Vers le prototypage rapide de programmes de fouille de données

Frédéric Flouvat[1,2], Fabien De Marchi[2,3], Jean-Marc Petit[2,4]

[1] Laboratoire LIMOS, UMR CNRS 6158
Université Clermont-Ferrand II,
63 177 Aubière, France
flouvat@isima.fr

[2] Laboratoire LIRIS, UMR CNRS 5205
[3] Université Lyon I [4] INSA Lyon
69 621 Villeurbanne, France
{fabien.demarchi, jmpetit}@liris.cnrs.fr

Résumé

Bien que guidées par des problèmes réels, les techniques mises au point dans le cadre de la recherche en fouille de données sont encore peu utilisées et leur mise en œuvre reste confinée à la discrétion de quelques programmeurs spécialistes. Ce "transfert technologique" est donc freiné par un certain nombre de verrous, parmi lesquels le temps nécessaire à la mise au point des programmes opérationnels. Dans cet article, nous considérons une classe importante et large de problèmes de fouille de données : les problèmes d'extraction de motifs intéressants dans les données. Dans cette classe, nous nous intéressons au sous-ensemble particulier des problèmes *représentables par des ensembles*. A partir d'une caractérisation théorique d'un problème donné, nous proposons une librairie facilitant la résolution de tels problèmes, basée sur l'utilisation d'algorithmes et de structures de données génériques, et passant à l'échelle. Les caractéristiques et optimisations des algorithmes sont transparentes pour le programmeur, et seule l'implantation des propriétés spécifiques à son problème est laissée à sa charge. D'après les premiers résultats, les programmes opérationnels obtenus affichent des performances très intéressantes au regard de la rapidité et la simplicité de leur obtention. La librairie en cours de développement, implantée en C++, est accessible sur internet et libre de droits.

Mots clés : Fouille de données, extraction de motifs, librairie C++.

1 Introduction

Bien que guidées par des problèmes réels, les techniques mises au point dans le cadre de la recherche en fouille de données sont encore peu utilisées et leur mise en œuvre reste confinée à la discrétion de quelques programmeurs spécialistes. Ce "transfert technologique" est donc freiné par un certain nombre de verrous, parmi lesquels *le temps nécessaire à la mise au point des programmes opérationnels*.

Pourtant, différents ateliers [3, 2, 13] ont eu pour objectif de mettre à la disposition de la communauté scientifique des implémentations des principaux algorithmes de fouille de données[1]. Ces travaux visent les problèmes phares de fouille de données : motifs fréquents et variantes, classification supervisée et classification non supervisée. Chaque implémentation est accompagnée d'une présentation de l'algorithme, et permet de résoudre un problème par-

[1] La motivation première a été de permettre une évaluation plus juste des travaux du domaine.

ticulier. Les implémentations et les codes sources des algorithmes sont accessibles sur internet et libres de droits [3, 2, 13].

Toutefois, l'adaptation de ces programmes pour résoudre tout autre problème que le problème initialement étudié s'avère difficile : une modifications du code, même légère, pour la prise en compte de caractéristiques spécifiques, requiert une connaissance approfondie des techniques et optimisations mises en œuvre. En effet, les différences de performances entre les programmes sont liées, pour une grande part, à la capacité du programmeur à développer des structures de données originales, et à assurer une gestion brillante de la mémoire et des entrées/sorties. Le corollaire est que le code source, fusse-t-il libre et (parfois) commenté, est en général complexe et intimement lié à un problème donné.

Récemment, nous avons été confrontés à ce problème en étudiant différentes représentations condensées des motifs fréquents [11] telles que les motifs générateurs (aussi appelés clés ou libres) fréquents [29, 7] ou les motifs essentiels fréquents [9]. En théorie, l'extraction de ces représentations correspond au simple ajout d'une contrainte anti-monotone au prédicat "être fréquent". Pourtant, l'adaptation d'une implémentation d'un algorithme classique de découverte des motifs fréquents telle que l'implémentation d'*Apriori* de C. Borgelt [5], s'est avérée longue et complexe.

Cette constatation s'est renforcée lorsque nous avons utilisé des algorithmes d'extraction de motifs fréquents pour résoudre des problèmes totalement différents tels que l'extraction de dépendances d'inclusion [26] et la résolution de certaines phases du processus de réécriture de requêtes dans un système d'intégration de sources de données hétérogènes [19, 18]. Dans ces contextes, il ne suffisait pas de rajouter une contrainte au prédicat, mais il fallait aussi redéfinir des aspects majeurs de l'implémentation tels que le stockage et l'accès aux données, le prédicat et la solution en sortie de l'algorithme.

Contribution Dans cet article, nous considérons une classe importante et large de problèmes de fouille de données : les problèmes d'extraction de motifs intéressants dans les bases de données, et plus particulièrement aux problèmes *représentables par des ensembles* [24]. A partir d'une caractérisation théorique d'un problème donné, nous proposons une librairie facilitant la résolution de tels problèmes, basée sur l'utilisation d'algorithmes et de structures de données génériques, et passant à l'échelle. Les caractéristiques et optimisations de l'algorithme sont transparentes pour le programmeur, et seule l'implantation des propriétés spécifiques à son problème est laissée à sa charge. De plus, nous présentons, au travers d'un exemple, une méthodologie pour guider l'utilisateur dans ces choix, et l'aider à reformuler (si possible) son problème sous la forme d'un problème d'extraction de motifs intéressants représentable par des ensembles ; nous l'assistons alors dans le développement des différents composants.

Cette librairie est directement issue de notre expérience dans le développement et l'adaptation d'algorithmes de découverte de motifs pour résoudre des problèmes tels que la découverte de représentations condensées des motifs fréquents [11], l'extraction de dépendances fonctionnelles [22] et d'inclusion [26], et la réécriture de requêtes en terme de vues en présence de contraintes de valeurs [19, 18].

Ce travail en cours de réalisation affiche des performances très intéressantes au regard de la rapidité et la simplicité d'obtention d'un programme opérationnel. La librairie en cours de développement, implantée en C++, est accessible sur internet et libre de droits[2].

[2] http ://liris.cnrs.fr/~fflouvat/index.html

Organisation de l'article La section suivante rappelle le cadre théorique sur lequel repose notre travail. Dans la section 3, nous donnons l'exemple de trois problèmes pouvant s'insérer dans ce cadre. La section 4 propose une méthodologie permettant, pour un problème donné, de vérifier son adéquation au cadre théorique. La section 5 présente les différents modes de parcours disponibles dans la version courante, ainsi que des pistes pour choisir l'un ou l'autre de ces parcours. Des éléments de notre implémentation générique, ainsi que quelques résultats expérimentaux sont exhibés dans la section 6. La section 7 fait un état de l'art des contributions exisantes face au problème étudié. Nous concluons ce travail et mettons en avant quelques perspectives dans la section 8.

2 Cadre de l'étude

Nous rappelons ici le cadre théorique défini dans [24], avec des modifications mineures, définissant les problèmes de découverte de motifs intéressants représentable par des ensembles. Ce cadre est celui que nous utilisons dans cet article.

Bordures d'une théorie Soient une base de données $\mathbf{d}$, un langage fini $\mathcal{L}$ représentant des motifs (au sens large), et un prédicat Q permettant d'évaluer si un motif $\varphi \in \mathcal{L}$ est "intéressant" dans $\mathbf{d}$. Soit la tâche de fouille de données consistant à extraire les motifs intéressants de $\mathbf{d}$ relativement à $\mathcal{L}$ et Q, appelé théorie de $\mathbf{d}, \mathcal{L}, Q$, et définie par :

$Th(\mathcal{L}, \mathbf{d}, Q) = \{\varphi \in \mathcal{L} | Q(\mathbf{d}, \varphi) \text{ est vraie}\}$.

Supposons en outre qu'une relation de spécialisation/généralisation, notée $\preceq$, est définie sur les éléments de $\mathcal{L}$. On dira que φ est plus général (resp. plus spécifique) que θ, si $\varphi \preceq \theta$ (resp. $\theta \preceq \varphi$). Nous supposons que le prédicat Q est anti-monotone (resp. monotone) respectivement à l'ordre $\preceq$. Ce qui signifie que pour chaque $\theta, \varphi \in \mathcal{L}$ tels que $\varphi \preceq \theta$, on a : $Q(\mathbf{d}, \varphi)$ est faux (resp. vrai) $\Longrightarrow$ $Q(\mathbf{d}, \theta)$ est faux (resp. vrai).

Lorsque le prédicat est anti-monotone, l'ensemble $Th(\mathcal{L}, \mathbf{d}, Q)$ est dit "fermé par le bas". Dans ce cas, il peut alors être représenté de façon équivalente par ses bordures positive et négative :

- *Sa bordure positive*, notée $\mathcal{B}d^+(Th(\mathcal{L}, \mathbf{d}, Q))$, est composée des motifs intéressants les plus spécialisés par rapport à la relation d'ordre.
- *Sa bordure négative* , notée $\mathcal{B}d^-(Th(\mathcal{L}, \mathbf{d}, Q))$, est composée des motifs non intéressants les plus généraux par rapport à la relation d'ordre.

Si le prédicat est monotone, l'ensemble $Th(\mathcal{L}, \mathbf{d}, Q)$ est dit "fermé par le haut", et la notion de bordures existe aussi. La bordure positive est alors composée des motifs intéressants les plus généraux, et la bordure négative des motifs non intéressants les plus spécialisés.

L'union de ces deux bordures constitue *la bordure* de $Th(\mathcal{L}, \mathbf{d}, Q)$.

Notons que lorsqu'une mesure est associée aux motifs intéressants, les bordures peuvent ne pas suffire pour "reconstruire" l'ensemble des motifs et leur mesure associée. C'est le cas notamment pour le *support* défini pour les motifs fréquents [1].

Dans le paragraphe suivant, la dernière hypothèse requise de ce papier est donnée. Elle permet de ramener le problème initial dans un cadre ensembliste via un isomorphisme.

Représentation ensembliste d'une théorie Soit $(\mathcal{L}, \preceq)$ l'ensemble ordonné composé de l'ensemble $\mathcal{L}$ de tous les motifs exprimables dans le langage, et d'un ordre partiel (relation de généralisation) $\preceq$ sur les motifs de $\mathcal{L}$. Soit également R un ensemble fini d'éléments (ou atomes). Il est parfois possible de définir un isomorphisme f de $(\mathcal{L}, \preceq)$ vers $(\mathcal{P}(R), \subseteq)$, ayant pour propriété de conserver l'ordre des éléments,

i.e.

$$X \preceq Y \Longleftrightarrow f(X) \subseteq f(Y)$$

Si de plus la fonction f^{-1} est calculable, on dit alors que $(\mathcal{L}, \preceq)$ (ou simplement $\mathcal{L}$ quand $\preceq$ est implicite dans le contexte) *a une représentation ensembliste*. Notons qu'une condition nécessaire est que le nombre total de motifs dans $\mathcal{L}$ soit une puissance de 2.

L'obtention d'une telle représentation a beaucoup d'avantages et permet d'envisager une généricité importante.

Par exemple, lors d'un parcours "par niveau" de l'espace de recherche, la génération des candidats du niveau k à partir des motifs vrais de niveau $k-1$ peut-être faite efficacement au sein d'un treillis des parties en utilisant l'algorithme *AprioriGen* [1]. De plus, certains algorithmes [14, 26] effectuent des passages d'une bordure à une autre dans leur stratégie d'exploration. Une telle opération est appelée *dualisation* puisque les deux bordures sont des représentations duales d'un même ensemble. Or, cette étape peut être réduite au calcul des transversaux minimaux d'un hypergraphe, dès l'instant où l'espace de recherche est un treillis des parties d'un ensemble. Ainsi, ces méthodes peuvent être utilisées sans modification pour résoudre un problème représentable par des ensembles, à l'application près de la fonction de transformation.

Néanmoins, notons que le problème d'extraction de séquences fréquentes n'est pas représentable par des ensembles. Même si ce type de problème est très intéressant, il échappe à l'objectif principal de ce papier.

3 Exemples de problèmes étudiés

Nous décrivons brièvement trois problèmes entrant dans ce cadre sur lesquels nous avons contribué dans le passé, et qui sont à l'origine de cette librairie. L'objectif de cette liste n'est pas l'exhaustivité, mais plutôt de tenter de dégager une expertise utile tant sur le plan technique que méthodologique.

3.1 Problèmes liés aux motifs fréquents

La découverte des motifs fréquents pour les règles d'association est certainement le problème la plus célèbre de ce cadre. C'est également le cas de plusieurs représentations condensées des motifs fréquents qui ont été étudiées [23, 8] : les motifs générateurs fréquents [29, 7] et les motifs essentiels fréquents [9]. Leur objectif est double : améliorer si possible l'efficacité de l'extraction des motifs fréquents, et compresser leur stockage pour un usage ultérieur. Leur extraction induit souvent l'ajout d'un prédicat anti-monotone à celui de la fréquence, la conjonction des deux prédicats étant trivialement anti-monotone. C'est le cas notamment des motifs générateurs fréquents et des motifs essentiels fréquents. Notre librairie permet donc le développement rapide de programmes pour l'extraction de tels motifs, avec l'insertion au besoin de contraintes anti-monotones supplémentaires.

Ce cadre nous a permi d'effectuer une étude expérimentale des jeux de données pour plusieurs représentations condensées dont les motifs générateurs fréquents et les motifs essentiels fréquents. Les résultats expérimentaux obtenus ont été à l'origine d'une nouvelle classification des jeux de données [11].

3.2 Extraction de contraintes dans les bases de données

La connaissance de contraintes dans les bases de données est précieuse pour un grand nombre d'applications telles que l'intégration des données, l'optimisation de requêtes ou la mise à jour à travers les vues. Lorsque les

contraintes ne sont pas disponibles, une approche consiste à les induire à partir des données disponibles, et naturellement conduit à des problèmes de fouille de données.

Dans le modèle relationnel, les classes de contraintes les plus importantes étant les dépendances fonctionnelles (DF) et les dépendances d'inclusion (DI), qui généralisent respectivement les notions de clé et clé étrangère, on peut définir les deux problèmes de fouille de données décrits ci-après.

La découverte des dépendances fonctionnelles Suivant l'approche choisie pour résoudre ce problème, le cadre décrit dans ce papier pourra ou non être utilisé. En effet, si la couverture des DF en sortie est la couverture canonique (ensemble des DF avec un seul attribut en partie droite et partie gauche minimale) alors le cadre peut être utilisé. En revanche, si l'on souhaite générer directement une couverture minimum en nombre de DF (base de Duquenne-Guigues par exemple), d'autres approches doivent être envisagées.

Focalisons nous sur la couverture canonique. Soit r une relation sur le schéma R. Le problème de la découverte de la couverture canonique peut être abordé en fixant un attribut A en partie droite, et en recherchant les ensembles d'attributs X tels que $r \models X \rightarrow A$. Le problème se décompose alors en autant de sous-problèmes qu'il y a d'attributs. Le prédicat considéré, i.e. X intéressant ssi $r \models X \rightarrow A$ satisfaite, est trivialement monotone. Par conséquent, chacun de ces sous-problèmes est une instance du cadre qui nous intéresse dans ce papier.

Notons que d'autres approches peuvent aussi être étudiées dans ce cadre, comme par exemple celle présentée dans [28].

La découverte des dépendances d'inclusion Soient r et s deux relations de schéma R et S respectivement. Une dépendance d'inclusion est une expression de la forme $R[X] \subseteq S[Y]$, où X et Y sont des séquences d'attributs incluses respectivement dans R et S. La DI $R[X] \subseteq S[Y]$ est satisfaite dans (r, s) si toutes les valeurs des X dans r sont aussi des valeurs de Y dans s. Cette notion généralise les contraintes de clés étrangères. Un ordre partiel sur les DI peut être défini de la manière suivante : si i et j sont deux DI, $j \preceq i$ si j peut être obtenu en effectuant la même projection sur les deux parties de i. Par exemple, $R[AB] \subseteq S[EF] \preceq R[ABC] \subseteq S[EFG]$. Le problème de fouille de données sous-jacent peut être formulé de la manière suivante : "Soit une base de données, trouver toutes les dépendances d'inclusion satisfaites dans cette base de données" [20, 24, 21, 26].

De plus, malgrès la notion d'ordre inhérente aux DI, ce problème est représentable par des ensembles [25]. L'intuition est la suivante : Soit I_1 l'ensemble des dépendances d'inclusion satisfaites de taille 1. La fonction ens permet d'associer à chaque dépendance d'inclusion de $\mathcal{L}$, un ensemble de dépendances d'inclusion unaires de la manière suivante : $ens(i) = \{j \in I_1 | j \preceq i\}$.

Par exemple, considérons la dépendance d'inclusion $R[ABC] \subseteq S[FGH]$, avec $\{A, B, C\} \in schema(R)$ et $\{F, G, H\} \in schema(S)$. On a alors $ens(R[ABC] \subseteq S[FGH]) = \{R[A] \subseteq S[F], R[B] \subseteq S[G], R[C] \subseteq S[H]\}$.

En considérant uniquement les DI dont la partie gauche respecte un ordre fixé (et sans perte d'information grâce à une règle d'inférence), on obtient que la fonction ens est bijective et admet une fonction inverse ens^{-1}.

Cette transformation est simple mais nécessite de restreindre les DI prises en compte afin d'assurer les bonnes propriétés de la transformation. Les détails sont omis, cf [25] pour plus d'informations.

3.3 Réécriture de requêtes

La réécriture de requêtes à partir de vues est une problématique importante, dont les applications vont de l'optimisation de requêtes à

l'intégration de sources de données hétérogènes.

Dans [19, 18], ce problème a été étudié en présence de contraintes de valeurs, et certaines étapes de la réécriture ont pu se formuler dans le cadre de fouille de données que nous abordons dans ce papier. Ainsi une implémentation a pu être développée, permettant le passage à l'échelle en fonction de nombre de vues traitées [18].

4 Aspects méthodologiques

Nous souhaitons dans cette section donner des éléments de méthodes pour permettre à un analyste d'utiliser concrètement ce que nous proposons. Pour cela, nous prenons un exemple classique des bases de données, à savoir la *découverte des clés minimales dans une relation*. Nous montrons un cheminement possible à effectuer, des aspects théoriques aux considérations plus pratiques, pour utiliser notre librairie et obtenir un code efficace pour ce problème.

Soit r une relation définie sur un schéma de relation R. Les attributs associés à R sont notés $schema(R)$. Afin de simplifier les notations, nous utilisons celles de l'algèbre relationnelle. Nous rappelons tout d'abord la définition des clés et super clés en évitant de manipuler explicitement un ensemble de DF afin de simplifier les notations.

Définition 1 *Soit r une relation sur R, et $X \subseteq schema(R)$. X est une* super clé *de r si on a pout tout $u, v \in r, u[X] \neq v[X]$. X est une* clé minimale *de r (ou simplement* clé*) si X est une super clé de r et $\forall Y \subset X, \exists u, v \in r, u[Y] = v[Y]$.*

Dans la suite, on note respectivement $SCle(r)$ et $Cle(r)$ l'ensemble des super clés et clés de R. Le problème qui nous intéresse est, à partir d'une relation r sur R, d'extraire toutes les clés minimales de r. Notons que le problème de décision qui consiste à savoir s'il existe, dans une relation donnée, une clé de taille inférieure à k donné est NP-Complet [4].

Dans la suite, nous nous assurons que le problème d'extraction des super clés d'une relation peut se formuler dans le cadre qui nous intéresse. L'extraction des clés minimales sera alors caractérisée dans ce cadre.

4.1 Définir l'espace de recherche

Dans un premier temps, il faut déterminer le langage qui permet d'exprimer les motifs de l'espace de recherche, puis une relation d'ordre partiel parmi ces motifs.

Ici, les super clés possibles sont tous les sous-ensembles de $schema(R)$. Ainsi, les motifs du langage sont les éléments de $\mathcal{P}(schema(R))$. L'ordre partiel $\preceq$ est trivialement l'inclusion ensembliste $\subseteq$.

4.2 Définir le prédicat

Identifier le prédicat est souvent aisé, il suffit d'écrire formellement ce que "intéressant" signifie dans le contexte. Pour un motif et une base de données, il faut écrire la condition qui rend ce motif intéressant dans cette base de données. Ici, le prédicat est "être une super clé".

Il nous faut alors démontrer la propriété de monotonie (ou d'anti-monotonie) liant la satisfaction du prédicat et l'ordre partiel entre les motifs. Nous allons montrer que le prédicat "être super clé" est monotone relativement à l'inclusion.

Propriété 1 *Soit r une relation sur R et $X \subseteq schema(R)$. Notons $P(X, r)$ le prédicat "être super clé".*
$P(X, r)$ est monotone par rapport à $\subseteq$.

Preuve 2 *Nous devons montrer que pour tout X,Y tel que $X \subseteq Y$, on a P(X,r) vrai $\Rightarrow$ P(Y,r) vrai.*
X est une super clé *de $r \Leftrightarrow |\pi_X(r)| = |r|$.*
$X \subseteq Y \Rightarrow |\pi_X(r)| \leq |\pi_Y(r)| \Rightarrow |\pi_Y(r)| = |r|$, i.e. Y est super clé.

De façon générale, notons que certains

prédicats ne vérifient pas cette propriété de monotonie (ou d'anti-monotonie) : c'est le cas notamment de certaines mesures statistiques [27].

4.3 Identifier la sortie

Il s'agit ici de caractériser quels motifs doivent-être donnés en sortie, en fonction de l'ensemble des motifs intéressants. Puisque les clés minimales d'une relation sont les super clés minimales par inclusion, et que $SCle(r)$ est clos par le haut, on a :

$$Cle(r) = \mathcal{B}d^{+}(SCle(r))$$

Nous verrons plus loin que cette caractérisation est importante pour guider l'analyste dans son choix de la stratégie d'exploration.

4.4 Exhiber une représentation ensembliste

De nombreux problèmes s'expriment naturellement de façon ensembliste, et pour eux cette étape est triviale. C'est le cas ici, puisque l'espace de recherche est lui-même un treillis des parties. La fonction de transformation est donc l'identité.

A notre connaissance, le seul problème pour lequel une transformation non triviale a été donnée est celui de la découverte des dépendances d'inclusion dans les bases de données (voir section 3.2) [25].

5 Choix de la stratégie d'exploration

5.1 Les algorithmes

Une fois le problème placé dans ce cadre, s'offrent à nous différentes stratégies d'exploration de l'espace de recherche. Actuellement, nous nous sommes focalisés sur deux types d'algorithmes issus de la découverte des motifs fréquents :

- un algorithme par niveau à la *Apriori* [1], qui effectue un parcours des éléments les plus généraux vers les éléments les plus spécifiques. Il permet de découvrir la théorie et les deux bordures.
- l'algorithme ABS [10] qui effectue une recherche par niveau dans une phase d'initialisation, avant d'alterner des va-et-vient entre les bordures [26]. Cet algorithme permet uniquement de trouver les deux bordures.

Initialement, ces algorithmes ne peuvent traiter que des prédicats anti-monotones. En effet, ils effectuent un parcours de "bas en haut" de l'espace de recherche (en totalité ou en partie), et exploitent la propriété d'anti-monotonie pour élaguer des éléments. Toutefois, Il est aussi possible d'exploiter ces stratégies avec des prédicats monotones, à condition d'accepter de ne trouver que les bordures de la théorie. Pour faire cela, il suffit d'inverser le prédicat monotone P, i.e. d'étudier $\neg P$ qui est anti-monotone. De plus, rappelons que la bordure positive (resp. la bordure négative) d'un prédicat monotone P est la bordure négative (resp. la bordure positive) de $\neg P$.

Exemple 1 *Pour découvrir la bordure positive des super clés, il faut donc étudier le prédicat "ne pas être une super clé", défini par $\neg P(X, d)$ est vrai si et seulement si $|\pi_X(r)| \neq |r|$, et rechercher la bordure négative liée à ce prédicat.*

Pour faire cela, nous avons le choix entre les deux types de parcours présentés précédemment.

Considérons la relation r (table 1).

La figure 1 montre le parcours par niveau effectué par l'algorithme Apriori. Les éléments foncés sont les éléments générés et testés par l'algorithme. Les chiffres en dessous sont les cardinalités des projections calculées par l'algorithme lors du test du prédicat. Les éléments

A	B	C	D	E
0	1	b	2	e
1	2	b	1	f
1	1	a	3	a
1	1	a	4	e
1	1	a	5	a
0	1	a	6	a
1	2	b	6	a

TAB. 1 – relation r

générés et testés sont tous "non super clés" à l'exception de $\{AD, CD, BD\}$*. La partie en clair correspond à l'espace de recherche qui n'a pas été exploré par l'algorithme. L'algorithme trouve la bordure positive, i.e.* $\{ABCE, DE\}$*, en quatre itérations. Les clés minimales sont donc* $\{AD, CD, BD\}$ *(la bordure négative).*

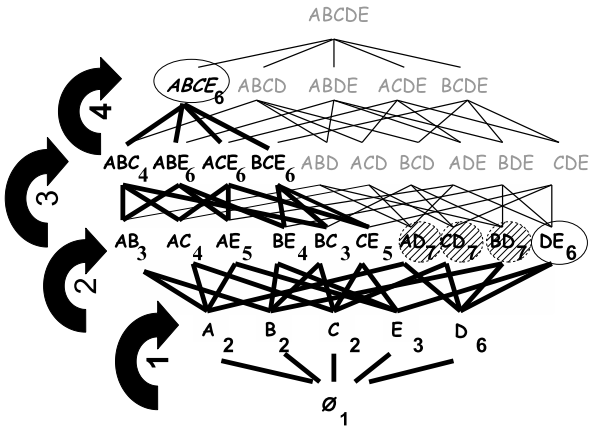

FIG. 1 – Exécution d'Apriori

La figure 2 montre le parcours effectué par ABS. Supposons que l'algorithme explore les deux premiers niveaux lors de sa phase d'initialisation. En s'appuyant sur les mauvais éléments découverts, ici $\{AD, CD, BD\}$*, l'algorithme effectue une dualisation est trouve directement la bordure positive (*$\{ABCE, DE\}$*).*

Dans notre contexte, l'intérêt de ces algorithmes est de pouvoir être utilisés pour résoudre d'autres problèmes que le problème pour lequel

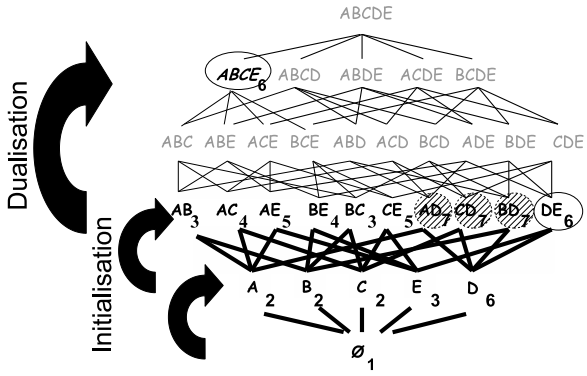

FIG. 2 – Exécution d'ABS

ils ont été conçus (celui des motifs fréquents). Par exemple, des algorithmes tels que $FP - growth$ [16] appliquent une stratégie d'extraction des motifs à partir des transactions. Cette approche est spécifique au problème des motifs fréquents, et ne peut pas être utilisée pour résoudre d'autres problèmes.

De plus, une grande partie des algorithmes de découverte des motifs fréquents fondent leur efficacité sur des méthodes optimisées de comptage du support. A l'opposé, l'objectif d'*Apriori* et d'*ABS* est de limiter le nombre de motifs testés.

5.2 Changement du sens de parcours des algorithmes

Un corrolaire intéressant de cette modélisation est qu'il est possible de changer le sens de parcours de l'espace de recherche pour chaque algorithme :

- Avec *Apriori*, l'espace de recherche peut être exploré de haut en bas.
- Avec *ABS*, l'initialisation peut commencer par le haut et alterner des dualisations entre les deux bordures comme précédemment.

Une approche "par le haut" permet notamment de trouver la théorie de prédicats monotones. Dans le cas d'un prédicat anti-monotone, il peut s'avérer pertinent en fonction de la nature des données d'utiliser une approche "par le

haut" : il faut alors aussi inverser le prédicat, et accepter de ne trouver que les bordures de la théorie.

Notons que pour l'implantation de ces deux variantes, il suffit de transformer les motifs en leur complémentaire au moment de tester le prédicat.

Exemple 2
Considérons une nouvelle relation s :

F	G	H	I	J
0	a	d	2	e
0	b	d	2	0
0	a	a	2	1
0	a	d	4	e
0	a	d	2	f

TAB. 2 – relation s

Le prédicat "être une super clé" est monotone. Il suffit donc de transformer chaque motif étudié par l'algorithme en son complément et d'appliquer le prédicat sur ce complément.

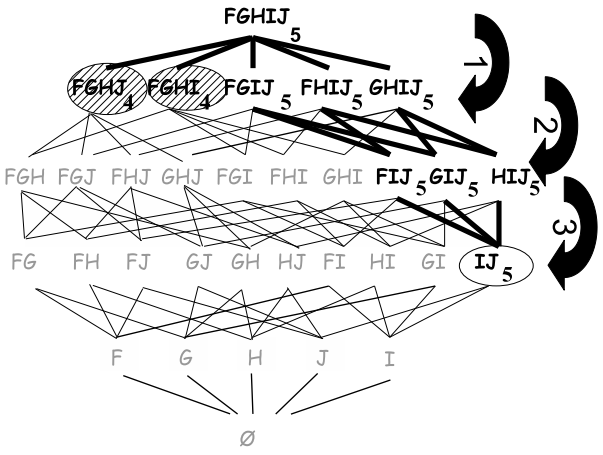

FIG. 3 – Exécution d'Apriori en changeant le sens de parcours

La figure 3 montre le parcours par niveau effectué par l'algorithme Apriori lorque le sens de parcours est inversé. Le parcours se fait de haut en bas à partir des super clés découvertes et en élaguant les motifs non super clés.

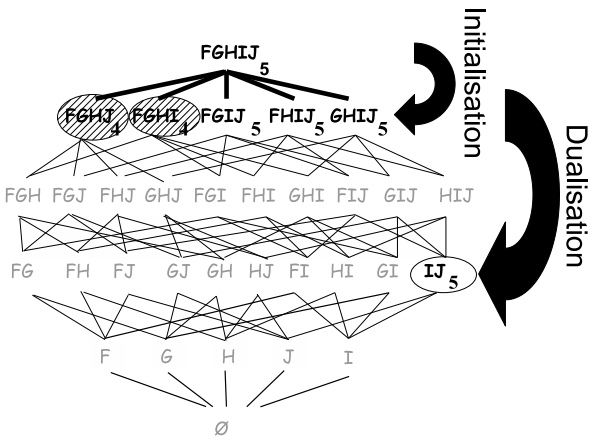

FIG. 4 – Exécution d'ABS en changeant le sens de parcours

La figure 4 montre le parcours effectué par ABS en changeant le sens de parcours et en explorant un niveau pour la phase d'initialisation.

5.3 Quel parcours choisir ?

La librairie présentée offre à l'utilisateur une grande liberté dans l'utilisation des algorithmes, ce qui lui permet de choisir la stratégie la plus adaptée à son problème. Le choix du mode de parcours de l'espace de recherche dépend en partie de la sortie que l'on souhaite obtenir, mais aussi des caractéristiques du problème étudié et/ou des données.
Par exemple, *ABS* sera particulièrement adapté à l'énumération des DI satisfaites dans une base de données. Une justification peut se résumer de la façon suivante : si toutes les généralisations de taille k d'une DI candidate i sont satisfaites, alors i a plus de chances d'être satisfaite lorsque k augmente. Ce résultat se justifie par une règle d'inférence des DF et des DI [26].
En revanche, différents travaux [3, 2] ont montré qu'*Apriori* était un algorithme hautement compétitif pour la découverte des motifs fréquents, dès lors que les données sont creuses

et que les solutions de "petite" taille.

Exemple 3 *Dans l'exemple 1, l'algorithme ABS teste moins de motifs que l'algorithme Apriori. Pour ce jeu de données, ABS devrait être plus efficace.*

Lorsque beaucoup de super clés sont de grande taille et peu de petite taille, comme c'est le cas dans l'exemple 2, changer le sens de parcours des algorithmes est une stratégie intéressante.

Notons qu'*ABS* est un algorithme qui adapte sa stratégie en fonction des données. Il effectue dans un premier temps un parcours par niveau de l'espace de recherche, puis change de stratégie lorsque celle-ci n'est plus appropriée et alterne des dualisations entre les deux bordures. Ce changement se fait dynamiquement en fonction des caractéristiques de l'espace de recherche déjà exploré. Ces caractéristiques sont issues des expérimentations réalisées dans [11], et peuvent être aussi appliquées à tout problème définis dans le cadre théorique précédemment introduit.

D'autres parcours génériques, par exemple en profondeur, peuvent bien sûr être considérés, chacun d'entre eux présentant des avantages suivant les configurations. Notre librairie est largement extensible ; nous verrons plus loin le modèle de conception utilisé pour les algorithmes implantés, pouvant être utilisés pour ajouter d'autres parcours.

6 Vers une implémentation générique

L'idée poursuivie dans ce travail – encore préliminaire en terme de développement – est de fournir une boîte à outils qui permette de développer rapidement un code efficace et robuste en utilisant tel ou tel algorithme préalablement choisi.

6.1 Aspects pratiques

La figure 5 représente le diagramme de classe d'un algorithme et des composants spécifiques au problème étudié, tels qu'ils sont implémentés au sein de la librairie. Ce diagramme de classe est directement issu du cadre théorique utilisé et des problèmes étudiés.

Les composants spécifiques au problème étudié sont devenus des paramètres de l'algorithme, qui est devenu une "boîte noire" pour l'utilisateur. Selon le problème étudié, jusqu'à cinq types de composants sont à redéfinir par l'utilisateur : les données en entrées de l'algorithme, la fonction d'initialisation du langage, le prédicat, les données en sortie de l'algorithme et la fonction de transformation des motifs du langage. Certains de ces composants doivent nécessairement être définis, tels que le prédicat et la fonction d'initialisation du langage, les autres dépendent du problème étudié.

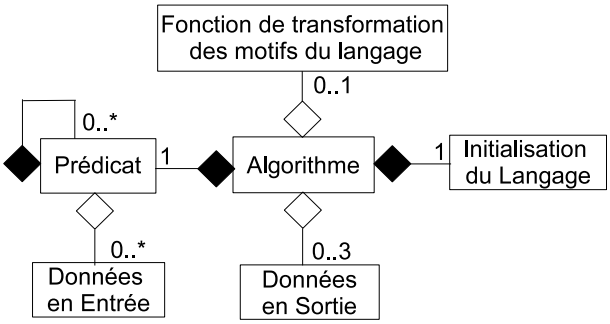

FIG. 5 – Diagramme de classe des différents types de composants manipulés dans la librairie

La suite décrit le fonctionnement de notre librairie en présentant les composants à redéfinir (i.e. à implémenter ou à réutiliser) pour chaque problème et comment utiliser les algorithmes implémentés afin de résoudre un nouveau problème.

6.1.1 Les composants à redéfinir

Les données en entrée de l'algorithme Lorsque le prédicat utilise des données pour déterminer les motifs "intéressants", l'utilisateur doit redéfinir le composant "Données en Entrée". Comme le montre la figure 5, ce composant est uniquement utilisé dans le test du prédicat, il est totalement indépendant de l'algorithme. L'utilisateur peut donc développer de nouveaux composants, utiliser ceux de la librairie, ou ceux de la STL, sans aucune restriction.

Le coût et le nombre des entrées/sorties est un facteur prédominant dans les performances de tout algorithme de fouille de données. Les principales alternatives pour accéder aux données sont :

- effectuer des requêtes SQL. Les données sont stockées dans un SGBD, et accédées par des requêtes SQL.
- effectuer des tests en mémoire vive. Les données sont chargées en mémoire vive dans des structures de données adaptées. L'accès se fait alors directement sur ces structures de données.

L'avantage de la première approche est d'interagir directement avec un SGBD et de limiter l'espace mémoire occupé alors que la deuxième approche favorise plus les performances.

L'initialisation du langage Ce composant doit être nécessairement développé et permet de définir "l'alphabet" du langage, i.e. les motifs correspondant aux singletons dans la représentation ensembliste.

Exemple 4 *Pour le problème de la découverte des clés minimales, cette phase correspond à la lecture du schéma de la relation.*

Le prédicat L'utilisateur doit ensuite implémenter le composant du prédicat (figure 5). Ce composant est une fonction (ou un foncteur) qui doit prendre en paramètre un motif et retourner vrai si celui-ci respecte le prédicat. Le prédicat peut aussi prendre en paramètre plusieurs données en entrée. Par exemple, dans le cadre de la découverte des DI entre deux relations, le prédicat prendra en paramètre les deux relations concernées.

Dans certains cas, les motifs sont associés à une mesure d'intérêt tel que le support pour les motifs fréquents. Le prédicat peut être alors utilisé pour calculer cette mesure et l'associer aux motifs.

Finalement, notons qu'il est possible de définir des prédicats issus de la conjonction de plusieurs prédicats.

Les données en sortie de l'algorithme L'utilisateur devra finalement développer ou utiliser des classes pour stocker et/ou enregistrer les solutions découvertes par l'algorithme (la théorie, la bordure positive et/ou la bordure négative). Ces classes peuvent être utilisées pour stocker les solutions en mémoire, pour les enregistrer dans un fichier dans un format donné ou sur un autre support. La définition de ce composant permet également d'effectuer un éventuel post-traitement sur la sortie ; filtrage de certains éléments, application d'un critère de maximalité ou d'un ordre de sortie particulier.

Quelques problèmes déjà implantés Dans sa version actuelle, notre librairie contient des composants pour la découverte des motifs fréquents et la découverte des clés minimales d'une relation, qui constituent une boîte à outils à la disposition de l'analyste :

- Des composants d'accès à des données stockées dans des fichiers : base de données de transactions au format de FIMI [3, 2], ou données tabulaires au format CSV d'Excel.
- Des structures de données de type "*trie*", particulièrement adaptées au comptage du support et factorisant l'espace mémoire occupé.
- Les prédicats "être une super clé" et "être fréquent".

6.1.2 Appel d'un algorithme

Un algorithme est un objet que l'on instancie en lui passant en paramètre des instances des composants nécessaires.

Pour inverser le sens de parcours et adopter une stratégie "par le haut", il suffit de passer en paramètre de l'algorithme la fonction, déjà implémentée dans la librairie, $f(X) = E - X$ avec E l'ensemble des éléments de taille 1 du langage. Cette fonction est ensuite utilisée pour transformer tous les motifs testés par rapport au prédicat en leur complémentaire.

Cette simple opération a pour conséquence d'inverser le sens de parcours de l'algorithme. Par conséquent, il est important de noter que ce changement de sens de parcours ne modifie pas la complexité de l'agorithme, ni son efficacité.

En outre, si besoin est, la librairie intègre un composant permettant d'inverser simplement un prédicat.

6.2 Performances-Optimisations

Les performances des composants utilisateur dépendent directement de la façon dont ils ont été implémentés. Toutefois, un certain nombre de composants sont disponibles afin de faciliter l'obtention d'une implémentation efficace. Par exemple, la librairie comprend deux structures de "*trie*" : la première favorise la compression des données, alors que la deuxième favorise les temps d'accès aux données.

Par ailleurs, le développement d'algorithmes modulaires et génériques implique nécessairement un surcoût à l'exécution par rapport à des solutions dédiées, essentiellement dû à des passages de paramètres supplémentaires, et à l'utilisation d'itérateurs.

Un certain nombre de techniques d'optimisations dédiées peuvent en outre influer fortement sur les performances. La découverte des motifs fréquents est un exemple typique, où tous les algorithmes utilisent des techniques astucieuses de comptage.

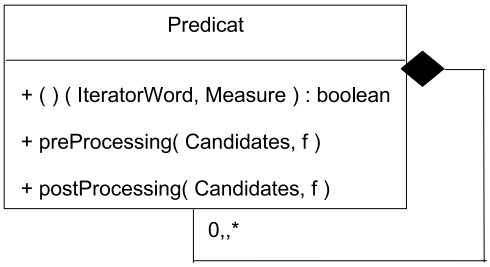

FIG. 6 – Prototype du prédicat avec les optimisations

En général ces optimisations sont liées au test du prédicat, et peuvent être perçues comme des pré-traitements ou des post-traitements du prédicat. Pour cette raison, deux méthodes ont été ajoutées au prédicat (figure 6) : "preProcessing" et "postProcessing". Ces deux méthodes sont exécutées dans les algorithmes avant et après la phase de test des candidats. Elles prennent en paramètres l'ensemble des candidats, ainsi que la fonction de transformation des motifs du langage. Elles peuvent être utilisées pour optimiser l'implémentation.

Par exemple, considérons la découverte des motifs fréquents :

- la méthode "preProcessing" peut par exemple servir à mettre à jour le support des candidats avant d'effectuer le test du prédicat. Ainsi il est possible de faire un parcours de la base de données à chaque itération de l'algorithme au lieu d'un pour chaque motif testé (comme cela est fait dans l'adaptation à ce cadre de l'algorithme *Apriori* [24]).
- la méthode "postProcessing" peut être utilisée pour reconstruire la base de données en supprimant des articles ou des transactions qui ne sont plus utilisés.

Il est également possible d'améliorer considérablement les performances des implémentations en tirant partie des spécificités

de certaines structures de données, comme les "*tries*" dans le cas des motifs fréquents. Comme le montre la figure 7, il est possible de développer au sein d'une même classe plusieurs versions d'une même méthode : une version générique et des versions spécialisées. La méthode générique est automatiquement utilisée si aucune méthode spécialisée ne correspond à la structure de données traitée.

```
Class isFrequent{

    ...

    template< class Data, class Candidates>
    count( Data , Candidates ){ ... }


    template< class Candidates>
    count( TrieData , Candidates ){ ... }

};
```

FIG. 7 – Exemple d'implémentation du prédicat "être fréquent" contenant une méthode générique pour le comptage du support et une spécifique lorsque les données sont stockées dans un trie.

6.3 Expérimentations

L'évaluation objective du temps de développement est très difficile. Le seul recul que nous avons par rapport à cet aspect provient de nos propres travaux [11, 18]. De manière générale, nous avons constaté que l'adaptation des implémentations existantes était extrêmement fastidieuse. A titre d'indication, l'utilisation de notre librairie pour résoudre le problème de la découverte des clés minimales d'une relation a mis moins d'une journée de travail.

Dans la suite de cette section, nous allons présenter quelques expérimentations réalisées à partir de notre librairie pour la découverte des motifs fréquents. Les composants nécessaires à la résolution de ce problème ont été implémentés, avec une version optimisée du test du prédicat. Les expérimentations on été faites avec l'algorithme *Apriori* sur des jeux de données de FIMI [3, 2].

Nous avons comparé l'efficacité de notre programme avec deux autres implémentations d'*Apriori* spécialement dédiées à ce problème : l'implémentation de B. Goethals [12] et l'implémentation de C. Borgelt [6]. La première est une implémentation classique de l'algorithme *Apriori*, la deuxième est la meilleure implémentation d'*Apriori* connue à ce jour. Elle est développée en C et est fortement optimisée. Soulignons que l'implémentation développée à partir de la librairie ne contient pas toutes les optimisations de ces deux autres programmes.

L'objectif de ces expérimentations est de montrer qu'une implémentation issue de la librairie permet le passage à l'échelle, et reste comparable aux meilleures implémentations, tout en étant générique et modulaire.

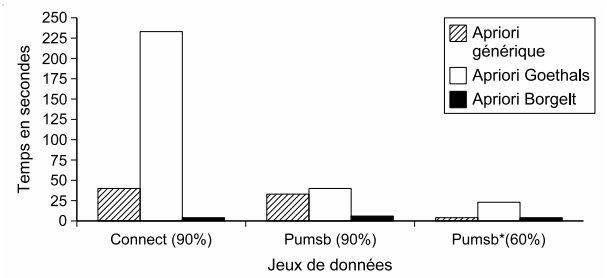

FIG. 8 – Expérimentation de trois implémentations d'Apriori sur troix jeux de données.

La figure 8 montre les temps d'exécutions obtenus pour des seuils de support donnés (entre parenthèses sur la figure) pour les jeux de données *Connect*, *Pumsb* et *Pumsb∗*. L'implémentation issue de la librairie est celle notée "Apriori générique". Les résultats obtenus montrent que les performances

d'"Apriori générique" se situent entre ceux des implémentations de B. Goethals et de C. Borgelt. Ces résultats sont donc très encourageant au regard de la rapidité et la simplicité d'obtention d'un programme opérationnel.

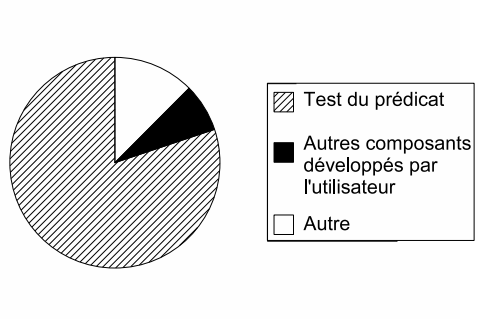

FIG. 9 – Répartition du temps d'exécution d'"Apriori générique" pour Connect (90%).

De plus lors des différentes expérimentations réalisées, il est apparu que la majeure partie du temps d'exécution était liée à des composants développés (ou réutilisés) par l'utilisateur. La figure 9 montre la répartition du temps d'exécution d'*Apriori* développé à partir de la librairie pour le jeux de données *Connect* testé précédemment. Par conséquent une implémentation plus optimisée de ces composants permettrait d'améliorer considérablement les performances globales de l'implémentation pour ce problème.

7 Travaux connexes

7.1 Implémentations des algorithmes de fouille de données

D'autres travaux proposent des outils de fouille de données et d'apprentissage libres de droits, sous forme de groupes d'algorithmes, les plus célèbres étant certainement *Weka* [30] et *Illimine* [15].

Toutefois, ces projets ne font que regrouper des implémentations d'algorithmes dédiées et optimisées pour la résolution de problèmes précis. Les codes sources ne sont pas toujours disponibles, et lorsqu'ils le sont leur modification s'avère difficile voir impossible de part la complexité de l'implémentation. De plus, ces implémentations n'étant pas prévues initialement pour être modifiées, aucune méthode ou documentation n'est proposée afin de guider l'utilisateur dans ses modifications.

7.2 La librairie DMTL

A notre connaissance, un seul travail cherche à simplifier le rôle des programmeurs, et a donné lieu à la librairie DMTL (Data Mining Template Library) [17]. Elle permet à l'utilisateur d'adapter les algorithmes implémentés par l'équipe de M. Zaki afin de résoudre des problèmes de découverte de motifs fréquents.

DMTL est une collection d'algorithmes et de structures de données. Les structures de données génériques définies permettent de traiter des motifs variés, et permettent la conception d'algorithmes génériques d'extraction de motifs fréquents. La librairie permet d'accéder à différents types de base de données en mémoire ou sur le disque, et de les gérer de manière horizontale ou verticale.

Dans leur librairie, les auteurs représentent les différents types de motifs sous forme de graphes. La figure 10 illustre des motifs tels que les ensembles, les séquences, les arbres et les graphes. Ainsi, les ensembles d'articles sont par exemple des graphes ayant pour noeuds les articles et aucune arête.

Concrétement, une même structure de données est utilisée pour représenter tous les motifs sous forme de graphe, et chaque type de motifs est différencié par un ensemble de propriétés :

– propriétés sur la structure des motifs. Elles permettent de définir le type de graphe représentant les motifs, i.e. les propriétés

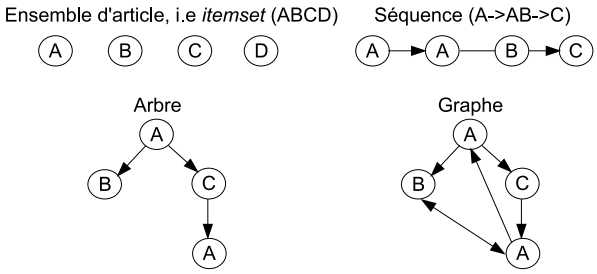

FIG. 10 – Représentations des motifs dans la librairie DMTL [31]

sur le arêtes et les noeuds.
- propriétés utilisées pour l'extraction. Ces propiétés sont en fait des méthodes permettant par exemple de définir comment générer un nouveau motif à partir de deux motifs déjà étudiés, ou comment compter le support des motifs.

Dans ce contexte, l'ajout d'un nouveau type de motif à la librairie correspond simplement à l'ajout de nouvelles propriétés.

Les auteurs disent utiliser le cadre théorique de l'*analyse formelle de concepts* pour représenter l'association entre les types de motifs et leurs propriétés. Soient l'ensemble des motifs M pouvant être traités par la librairie, l'ensemble G des toutes les propriétés des motifs, et une relation I indiquant si un type de motif est conforme à une propriété, alors (G, M, I) est un contexte. Si $A \subseteq M$ est une collection de différents types de motifs, et $B \subseteq G$ un ensemble de propriétés communes à tous les motifs de A, alors (A, B) est un concept formel du contexte (G, M, I). La hiérarchie des concepts (un treillis) issue de cette représentation est utilisée par les auteurs pour développer des algorithmes pouvant traiter différents types de motifs.

Pour extraire un nouveau type de motif par un des algorithmes implémentés dans la librairie, l'utilisateur doit redéfinir différentes méthodes dont la génération des candidats et le comptage du support. Notons que, dans le cas des ensembles, la méthode de construction des nouveaux motifs ne génère pas de doublons. Par exemple, considérons la méthode utilisée dans l'algorithme *Apriori* pour construire des motifs de taille $k+1$ à partir des motifs de taille k. Par cette méthode, il est impossible de générer à une même itération plusieurs fois les mêmes motifs, comme par exemple ABC et CBA. Toutefois, ce n'est pas nécessairement le cas de motifs plus complexes tels que les graphes. Dans ces cas, l'utilisateur doit donc développer en plus une méthode vérifiant que chaque motif candidat est généré exactement une fois.

La librairie contient actuellement deux types de stratégies d'exploration de l'espace de recherche : une stratégie effectuant un parcours en largeur (ou par niveau) de type *Apriori* et une stratégie effectuant un parcours en profondeur. Ces stratégies permettent de trouver l'ensemble de motifs fréquents de la base de données étudiée.

Quoique similaire dans l'esprit, notre proposition se démarque de DMTL en différents points :
- le cadre théorique est différent.
- tout prédicat monotone ou anti-monotone peut être pris en compte alors que DMTL se focalise sur la fréquence.
- plus de stratégies de parcours disponibles.
- possibilité de découvrir la théorie ainsi que les bordures positive et négative.

Finalement, notons que les stratégies définies dans DTML explorent nécessairement tous les motifs fréquents. Or cet espace de recherche peut être très important. La majeure partie des algorithmes d'extractions de motifs fréquents font des "sauts" dans l'espace de recherche pour découvrir des motifs fréquents maximaux, et ainsi élaguer de grandes parties de l'espace de recherche. DMTL ne contient pas ce genre de stratégies.

Néanmoins, notre librairie est dans sa forme actuelle moins achevée que DMTL sur certains aspects tels que l'accès aux données.

8 Conclusion

Nous avons présenté dans cet article un travail en cours permettant le prototypage rapide de programmes de fouilles de données. Notre approche est basée sur un cadre théorique [24] qui définit les problèmes de découverte de motifs intéressants dans les données. Notre travail concerne de très nombreux problèmes de fouille de données mais avec un point de vue nouveau : celui de la généricité et du temps de développement nécessaire pour obtenir des programmes fiables, robustes et passant à l'échelle.

Les perspectives de ce travail sont riches et nombreuses. Tout d'abord d'un point de vue théorique, nous pouvons essayer de relaxer certaines contraintes afin de pouvoir envisager le traitement de plus de problèmes, comme la découverte de sous-séquences fréquentes dans une séquence. Ensuite, on peut imaginer le développement d'une approche plus déclarative de ces problèmes de fouilles de données à l'instar de l'optimisation de requêtes en base de données. En définissant un langage de requêtes et un modèle de coût, il semble envisageable de construire un optimiseur de requêtes qui choisirait la "meilleure" implémentation, libérant le développeur de cette tâche difficile.

Références

[1] Rakesh Agrawal and Ramakrishnan Srikant. Fast algorithms for mining association rules in large databases. In Jorge B. Bocca, Matthias Jarke, and Carlo Zaniolo, editors, *VLDB*, pages 487–499. Morgan Kaufmann, 1994.

[2] Roberto J. Bayardo Jr., Bart Goethals, and Mohammed Javeed Zaki, editors. *FIMI '04, Proceedings of the IEEE ICDM Workshop on Frequent Itemset Mining Implementations, Brighton, UK, November 1, 2004*, volume 126 of *CEUR Workshop Proceedings*. CEUR-WS.org, 2004.

[3] Roberto J. Bayardo Jr. and Mohammed Javeed Zaki, editors. *FIMI '03, Proceedings of the IEEE ICDM Workshop on Frequent Itemset Mining Implementations, Melbourne, Florida, USA, November 19, 2003*, volume 90 of *CEUR Workshop Proceedings*. CEUR-WS.org, 2003.

[4] Catriel Beeri, Martin Dowd, Ronald Fagin, and Richard Statman. On the structure of armstrong relations for functional dependencies. *J. ACM*, 31(1) :30–46, 1984.

[5] Christian Borgelt. Efficient implementations of Apriori and Eclat. In Bayardo Jr. and Zaki [3].

[6] Christian Borgelt. Recursion pruning for the apriori algorithm. In Bayardo Jr. et al. [2].

[7] Jean-François Boulicaut, Artur Bykowski, and Christophe Rigotti. Free-sets : A condensed representation of boolean data for the approximation of frequency queries. *Data Min. Knowl. Discov.*, 7(1) :5–22, 2003.

[8] Toon Calders and Bart Goethals. Minimal k-free representations of frequent sets. In Nada Lavrac, Dragan Gamberger, Hendrik Blockeel, and Ljupco Todorovski, editors, *PKDD*, volume 2838 of *Lecture Notes in Computer Science*, pages 71–82. Springer, 2003.

[9] Alain Casali, Rosine Cicchetti, and Lotfi Lakhal. Essential patterns : A perfect cover of frequent patterns. In A. Min Tjoa and Juan Trujillo, editors, *DaWaK*, volume 3589 of *Lecture Notes in Computer Science*, pages 428–437. Springer, 2005.

[10] Frédéric Flouvat, Fabien De Marchi, and Jean-Marc Petit. ABS : Adaptive Borders Search of frequent itemsets. In Bayardo Jr. et al. [2].

[11] Frédéric Flouvat, Fabien De Marchi, and Jean-Marc Petit. A thorough experimental study of datasets for frequent itemsets. In *ICDM*, pages 162–169. IEEE Computer Society, 2005.

[12] Bart Goethals. Apriori implementation. University of Antwerp. http ://www.adrem.ua.ac.be/˜goethals/, 2005.

[13] Bart Goethals, Siegfried Nijssen, and Mohammed Javeed Zaki. Open source data mining : workshop report. *SIGKDD Explorations*, 7(2) :143–144, 2005.

[14] Dimitrios Gunopulos, Roni Khardon, Heikki Mannila, Sanjeev Saluja, Hannu Toivonen, and Ram Sewak Sharm. Discovering all most specific sentences. *ACM Trans. Database Syst.*, 28(2) :140–174, 2003.

[15] Jiawei Han and Data Mining Group. IlliMine project. University of Illinois Urbana-Champaign Database and Information Systems Laboratory. http ://illimine.cs.uiuc.edu/, 2005.

[16] Jiawei Han, Jian Pei, Yiwen Yin, and Runying Mao. Mining frequent patterns without candidate generation : A frequent-pattern tree approach. *Data Min. Knowl. Discov.*, 8(1) :53–87, 2004.

[17] Mohammad Hasan, Vineet Chaoji, Saeed Salem, Nagender Parimi, and Mohammed Zaki. DMTL : A generic data mining template library. In *Workshop on Library-Centric Software Design (LCSD'05), with Object-Oriented Programming, Systems, Languages and Applications (OOPSLA'05) conference, San Diego, California*, 2005.

[18] Hélène Jaudoin and Frédéric Flouvat. Techniques de fouille de données pour la réécriture de requêtes en présence de contraintes de valeurs. In Gilbert Ritschard and Chabane Djeraba, editors, *EGC*, volume RNTI-E-6 of *Revue des Nouvelles Technologies de l'Information*, pages 77–88. Cépaduès-Éditions, 2006.

[19] Hélène Jaudoin, Jean-Marc Petit, Christophe Rey, Michel Schneider, and Farouk Toumani. Query rewriting using views in presence of value constraints. In Ian Horrocks, Ulrike Sattler, and Frank Wolter, editors, *Description Logics*, volume 147 of *CEUR Workshop Proceedings*. CEUR-WS.org, 2005.

[20] Martti Kantola, Heikki Mannila, Kari-Jouko Räihä, and Harri Siirtola. Discovering functional and inclusion dependencies in relational databases. *International Journal of Intelligent Systems*, 7 :591–607, 1992.

[21] Andreas Koeller and Elke A. Rundensteiner. Discovery of high-dimensional. In *ICDE*, pages 683–685, 2003.

[22] Stéphane Lopes, Jean-Marc Petit, and Lotfi Lakhal. Efficient discovery of functional dependencies and armstrong relations. In Carlo Zaniolo, Peter C. Lockemann, Marc H. Scholl, and Torsten Grust, editors, *EDBT*, volume 1777 of *Lecture Notes in Computer Science*, pages 350–364. Springer, 2000.

[23] Heikki Mannila and Hannu Toivonen. Multiple uses of frequent sets and condensed representations (extended abstract). In *KDD*, pages 189–194, 1996.

[24] Heikki Mannila and Hannu Toivonen. Levelwise search and borders of theories in knowledge discovery. *Data Min. Knowl. Discov.*, 1(3) :241–258, 1997.

[25] Fabien De Marchi, Frédéric Flouvat, and Jean-Marc Petit. Adaptive strategies for mining the positive border of interesting patterns : Application to inclusion dependencies in databases. In Jean-François Boulicaut, Luc De Raedt, and Heikki Mannila,

editors, *Constraint-Based Mining and Inductive Databases*, volume 3848 of *Lecture Notes in Computer Science*, pages 81–101. Springer, 2005.

[26] Fabien De Marchi and Jean-Marc Petit. Zigzag : a new algorithm for mining large inclusion dependencies in database. In *ICDM*, pages 27–34. IEEE Computer Society, 2003.

[27] Shinichi Morishita and Jun Sese. Traversing itemset lattice with statistical metric pruning. In *PODS*, pages 226–236. ACM, 2000.

[28] Noel Novelli and Rosine Cicchetti. Functional and embedded dependency inference : a data mining point of view. *Inf. Syst.*, 26(7) :477–506, 2001.

[29] Nicolas Pasquier, Yves Bastide, Rafik Taouil, and Lotfi Lakhal. Discovering frequent closed itemsets for association rules. In Catriel Beeri and Peter Buneman, editors, *ICDT*, volume 1540 of *Lecture Notes in Computer Science*, pages 398–416. Springer, 1999.

[30] Ian H. Witten and Eibe Frank. *Data Mining : Practical Machine Learning Tools and Techniques with Java Implementations.* Morgan Kaufmann, 1999.

[31] Mohammed Javeed Zaki, Nagender Parimi, Nilanjana De, Feng Gao, Benjarath Phoophakdee, Joe Urban, Vineet Chaoji, Mohammad Al Hasan, and Saeed Salem. Towards generic pattern mining. In Bernhard Ganter and Robert Godin, editors, *ICFCA*, volume 3403 of *Lecture Notes in Computer Science*, pages 1–20. Springer, 2005.