closed Itemsets for Association Rules ». Dans C. Beeri et P. Buneman, editors, *Database Theory - ICDT '99, 7th International Conference, Jerusalem, Israel, Janvier 10-12, 1999, Proceedings*, volume 1540 of *Lecture Notes in Computer Science*, pages 398–416. Springer. **cité pages 5,36**

[Pei et al. 2000] Jian Pei and Jiawei Han and Runying Mao (2000). « CLOSET : An Efficient Algorithm for Mining Frequent Closed Itemsets ». Dans *ACM SIGMOD Workshop on Research Issues in Data Mining and Knowledge Discovery*, pages 21–30. **cité page 37**

[Plotkin 1970] G. Plotkin (1970). « A note on inductive generalisation ». *Machine Intelligence*, 5 :153–163. **cité pages 6,58**

[Savasere et al. 1995] Ashoka Savasere and Edward Omiecinski and Shamkant B. Navathe (1995). « An Efficient Algorithm for Mining Association Rules in Large Databases ». Dans *The VLDB Journal*, pages 432–444. **cité page 34**

[Termier et al. 2001] Alexandre Termier and Marie-Christine Rousset and Michèle Sebag (2001). « Combining Statistics and Semantics for Word and Document Clustering. ». Dans *Ontology Learning IJCAI workshop*. **cité page 11**

[Termier et al. 2002] Alexandre Termier and Marie-Christine Rousset and Michèle Sebag (2002). « TreeFinder : a First Step towards XML Data Mining ». Dans *International Conference on Data Mining ICDM'02, Maebashi, Japon*. **cité pages 11,55**

[Toivonen 1996] Hannu Toivonen (1996). « Sampling Large Databases for Association Rules ». Dans T. M. Vijayaraman, A. P. Buchmann, C. Mohan, et N. L. Sarda, editors, *In Proc. 1996 Int. Conf. Very Large Data Bases*, pages 134–145. Morgan Kaufman. **cité pages 5,34**

[Wille 1982] R. Wille (1982). « *Ordered Sets* », chapter Restructuring lattices theory : An approach based on hierarchies of concepts, pages 445–470. Reidel, Dordrecht-Boston, i. rival edition. **cité page 27**

[XQY] XQY. « http ://www.w3.org/TR/xquery/ ». **cité page 51**

[Yang et al. 2003a] Liang Huai Yang and Mong Li Lee and Wynne Hsu (2003a). « Efficient Mining of XML Query Patterns for Caching ». Dans *In Proc. 29th Very Large Databases Conference (VLDB'03), Berlin, Allemagne*. **cité page 53**

[Yang et al. 2003b] Liang Huai Yang and Mong Li Lee and Wynne Hsu and Sumit Acharya (2003b). « Mining Frequent Quer Patterns from XML Queries ». Dans *In Proc. Eighth International Conference on Database Systems for Advanced Applications (DASFAA '03), Kyoto, Japon*, page 355.

[Zaki 2002] Mohammed J. Zaki (2002). « Efficiently Mining Frequent Trees in a Forest ». Dans *In Proc. 8th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*. **cité pages 8,9,10,11,44**

[Zaki et Gouda 2003] Mohammed J. Zaki and Karam Gouda (2003). « Fast Vertical Mining Using Diffsets ». Dans *In Proc. 9th International Conference on Knowledge Discovery and Data Mining, Washington, DC*. **cité pages 40,127**

[Zaki et Hsiao 2002] Mohammed J. Zaki and Ching-Jui Hsiao (2002). « CHARM : An Efficient Algorithm for Closed Itemset Mining ». Dans *In Proc. 2nd SIAM International Conference on Data Mining, Arlington*. **cité pages 37,40,96**

[Zaki et al. 1997] M. J. Zaki and S. Parthasarathy and M. Ogihara and W. Li (1997). « New algorithms for fast discovery of association rules ». Dans D. Heckerman, H. Mannila, D. Pregibon, et R. Uthurusamy, editors, *In Proc. of the Third International Conference on Knowledge Discovery and Data Mining (KDD-97)*, pages 283–286. AAAI Press. **cité pages 37,38,127**