Découverte efficace d'arbres fréquents : l'algorithme DryadeParent

Alexandre Termier, Marie-Christine Rousset, Michèle Sebag, Kouzou Ohara, Takashi Washio, Hiroshi Motoda
I.S.I.R., Osaka University
Mihogaoka 8-1, Ibaraki, Osaka 567-0047.
Tel. : 06 6879 8542 - Fax. : 06 6879 8544
E-mail : termier@ar.sanken.osaka-u.ac.jp

Résumé :

Dans ce papier, nous présentons DryadeParent, un nouvel algorithme de fouille de données permettant de découvrir des motifs arborescents fréquents dans des données représentées sous forme d'arbres. Cet algorithme se distingue des autres algorithmes conçus pour la même tâche par la méthode novatrice utilisée pour découvrir les motifs arborescents fréquents, dite méthode des *accrochages*.
Des expérimentations détaillées, aussi bien sur des données réelles que sur des données aléatoires, montrent que notre algorithme est dans la plupart des cas nettement plus rapide que les algorithmes utilisant une méthode classique. Ces expériences montrent également que les performances en temps de calcul des algorithmes utilisant la méthode classique varient beaucoup suivant la structure des motifs arborescents à découvrir. Par contre l'algorithme DryadeParent est peu sensible à cette structure, offrant ainsi de bonnes performances quelle que soit la structure des motifs fréquents à découvrir.

Abstract :

In this paper, we introduce DryadeParent, a new data-mining algorithm for the discovery of frequent subtrees in tree-shaped data. This algorithm, unlike the state of the art tree mining algorithms, uses a new method to find frequent subtrees, called *hooking method*.
Detailed experiments, performed on real data as well as on artificial data, show that our algorithm is in most cases faster than algorithms using a classical method. These experiments also show that the compute-time performances of classical algorithms vary a lot with the structure of the frequent subtrees to discover. In the other hand, DryadeParent's performances are nearly unaffected by this structure, enabling the algorithm to deliver good performances whatever the structure of the frequent subtrees to discover.

Introduction :

Le domaine de l'Informatique appelé «fouille de données» consiste à analyser automatiquement d'importantes quantités de données numérisées pour en extraire des connaissances potentiellement nouvelles et intéressantes. L'une des tâches principales de la fouille de données est de découvrir des motifs fréquents dans les données. Par exemple, analyser des tickets de caisse pour découvrir quels sont les articles que les clients achètent fréquemment ensemble, afin de comprendre leurs habitudes d'achat. Ces dernières années, de plus en plus de données sont produites sous forme structurée, c'est-à-dire que ces données se conforment à une structure de séquence, d'arbre ou de graphe. C'est en particulier le cas pour les données produites en bioinformatique, pour les réseaux informatiques, et les documents XML.
Nous présentons dans ce papier un nouvel algorithme pour la découverte de motifs arborescents fréquents dans des données structurées sous forme d'arbre, appelé DryadeParent. Cet algorithme utilise une méthode originale, dite méthode des accrochages. Nous montrons dans nos expérimentations que cette méthode permet de réduire significativement le temps de calcul par

rapport aux autres méthodes de recherche de motifs arborescents fréquents. Nous montrons de plus que notre méthode est peu affectée par la structure des motifs fréquents à découvrir, contrairement aux méthodes classiques qui exhibent un dépendance exponentielle par rapport au facteur de branchement de ces motifs.
Le plan de ce papier est le suivant : dans la section 2, nous donnons les principales notations et définitions, dans la section 3 nous présentons brièvement l'état de l'art pour la recherche d'arbres fréquents, la section 4 est consacrée à l'algorithme DryadeParent, la section 5 reporte les résultats d'expérimentations détaillées et la section 6 conclut le papier et donne quelques perspectives.

2. Notation et définitions :

Soit $L=(l_1,...l_n)$ un ensemble d'étiquettes. Un *arbre étiqueté* $T=(N, A, root(T), \varphi)$ est un graphe connecté acyclique, où N est l'ensemble des noeuds, $A \subset N \times N$ est une relation binaire sur N définissant l'ensemble des arêtes, $root(T)$ est un noeud particulier appelé *racine*, et φ est une fonction d'étiquetage $\varphi : N \rightarrow L$ assignant une étiquette à chaque noeud de l'arbre. Nous supposons sans perte de généralité que les arêtes ne sont pas étiquetées. Nous supposons aussi que le lecteur est familier avec les notions de: *enfant*, *parent*, *ancêtre* et *descendant* pour les noeuds d'un arbre.
Un arbre est un *arbre d'attributs* si φ est telle que deux noeuds frères ne peuvent avoir le même label (voir [AU05] pour plus de détails).
Inclusion d'arbre : Soit $AT=(N1, A1, root(AT), \varphi 1)$ un arbre d'attributs et $T=(N2, A2, root(T), \varphi 2)$ un arbre. AT est un *sous-arbre induit* de T s'il existe un mapping injectif $\mu : N1 \rightarrow N2$ tel que : 1) μ préserve les labels : $\forall u \in N1 \ \varphi 1(u)=\varphi 2(\mu(u))$ et 2) μ préserve la relation de parenté : $\forall u,v \in N1$ $(u,v) \in A1 \Leftrightarrow (\mu(u),\mu(v)) \in A2$. Nous écrirons cette relation $AT \leq T$, lu : AT est *inclus* dans T.
Si $AT \leq T$, l'ensemble des mappings supportant l'inclusion est noté $EM(AT,T)$. L'ensemble des *occurrences* de AT dans T, noté $Locc(AT,T)$, est l'ensemble des noeuds de T sur lesquels la racine de T est projetée par un mapping de $EM(AT,T)$.
Arbre d'attributs fréquent : Soit $TD=\{T_1,...T_m\}$ une base de données d'arbres. Le *datatree* D_{TD} est l'arbre dont la racine est un noeud sans étiquette, et ayant pour sous-arbres directs les arbres $\{T_1,...T_m\}$. Soit ε un seuil de fréquence absolu. AT est un arbre d'attributs fréquent de D_{TD} si $support(AT) \geq \varepsilon$, où $support(AT)$ est le nombre des arbres T_i de $\{T_1,...T_m\}$ tels que $AT \leq T_i$. L'ensemble des arbres d'attributs fréquents est noté $F(D_{TD}, \varepsilon)$, que nous abrévierons en F dans le reste de ce papier.
Arbre fermé : Un arbre d'attributs fréquent est *fermé* s'il est maximal, par rapport à l'inclusion, pour l'ensemble de ses occurrences, c.a.d. plus formellement qu'un arbre d'attributs fréquent $AT \in F$ est fermé soit s'il n'est pas inclus dans aucun autre arbre d'attributs fréquent, soit dans le cas où il est inclus dans un arbre d'attributs fréquent $AT' \in F$ s'il existe un mapping dans $EM(AT, D_{TD})$ qui ne se retrouve pas dans les mappings de $EM(AT', D_{TD})$. L'ensemble des arbres d'attributs fréquents fermés est noté C.

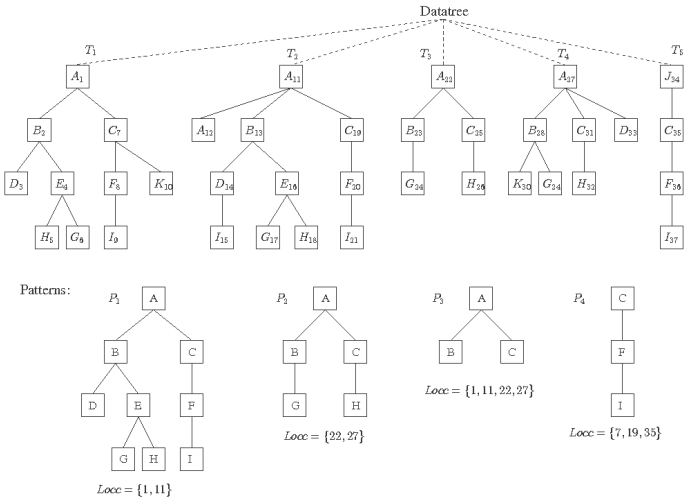

Figure 1 : Exemple de datatree (les identifiants des noeuds sont en indice), et patterns pour ε=2

Dans l'exemple de la figure 1, les arbres fréquents P_1 et P_2 sont fermés car ils ne sont pas inclus dans aucun autre arbre d'attributs fréquent, P_3 est fermé car bien qu'il soit inclus dans P_1 et P_2, ni les occurrences de P_1 ni celles de P_2 ne couvrent toutes les occurrences de P_3, et de la même façon P_4 est aussi fermé.

Le problème qui nous intéresse est de trouver tous les arbres d'attributs fréquents fermés pour un datatree et un seuil de support donné. L'intérêt de rechercher des arbres fréquents fermés

est qu'à partir des arbres de *C* on peut reconstruire tous les arbres de *F* (pas de perte d'information), mais il y a beaucoup moins d'arbres dans *C* que dans *F*, ce qui permet de réduire le temps de calcul. À partir de maintenant nous appelerons *patterns* les éléments de *C*.

3. État de l'art :

Les premiers algorithmes de recherche d'arbres fréquents sont apparus en 2002 dans les travaux de Asai et al. [As02] et Zaki [Za02]. Ces travaux étendent le populaire algorithme Apriori [AS94] à la découverte d'arbres fréquents, en utilisant le même principe *générer et tester* : un arbre candidat est généré, puis il est testé contre la base de données afin de déterminer son support. Si ce support dépasse le seuil de fréquence requis, le candidat est retenu et va servir de base pour construire de nouveaux candidats. La construction d'un candidat à partir d'un arbre fréquent se fait en ajoutant exactement une nouvelle arête à cet arbre fréquent.
Les deux algorithmes précédent se basaient sur des définitions d'arbres imposant que l'ordre des enfants des noeuds soit préservé par le mapping définissant l'inclusion. Cette contrainte sera levée dans les algorithmes présentés dans [As03], [NK03], [Za05], par l'emploi de *formes canoniques*.
Les approches les plus récentes recherchent non pas tous les arbres fréquents mais seulement les arbres fréquents fermés, afin de gagner en performance. À l'heure actuelle ces approches sont au nombre de trois : les algorithmes CMTreeMiner [Chi04] et Clott [AU05] étendent les approches précédentes aux arbres fréquents fermés, tandis que dans [Te04] nous introduisons avec l'algorithme Dryade un nouveau principe d'accrochage, dans cet article la définition d'inclusion d'arbre utilisée est très générale et ne peut être comparée avec les autres travaux.
C'est pourquoi nous introduisons dans le présent article l'algorithme DryadeParent, qui utilise aussi le principe d'accrochage mais avec une définition d'inclusion d'arbre identique à celle utilisée dans [Chi04] et [AU05], afin de pouvoir comparer les différentes méthodes.

4. L'algorithme DryadeParent

Le but de l'algorithme DryadeParent est de découvrir tous les patterns de C, niveau de profondeur par niveau de profondeur, en partant de la racine et en finissant par les feuilles les plus profondes suivant une méthode «en largeur d'abord». Faute de place, nous ne donnons dans cette section qu'une approche intuitive de l'algorithme DryadeParent, et référons le lecteur intéressé à [Te05] pour plus de détails

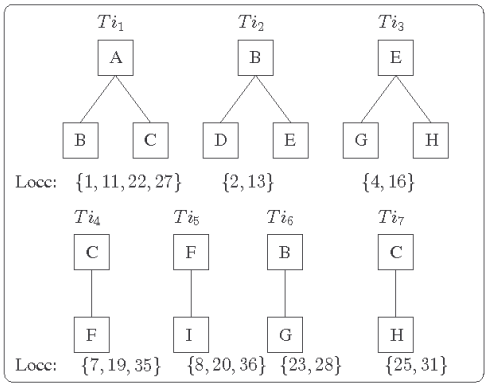

Figure 2 : Les tuiles de notre exemple

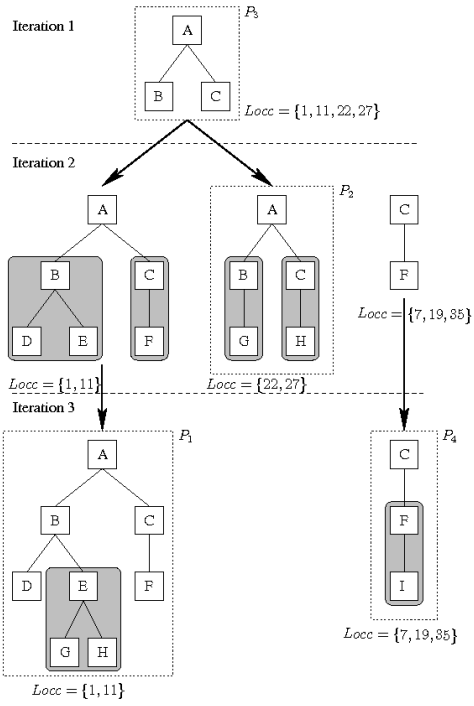

Figure 3 : Processus de découverte de DryadeParent

Tuiles : L'idée de l'algorithme DryadeParent est de construire les patterns niveau de profondeur par niveau de profondeur par le biais d'*accrochages* (définis plus tard) des arbres d'attributs fréquents de profondeur 1, que nous appelons *tuiles*.
Le problème de trouver ces tuiles peut être reformulé comme un problème propositionel de découverte d'itemsets fréquents comme suit : pour chaque étiquette l, utiliser un algorithme de découverte d'itemsets fréquents fermés comme LCM2 [Uno04] pour calculer tous les ensembles fréquents fermés d'étiquettes des noeuds fils pour les noeuds d'étiquette l. Les arbres d'attributs fréquents fermés de profondeur 1 sont obtenus en contruisant des arbres dont la racine porte l'étiquette l et les fils de la racine ont les étiquettes indiquées par les ensembles fréquents fermés précédents. La figure 2 montre les tuiles de notre exemple.
Accrochage des tuiles : Les tuiles précédemment calculées peuvent être *accrochées* ensemble, c.a.d. qu'une tuile dont la racine a l'étiquette l devient un sous-arbre d'une autre tuile ayant une feuille de label l, pour construire un arbre plus complexe. Une stratégie adaptée est nécéssaire pour éviter autant que possible de construire des arbres d'attributs qui se révèleraient non-fermés dans une itération suivante. Notre stratégie consiste à construire des arbres d'attributs qui sont isomorphes aux k premiers niveaux de profondeur des patterns, chaque itération de l'algorithme ajoutant un niveau de profondeur de plus à l'isomorphisme. Pour cela, la première tâche de DryadeParent est de déterminer quelles sont les tuiles qui correspondent aux niveaux de profondeur 0 et 1 des patterns, que nous appelons les *tuiles racines*. Certaines de ces tuiles peuvent être découvertes immédiatement, ce sont les tuiles qui ne peuvent s'accrocher sur aucune autre tuile, et ne peuvent donc pas être des sous-arbres. Ces tuiles seront le point de départ de la première itération de DryadeParent. Dans notre exemple c'est le cas pour Ti_1, comme montré dans la figure 3. Pour les autres tuiles racines, elles peuvent également être utilisées comme sous-arbres dans d'autres patterns que celui pour lequel elles sont racine : elles ne seront utilisées en tant que racine dans l'algorithme qu'à partir du moment où nous serons sûrs que ces tuiles ne sont pas que des sous-arbres, afin de ne pas générer d'arbres d'attributs non fermés. C'est le cas pour Ti_4 dans notre exemple, qui peut être accrochée sur Ti_1. Ce n'est qu'à partir de l'itération 2 que cette tuile sera utilisée comme tuile racine pour construire le pattern P_4.
Le calcul d'un ensemble de tuiles à accrocher sur une tuile racine pour construire de nouveaux arbres d'attributs fréquents fermés est, là encore, effectué par un algorithme propositionel de calcul d'itemsets fréquents fermés. Les arbres d'attributs obtenus par ces accrochages servent de point de départ pour l'itération suivante.
Le processus complet pour notre exemple est illustré dans la figure 3. Sur la tuile racine Ti_1, il est possible d'accrocher soit $\{Ti_2, Ti_4\}$, soit $\{Ti_6, Ti_7\}$, ce dernier accrochage produisant le pattern P_2. Remarquez les occurrences différentes des deux arbres d'attributs construits à partir de Ti_1. Sur l'arbre résultant de l'accrochage de $\{Ti_2, Ti_4\}$ sur Ti_1, on peut alors accrocher Ti_3, ce qui donne le pattern P_1. La tuile Ti_4 n'est pas qu'un sous-arbre de P_1, elle a aussi une occurrence qui n'apparaît pas dans P_1 (le noeud d'identifiant 35) : cette tuile est donc utilisée comme tuile racine, sur laquelle l'algorithme accroche Ti_5, ce qui permet de découvrir le pattern P_4.
La correction et la complétude de ce mécanisme ont été prouvées dans un cas encore plus général dans [Te04these].

5. Expériences

Cette section présente les résultats de la validation expérimentale de DryadeParent, aussi bien sur des jeux de données réels que sur des jeux de données artificiels. Tous les temps de calcul ont été mesurés sur une machine équipée d'un processeur Intel Xéon 2.8 GHz avec 2GB de mémoire (Linux distribution Rocks 3.3.0). DryadeParent est écrit en C++ et utilise l'algorithme de calcul d'itemsets fréquents fermés LCM2 [Uno04], que nous a fourni Takeaki Uno. Les temps de calcul incorporent le chargement et le préprocesing des données.
Notre objectif est de comparer les résultats de DryadeParent et de CMTreeMiner, qui sont à l'heure actuelle les deux seuls algorithmes implémentés pour le problème qui nous intéresse.

Jeux de données réels : La figure 4 montre les temps de calcul obtenus par DryadeParent et

CMTreeMiner pour deux jeux de données classiques, CSLOGS [Za02] et NASA [Chi04],[CA01].

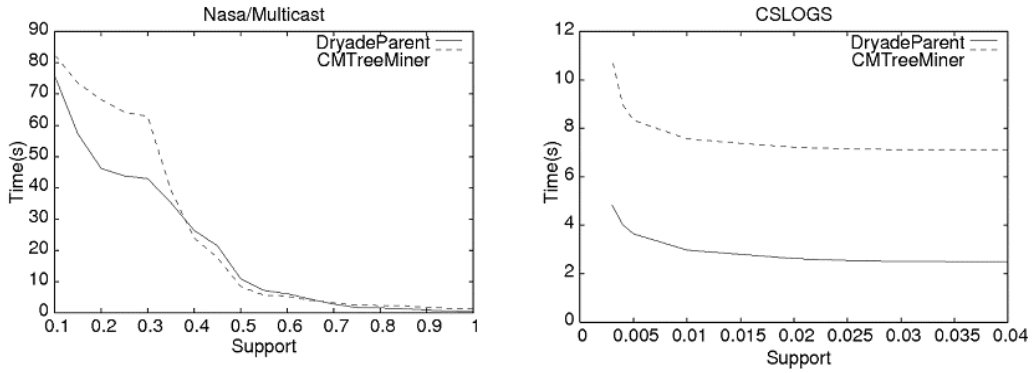

Figure 4 : Temps de calcul / support pour les jeux de données NASA et CSLOGS

DryadeParent est plus de deux fois plus rapide que CMTreeMiner sur le jeu de données CSLOGS. Pour le jeu de données NASA les performances sont similaires, DryadeParent ayant l'avantage sur les valeurs de support les plus basses. Nous nous sommes rendus compte que les patterns trouvés dans ces deux jeux de données étaient très différents : NASA contient des patterns profonds avec un facteur de branchement faible, tandis que CSLOGS contient des patterns peu profonds avec un branchement plus élevé. Nous avons utilisé des jeux de données artificiels pour mieux comprendre l'influence de la structure des patterns sur les performances des deux algorithmes.

Jeux de données artificiels : Dans les études habituelles sur les algorithmes de découverte d'arbres fréquents, au mieux la longueur (c.a.d.le nombre de noeuds) des patterns trouvés est reportée, sans aucune indication sur la structure de ces patterns. Toutefois, le facteur de branchement et la profondeur des patterns interviennent directement dans le processus de génération de candidats, donc ils doivent avoir un impact sur le temps de calcul. Pour vérifier cette hypothèse, nous avons écrit un générateur d'arbres aléatoires qui génère des arbres avec comme paramètres un nombre de noeuds N et un facteur de branchement moyen b. Les noeuds sont étiquetés avec leur identifiant en ordre préfixe, donc dans un arbre il n'y a pas deux noeuds avec la même étiquette. Nous avons généré des arbres avec N=100 noeuds et b variant entre 1.0 et 5.0 par incréments de 0.1. Pour chaque valeur de b nous avons généré 10000 arbres aléatoires et les avons regroupés suivant leur profondeur d, pour un couple (b,d) nous avons produit un temps de calcul moyen en faisant la moyenne des temps de calcul pour tous les arbres de facteur de branchement moyen b et de profondeur d. La figure 5(a) montre le logarithme de ces temps de calcul par rapport au facteur de branchement moyen b, tandis que la figure 5(b) montre le logarithme de ces temps de calcul moyen par rapport à la profondeur d.

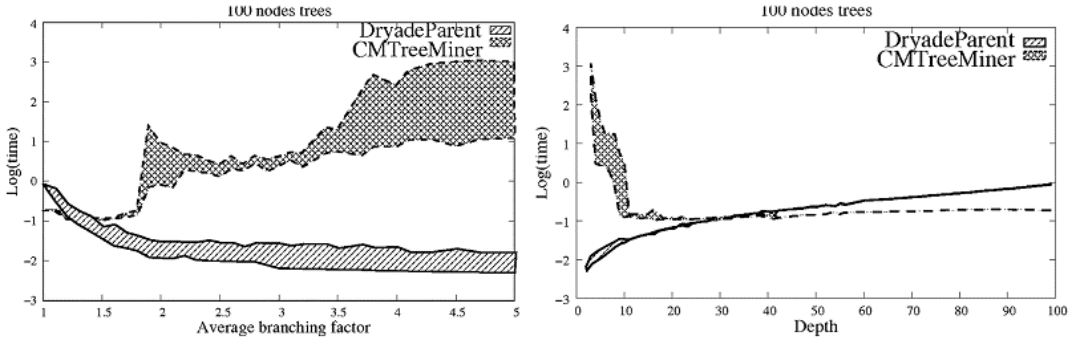

Fig. 5(a) Log(temps) / b Fig. 5(b) Log(temps) / d

La figure 5(a) montre que DryadeParent est plus rapide que CMTreeMiner de plusieurs ordres de grandeur pour des valeurs de facteur de branchement moyen excédant 1.3, ce qui est le cas dans la plus grande partie de nos expériences. Pour des valeurs de branchement moyen plus faibles,

CMTreeMiner a un petit avantage. Les patterns ayant un faible facteur de branchement moyen ont nécéssairement une grande profondeur, ce qui est confirmé par la figure 5(b). Cette figure montre que le temps de calcul pour DryadeParent dépend linéairement de la profondeur des patterns. Ce n'est pas surprenant : chaque itération de DryadeParent calcule un niveau de profondeur de plus pour les patterns, donc plus les patterns seront profonds plus il faudra faire d'itérations.
Par contre CMTreeMiner affiche un dépendance sur le facteur de branchement moyen, mais pour une valeur donnée de *b* le temps de calcul peut varier considérablement, et est particulièrement élevé pour les faibles profondeurs. Les contraintes sur le générateur d'arbres aléatoire imposent qu'un arbre de faible profondeur avec un fort facteur de branchement moyen aie quelques noeuds avec un très grand facteur de branchement. Nous avons représenté dans la figure 6 le temps de calcul en fonction du facteur de branchement *maximal* des noeuds des patterns.

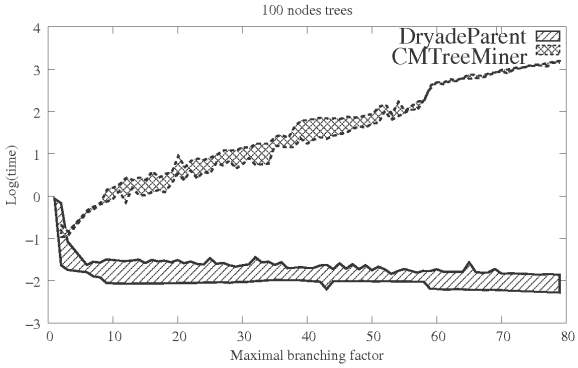

Figure 6 : Log(temps) / facteur de branchement maximal

DryadeParent n'est presque pas affecté par le facteur de branchement maximal, par contre le temps de calcul de CMTreeMiner dépend exponentiellement de ce paramètre.

6. Conclusion et perspectives

Dans ce papier, nous avons présenté l'algorithme DryadeParent, basé sur le calcul de tuiles dans les données et sur une stratégie efficace d'accrochage qui reconstruit les patterns à partir de ces tuiles.
Des expériences détaillées ont montré que DryadeParent est plus rapide que CMTreeMiner dans la plupart des cas, et que ses performances sont robustes par rapport à la structure des patterns à trouver. Nous avons proposé de nouveaux jeux de donnés artificiels prenant en compte la structure des patterns pour tester le comportement des algorithmes de découverte d'arbres fréquents.
Améliorer ces tests et faire des analyses encore plus détaillées est l'une de nos futures directions de recherche. Un autre de nos objectifs est d'étendre DryadeParent à des structures plus générales que les arbres d'attributs.

References

[As02] T. Asai, K. Abe, S. Kawasoe, H. Arimura, H. Sakamoto et S. Arikawa. Efficient substructure discovery from large semi-structured data. In *Proc. of the Second SIAM International Conference on Data Mining (SDM2002), Arlington, VA.*, pages 158-174, Avril 2002.
[As03] T. Asai, H. Arimura, T. Uno, et S. Nakano. Discovering frequent substructures in large unordered trees. In *Proc. of the 6th International Conference on Discovery Science (DS'03)*, pages 47-61, 2003.
[AS94] R. Agrawal et R. Srikant. Fast algorithms for mining association rules. In *Proc. of the 20th VLDB Conference, Santiago, Chile*, 1994.
[AU05] H. Arimura et T. Uno. An output-polynomial time algorithm for mining frequent closed attribute trees. In *15th International Conference on Inductive Logic Programing (ILP'05),* 2005.
[CA01] R. Chalmers et K. Almeroth. Modeling the branching characteristics and efficiency gains of global multicast trees. In *Proc. of the IEEE INFOCOM'01*, Avril 2001.
[Chi04] Y. Chi, Y. Yang, Y. Xia et R.R. Muntz. CMTreeMiner : Mining both closed and maximal

frequent subtrees. In *Proc. of the 8th Pacific-Asia Conference on Knowledge Discovery and Data Mining (PAKDD'04),* 2004.
[NK03] S. Nijssen et J.N. Kok. Efficient discovery of frequent unordered trees. In *First International Workshop on Mining Graphs, Trees and Sequences (MGTS'03)*, 2003.
[Te04] A. Termier, M.C. Rousset et M. Sebag. Dryade : a new approach for discovering closed trees in heterogeneous tree databases. In *Proc. of 4th International Conference on Data Mining (ICDM'04), Brighton, England,* pages 543-546, 2004.
[Te04these] A. Termier. Extraction d'arbres fréquents dans un corpus hétérogène de données semi-structurées : application à la fouille de documents XML. Thèse de doctorat, LRI, Université de Paris-Sud, France, Avril 2004.
[Te05] A. Termier, M.C. Rousset, M. Sebag, K. Ohara, T. Washio et H. Motoda. Computation-time efficient and robust attribute tree mining with DryadeParent. In *Third International Workshop on Mining Graphs, Trees and Sequences (MGTS'05)*, 2005.
[Uno4] T. Uno, M. Kiyomi et H. Arimura. LCM v2.0 : Efficient mining algorithms for frequent/closed/maximal itemsets. In *Second Workshop on Frequent Itemset Mining Implementations (FIMI'04),* 2004.
[Za02] M.J. Zaki. Efficiently mining frequent trees in a forest. In *Proc. of the 8th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, Juillet 2002.
[Za05] M.J. Zaki, Efficiently mining frequent embedded unordered trees. *Fundamenta Informatica, special issue on Advances in Mining Graphs, Trees and Sequences*, 65(1-2):33-52, Mars/Avril 2005.

Remerciements
Les auteurs tiennent à remercier Takeaki Uno pour leur avoir fourni le code source de l'algorithme LCM2, ainsi que Yun Chi pour avoir mis à leur disposition son algorithme CMTreeMiner et le jeu de données NASA.