UNIVERSITÉ DU QUÉBEC À MONTRÉAL

MOTIFS SÉQUENTIELS DE DONNÉES LIÉES

MÉMOIRE

PRÉSENTÉ

COMME EXIGENCE PARTIELLE

DE LA MAÎTRISE EN INFORMATIQUE

PAR

TOMAS MARTIN

OCTOBRE 2017

UNIVERSITÉ DU QUÉBEC À MONTRÉAL
Service des bibliothèques

Avertissement

La diffusion de ce mémoire se fait dans le respect des droits de son auteur, qui a signé le formulaire *Autorisation de reproduire et de diffuser un travail de recherche de cycles supérieurs* (SDU-522 – Rév.07-2011). Cette autorisation stipule que «conformément à l'article 11 du Règlement no 8 des études de cycles supérieurs, [l'auteur] concède à l'Université du Québec à Montréal une licence non exclusive d'utilisation et de publication de la totalité ou d'une partie importante de [son] travail de recherche pour des fins pédagogiques et non commerciales. Plus précisément, [l'auteur] autorise l'Université du Québec à Montréal à reproduire, diffuser, prêter, distribuer ou vendre des copies de [son] travail de recherche à des fins non commerciales sur quelque support que ce soit, y compris l'Internet. Cette licence et cette autorisation n'entraînent pas une renonciation de [la] part [de l'auteur] à [ses] droits moraux ni à [ses] droits de propriété intellectuelle. Sauf entente contraire, [l'auteur] conserve la liberté de diffuser et de commercialiser ou non ce travail dont [il] possède un exemplaire.»

TABLE DES MATIÈRES

LISTE DES FIGURES

LEXIQUE

Item Élément i tiré d'un ensemble de n d'éléments distincts, l'univers des items $\mathscr{I} = \{i_1, i_2, \ldots i_n\}$.

Itemset Ensemble composé d'éléments dans l'univers des items. Tous les itemsets forment l'univers des itemsets $\mathscr{I}+$.

Séquence Ensemble ordonné d'itemsets. Ces itemsets peuvent être singletons ou de plus grande taille. Toutes ces séquences forment l'univers des séquences $\mathscr{S}$.

Motif Ensemble à rechercher dans la base. Peut représenter un itemset, une séquence ou un autre type plus complexe.

Support Comptage du nombre de fois qu'un motif est présent dans les séquences de la base. Peut être un entier ou un pourcentage.

Fréquence Un motif est dit fréquent si son support est supérieur ou égal à un seul défini. Par opposition, un motif est dit infréquent s'il n'est pas déterminé comme fréquent.

Candidat Motif dont la fréquence est à évaluer.

Fouille verticale Fouille à partir d'une représentation de la base sous la forme d'un index inversé.

Appariement Action de rechercher un motif dans une séquence. Ces actions dépendent du type de motif.

RÉSUMÉ

Intégrer des connaissances du domaine dans le processus de fouille de motifs séquentiels fréquents (Mabroukeh et Ezeife, 2010; Mooney et Roddick, 2013) et ce, avec des temps d'exécution relativement raisonnables, est un processus extrêmement couteux en temps de calcul.

Une telle augmentation du processus de fouille, proposée originellement dans (Adda *et al.*, 2006; Adda *et al.*, 2007) permet d'enrichir l'espace des motifs avec des données liées et ainsi obtenir des motifs séquentiels beaucoup plus intéressants pour l'utilisateur. En effet, ces motifs tirent parti à la fois d'une abstraction au niveau des items (catégorisation) ainsi que de l'ajout de propriétés hiérarchisées entre ces mêmes items (triplets RDF). Ceci permettant de mieux saisir le contexte supportant ledit motif, par rapport à un motif séquentiel classique ou une règle d'association.

L'inconvénient majeur de cette augmentation est que les espaces de recherche augmentent considérablement rendant les temps d'exécution prohibitifs en pratique. Des solutions algorithmiques efficaces, inspirées par les meilleures approches existantes de fouille de motifs séquentiels sont à développer.

Dans ce travail, nous proposons deux variantes de la méthode de fouille de motifs séquentiels de données liées qui augmentent sensiblement son efficacité. La seconde méthode — qui est la contribution principale de notre travail — exploite une nouvelle technique d'appariement à l'aide d'un codage vertical ainsi qu'une nouvelle stratégie de parcours de l'espace des motifs tirant profit de mécanismes de pré-élagage.

Mots clés Fouille de données, Données liées, Motifs séquentiels fréquents, Web sémantique, Ontologies

INTRODUCTION

La fouille de motifs séquentiels ne dispose pas encore d'un algorithme reconnu comme optimal. En effet, beaucoup de solutions relativement efficaces existent au sein de la littérature mais aucune n'est universellement adoptée (Mabroukeh et Ezeife, 2010; Mooney et Roddick, 2013). Même si en théorie un algorithme de fouille de motifs séquentiels est en mesure d'extraire une grande partie de l'information latente dans les données, en pratique, la nature fortement combinatoire du problème fait qu'il est difficile d'en analyser de grandes quantités. De plus, comparativement à la fouille d'itemsets fréquents (Aggarwal et Han, 2014), à ressources égales et jeu de données égal, le processus de fouille est beaucoup plus long pour la fouille de motifs séquentiels fréquents.

En outre, le nombre de motifs extraits est considérablement trop important pour pouvoir être analysé par des humains qui doivent eux posséder des connaissances du domaine analysé (experts) sous peine de ne pas pouvoir interpréter les motifs extraits. Généralement, pour être en mesure de trouver des motifs utiles, il est nécessaire de fouiller avec des supports minimaux extrêmement bas et donc d'augmenter encore plus le nombre de motifs produits et le temps nécessaire pour tous les produire. Ainsi, à l'heure actuelle nous sommes donc loin de la métaphore selon laquelle les algorithmes de fouille de motifs sont capables d'extraire automatiquement les diamants d'information cachés dans les masses de données inutiles. Autrement dit, en 2016, les défis auxquels font face la fouille de motifs séquentiels et d'itemsets sont à la fois d'ordre technique (comment cherche t'on ?) mais aussi d'ordre conceptuel (que doit on chercher ?).

Dans l'idéal, il serait par conséquent extrêmement intéressant de disposer d'une technique qui à la fois permettrait de rendre l'information produite plus facilement interpré-

table par tout un chacun, tout en permettant d'augmenter les seuils de support afin de réduire le nombre de motifs produits (meilleure synthèse) et d'accélérer le calcul des résultats (moins de ressources).

Notre travail s'inscrit dans cette ligne en faisant appel à une classe de motifs relativement peu explorée, les motifs séquentiels de données liées initialement proposes dans (Adda *et al.*, 2006). Dans ce mémoire notre objectif est donc de proposer un nouvel algorithme de fouille de motifs séquentiels efficace exploitant des données liées. Au lieu de construire des motifs à l'aide des données brutes, ces motifs sont construits en combinant des objets issus directement d'une ontologie. En effet, ces motifs intègrent des liens labellisés entre les objets les composant et exploitent deux hiérarchies distinctes, une pour les objets et une pour les liens labellisés.

Notre travail approfondit les idées proposées dans (Adda *et al.*, 2007; Adda *et al.*, 2006; Adda *et al.*, 2010; Adda, 2008) visant à extraire des motifs séquentiels composés de classes et propriétés provenant d'une ontologie plutôt que des motifs séquentiels classiques composés des items présents dans les données brutes telles que des logs de serveurs web ou des historiques de commandes d'achats d'utilisateurs. Même si cette idée est très prometteuse, les performances d'un tel outil ne sont pas encore au rendez-vous, en majeure partie à cause de la taille de ce nouvel espace des motifs. À notre connaissance, mis à part xPMiner (Adda *et al.*, 2007), l'ancêtre et l'inspiration majeure de notre approche, il n'existe pas encore aujourd'hui de concurrent direct à nos deux approches.

Ce mémoire va s'articuler autour de quatre grandes parties. Dans un premier temps, nous décrirons nos nouveaux motifs séquentiels de données liées et justifierons du bien fondé de ceux-ci.

Dans une seconde partie, nous proposerons un état de l'art des algorithmes de fouilles de motifs séquentiels à travers une cartographie exhaustive des publications sur le sujet

afin d'être en mesure d'identifier quelles sont les approches les plus efficaces existantes aujourd'hui. Dans la partie suivante, nous proposerons une vue d'ensemble sur les approches intégrant des connaissances externes au sein du processus de fouille.

Dans la quatrième partie, nous présenterons notre travail de conception d'un nouveau mécanisme d'appariement de motifs séquentiels de données liées avec les séquences utilisateurs. Nous en présenterons les avantages et les inconvénients.

Dans la dernière partie, nous effectuerons une comparaison des performances des approches présentées puis nous mettrons à profit les techniques les plus efficaces dans une nouvelle approche algorithmique en vue de décupler les performances de notre approche.

Pour conclure notre travail, nous dresserons un bilan des avancées de nos travaux en tenant compte des limitations ainsi que des éventuels travaux futurs.

CHAPITRE I

PROBLÉMATIQUE

1.1 Formalisation des motifs recherchés

Dans cette partie, nous allons définir la fouille d'itemsets fréquents et de séquences fréquentes qui constituent les bases sur lesquelles nous allons ensuite définir les motifs séquentiels de données liées.

Tout d'abord, l'univers des items $\mathscr{I} = \{i_1, i_2, \ldots i_n\}$ un ensemble de n éléments distincts. Il peut s'agit de chaines de caractères ou directement de nombres entiers. En règle générale, ces items sont pré-traités dans une fonction bijective qui crée une correspondance entre chaque item d'origine et un nombre entier. En outre, pour une plus grande efficacité de calcul ces nombres sont supposément contigus.

L'univers des itemsets $\mathscr{I}+$ est un ordre partiel construit à partir de toutes les combinaisons des éléments dans $\mathscr{I}$, $\mathscr{I}+ = \bigcup_{k=1}^{z} C_k^{\mathscr{I}}$. Remarquons que si $z = 1$ alors $\mathscr{I} = \mathscr{I}+$ et il s'agit de l'ensemble des itemsets singletons. Dans le cas contraire, l'univers des itemsets $\mathscr{I}+$ constitue l'ensemble de toutes les combinaisons possibles de taille entre 1 et z des éléments dans l'univers des items $\mathscr{I}$. Autrement dit, $\mathscr{I}+ = 2^{\mathscr{I}}$.

Une séquence s est un ensemble ordonné d'itemsets telle que $s = \langle \alpha_1, \alpha_2, \ldots, \alpha_m \rangle$ avec $\alpha_x \in \mathscr{I}+, x \in [1, m]$. Dans s, si l'itemset α_i apparait avant l'itemset α_j c'est à dire si $i < j$, alors on considère que $\alpha_i < \alpha_j$. Sachant que ces itemsets peuvent se

répéter un nombre illimité fois dans une séquence, l'univers des séquences $\mathscr{S}$ devient virtuellement de taille illimitée. Cependant, nous pouvons fixer une taille de maximale pour les séquences et ainsi rendre énumérative cet espace des séquences. Si nous fixons une taille k alors le nombre de séquences possibles pour cette taille $\mathscr{S}_k = \mathscr{I}+^k$ et par conséquent, l'ensemble de toutes les séquences $\mathscr{S} = \bigcup_{k=1}^{k_{max}} \mathscr{I}+^k$. Il va alors de de soi que $\mathscr{S} \supseteq \mathscr{I}+$ et que plus la limite k augmente plus la différence $\mathscr{S} - \mathscr{I}+$ augmente aussi. Intuitivement, il est aisé de comprendre la difficulté d'énumérer et de calculer l'ensemble des motifs dans $\mathscr{S}$ par rapport à $\mathscr{I}+$. Surtout si l'on admet que cette même énumération dans $\mathscr{I}+$ est déjà un problème complexe. Le cas particulier $\mathscr{S} = \mathscr{I}+$ n'est possible que lorsque $k = 1$. Bien sur, dans la pratique chaque jeu de données étant fini, une limite pour k existe toujours rendant $\mathscr{S}$ calculable. Une séquence de taille k est notée k-séquence. Toutefois, il existe deux définitions pour la taille d'une séquence :

- $s_k = \sum_{i=1}^{|s|} |s_i|$, basée sur la quantité totale d'items dans la séquence, cette définition est utilisée dans la majorité des publications sur le sujet, elle est la plus ancienne des deux (Agrawal et Srikant, 1995),
- $s_k = |s|$, proposée dans (Ayres *et al.*, 2002), basée sur le nombre d'itemsets dans la séquence, elle a peu été réutilisée par la suite,

Nous préférons la seconde définition qui en plus d'être plus simple permet de meilleures optimisations algorithmiques comme nous le montrerons dans notre état de l'art des algorithmes de fouille séquentiels.

Une séquence s est une sous-séquence de s' noté $s \sqsubseteq s'$ s'il existe une fonction injective $\phi : s \to s'$ préservant l'ordre qui permet de passer des itemsets de s aux itemsets de s' telle que $\alpha \sqsubseteq \phi(\alpha)$ (Zaki, 2001). La contrainte de préservation de l'ordre dans ϕ est définie à l'aide d'une fonction bijective monotone $\pi : s \to \{1, ..., |s|\}$ de la façon suivante $\forall x, y \in s, \pi(\phi(x)) \leq \pi(\phi(y)) \iff \pi(x) \leq \pi(y)$.

De plus, il existe une extension linéaire $\leq_{\mathscr{I}+}$ entre les occurrences o permettant d'ordonner celles-ci à l'intérieur de chaque s. Généralement, cette extension linéaire est

construite à l'aide de l'horodatage de chaque occurrence dans la base $\mathscr{D}$. Autrement dit, dans s les occurrences o sont triées dans l'ordre de leur insertion dans $\mathscr{D}$. Même si ce n'est en rien obligatoire, ce tri s'effectue habituellement dans l'ordre ascendant. En outre, chaque séquence possède un identifiant numérique unique sid et chaque itemset possède aussi son propre identifiant iid ou parfois eid pour identifiant d'événement.

Une base de données $\mathscr{D}$ contient des occurrences (ou transactions) o composées d'items $i \in \mathscr{I}$. À partir de cet ensemble d'occurences, la fouille d'itemsets fréquents cherche à trouver tous les sous-ensembles de $\mathscr{I}$ qui sont contenus dans au moins σ occurrences o qui forment la famille des itemsets fréquents $\mathscr{F}_{\mathscr{I}+|\sigma}$ en explorant l'espace des motifs $\langle \Gamma_{\mathscr{I}+}, \sqsubseteq_{\Gamma_{\mathscr{I}+}} \rangle$ sur lequel nous reviendrons plus tard. Dans l'immédiat, ce qu'il faut en comprendre est que tous les espaces des motifs (ex : itemsets, séquences) sont représentés à l'aide d'une paire $\Gamma_{\_}$ et $\sqsubseteq_{\Gamma_{\_}}$ correspondant respectivement la structure des motifs et l'opérateur permettant de vérifier la correspondance entre les données et les motifs.

De façon similaire, la fouille de motifs séquentiels est une généralisation de la fouille d'itemsets qui à partir d'une base de données $\mathscr{D}$ contenant des occurrences o composées d'items $i \in \mathscr{I}$ et dont chacune des occurrences appartient à un unique ensemble s dans l'univers des séquences $\mathscr{I}+$, cherche à trouver tous les sous-ensembles de S qui sont contenus dans au moins σ séquences s qui forment la famille des séquences fréquentes $\mathscr{F}_{\mathscr{I}|\sigma}$ en explorant l'espace des motifs $\langle \Gamma_{\mathscr{I}}, \sqsubseteq_{\Gamma_{\mathscr{I}}} \rangle$.

À partir de là, nous pouvons définir deux langages : le langage des données Δ et le langage des motifs Γ. Le premier Δ permet de représenter les éléments présents dans l'espace des données et le second Γ permet de représenter les éléments présentes dans l'espace des motifs. Pour qu'un motif m soit fréquent et appartienne à $\mathscr{F}_{\_|\sigma}$ il faut que sa représentation dans Γ doit être instanciée au moins par σ occurrences dans $\mathscr{D}$. Cette relation d'instanciation est notée $\lhd$.

Dans le cas de la fouille d'itemset et de la fouille de motifs séquentiels, les deux lan-

gages de données sont en réalité identiques puisque les items i apparaissent directement dans les motifs. Autrement dit, la relation d'instanciation $\lhd$ correspond à une égalité stricte entre tous les éléments dans le motif et les éléments correspondants dans une séquence. C'est-à-dire que pour qu'une séquence supporte un motif, tous les éléments du motif doivent apparaitre exactement dans la séquence en question. Il s'agit donc ici de l'inclusion ensembliste $\supseteq$.

Dans le cas de la fouille d'itemsets (ou de séquences) généralisés, la distinction entre les deux langages apparait. En effet, à cause de l'apparition d'une hiérarchie dans l'univers des items du langage de motifs, il n'est plus simplement question de tester l'égalité stricte entre les données et les motifs mais de tester la subsomption entre un élément dans une séquence et un élément dans un motif. Les éléments du langage de motifs deviennent maintenant des classes qui font partie d'une hiérarchie $H_C = \langle C, \sqsubseteq_C \rangle$. Ainsi, un élément dans le langage des données peut maintenant correspondre avec plusieurs classes puisque celles-ci partageant une hiérarchie commune pouvant se subsumer les unes aux autres.

Une séquence généralisée s est une sous-séquence généralisée de s' noté $s \sqsubseteq_{H_C} s'$ s'il existe une fonction injective $\phi_{H_C} : s \rightarrow s'$ préservant l'ordre qui permet de passer des itemsets de s aux itemsets potentiellement plus génériques de s' telle que $\alpha \sqsubseteq \phi_{H_C}(\alpha)$.

La relation d'instantiation $\lhd_{H_C}$ entre une séquence généralisée et un motif séquentiel généralisé est moins restrictive que la relation d'instantiation $\lhd$ pour les motifs séquentiels. En effet, $\lhd_{H_C}$ nécessite de satisfaire $\sqsubseteq_{H_C}$ pour chaque itemset plutôt que $\sqsubseteq$. C'est-à-dire $\forall \alpha \in s, \alpha \sqsubseteq_{H_C} \phi_{H_C}(\alpha)$. Nous pouvons définir $\sqsubseteq_{H_C}$ via $\alpha \sqsubseteq_{H_C} \beta \iff \forall i \in \alpha, \exists i' \in \beta, i' \in H_C \uparrow (i)$. En d'autres termes, tous les itemsets dans la séquence généralisée doivent être subsumés par des itemsets différents dans la super-séquence généralisée.

Cette hiérarchie de classes H_C peut être définie à l'intérieur d'une ontologie $\Omega =$

$\langle C, P, \sqsubseteq_\Omega, \rho \rangle$ au sein de laquelle C correspond aux classes, P représente les propriétés, $\sqsubseteq_\Omega$ est ici l'opérateur de généralisation qui fonctionne pour les deux hiérarchies ainsi que $\rho \subseteq C_\Omega \times P_\Omega \times C_\Omega$ qui lui représente les relations ternaires (triplets) entre les classes.

Puisqu'il est composé à l'aide de Ω, le langage de motifs devient maintenant Γ_Ω. Ceci permet d'enrichir encore plus Γ en lui ajoutant la capacité d'utiliser des liens/propriétés entre les classes déjà présentes. Le second avantage est que Ω renforce la correspondance aussi les entre Δ et Γ grâce aux instances peuplant une base de connaissances KB peuplant Ω. Nous définissons $KB = \langle O_\Omega, \lhd_{KB}, \rho_O \rangle$ dans laquelle O_Ω correspond à l'ensemble des instances, $\lhd_{KB}$ étant l'opérateur d'instanciation entre les éléments de Ω et KB ("rdf :type" ou plus communément "is a") et et $\rho_O \subseteq O_\Omega \times P_\Omega \times O_\Omega$ représentant les triplets entre les instances. Par la suite, le langage de données devient donc $\Delta_\Omega : \{s = (\zeta, \theta)\}$ avec ζ les classes de la séquence ($\zeta \in O$) et θ les propriétés($\theta \in 2^{\rho_O}$).

Ces nouvelles propriétés, à l'instar des classes, forment une hiérarchie $H_P = \langle P, \sqsubseteq_P \rangle$ définie dans Ω. À partir de là, il devient possible d'exploiter les relations inter-objets dans les motifs puisqu'elles deviennent alors présentes à la fois dans le langage de motifs grâce aux classes, aux propriétés et aux notions de domaine et codomaine, ainsi que dans le langage de données et aux relations inter-objets.

Ainsi, chaque motif extrait des séquences utilisateurs possède le format $\{\zeta, \theta\}$ avec $\zeta \sqsubseteq C_\Omega$ et $\theta \sqsubseteq \{C_\Omega \times P_\Omega \times C_\Omega\}$. Autrement dit, ζ représente la séquence de concepts, et θ représente l'ensemble des propriétés liant les concepts présents dans la séquence. Le lecteur attentif aura remarqué que ζ correspond aux motifs séquentiels classiques.

De façon abstraite, un motif est noté $\{\langle C_1, ..., C_k \rangle, \{P_1(x, y),...,P_{k'}(w, z)\}\}$ où $\zeta = \langle C_1, ..., C_k \rangle$ et $\theta = \{P_1(x, y),...,P_{k'}(w, z)\}$. Dans un contexte de logs webs, ce motif peut donc être interprété de la façon suivante : "Les utilisateurs visitent la classe C_1, puis la classe C_2, puis la classe C_3, sachant que C_1 et C_3 sont reliés par la propriété P_1,

et que C_2 et C_3 sont reliés par la propriété P_2".

Une séquence de données liées s est une sous-séquence de données liées de s' noté $s \sqsubseteq_\Omega s'$ s'il existe deux fonctions injectives préservant l'ordre ϕ_{Ω_C} et ϕ_{Ω_P} qui permettent respectivement de passer des classes de $s.\zeta$ aux classes de $s'.\zeta$ et de passer des triplets dans $s.\theta$ aux triplets dans $s'.\theta$. Pour être en mesure de simplifier les interactions entre les éléments dans $\_.\zeta$ et $\_.\theta$, nous utiliserons des entiers représentant des positions numériques au sein de chaque ensemble plutôt que les éléments d'origine. De plus, représenter les éléments dans les deux structures à l'aide de leurs positions respectives permet de s'assurer que les appariements des sujets et objets des propriétés correspondent à leurs appariements de classes sans avoir à rajouter une troisième contrainte. Ainsi, la correspondance entre les classes devient $\phi_{\Omega_C} : \{1, \ldots, |s.\zeta|\} \rightarrow \{1, \ldots, |s'.\zeta|\}$ et de même pour les propriétés, $\phi_{\Omega_P} : \{1, \ldots, |s.\theta|\} \rightarrow \{1, \ldots, |s'.\theta|\}$. Nous rappelons que toutes les fonctions $\phi\_$ sont injectives et préservent l'ordre.

- l'appariement de toutes les classes est défini par $\phi_{\Omega_C} : \forall i \in s.\zeta, \exists \phi_{\Omega_C}(i) \in \{1, ..., |s'.\zeta|\}$,
- l'appariement de toutes les propriétés est défini par $\phi_{\Omega_P} : \forall i = z(x, y) \in s.\theta \wedge x, y \in s.\zeta, \exists \phi_{\Omega_P}(i) \in \{1, ..., |s'.\theta|\}$,

La relation d'instantiation $\lhd_\Omega$ entre une séquence d'instances et un motif séquentiel de données liées englobe maintenant complètement $\lhd_{H_C}$ et a fortiori $\lhd$. En effet, $\lhd_\Omega$ nécessite de satisfaire les mêmes contraintes que $\sqsubseteq_\Omega$ mais entre un motif m et une séquence s. L'ontologie permet alors de définir un cadre pour les deux langages utilisés lors de l'extraction des motifs séquentiels de données liées.

Pour terminer cette partie sur les motifs recherchés, nous allons proposer quelques exemples de motifs. D'abord de motifs séquentiels de données liées simples apparentés aux motifs séquentiels généralisés puis de motifs séquentiels de données liées complets.

À partir d'une base composée de seulement deux séquences : $\langle Pau, Villa\_Navarre \rangle$ et $\langle Montréal, Hôtel\_Ritz - Carlton \rangle$. Ici, la séquence suivante $\langle Ville, Hôtel \rangle$ est un motif séquentiel généralisé. En effet, H_C nous indique que les objets Pau et $Montréal$ sont des instances de la classe $Ville$ et que $Hôtel\_Ritz - Carlton$ et $Villa\_Navarre$ sont instances de la classe $Hôtel$.

Ainsi, nous pouvons maintenant proposer un exemple de motif séquentiel de données liées comme nous l'avons précisé au-dessus, ces derniers sont à la fois constitués d'une séquence de classes, et d'un ensemble de propriétés reliant les classes présentes au sein de la séquence. A titre d'exemple, voici le type de motifs séquentiels généralisés que nous cherchons à découvrir : $\{\langle Ville, Hôtel \rangle, \{avoir\_un\_hôtel(0,1)\}\}$. Nous voyons bien grâce à l'ontologie Ω que $Ville$ et $Hôtel$ sont des classes (au sens RDFS ou OWL du terme) et que $avoir\_un\_hôtel(0,1)$ est une propriété de la classe $Ville$ (domaine) qui la lie à la classe $Hôtel$ (codomaine), 0 et 2 représentant ici les positions des éléments dans le motif. Évidemment, nous pouvons construire des modèles plus complexes : $\{\langle Capitale, Hôtel\_De\_Luxe, Musée \rangle, \{avoir\_au\_moins\_un\_hôtel\_de\_luxe(0,1\rangle,$ $avoir\_au\_moins\_un\_musée(0,2)\}\}$.

Pour faire le lien avec les arguments exposés au dessus, des motifs incorporant des relations entre éléments permettent de mieux comprendre la cause « sémantique » ou réelle de pourquoi ce motif est fréquent, autrement dit dans notre cas, "pourquoi cette suite d'événements a-t'-elle lieu ?". Ou au contraire, se rendre compte que deux événements non sémantiquement liés dans notre ontologie apparaissent fréquemment ensemble, c'est-à-dire qu'il existe probablement une propriété entre les deux mais elle n'apparaît pas (encore) dans l'ontologie. C'est la raison pour laquelle de telles connaissances peuvent s'avérer intéressantes entre autres pour la compréhension du passage d'un objet à un autre dans la séquence et apporter plus de précision aux séquences de classes qui non seulement gagnent en abstraction (hiérarchies) mais aussi en précision (propriétés). Toutefois, il faut bien comprendre que nous sommes en mesure d'exploiter

les propriétés dans les motifs parce que nous savons qu'elles existent entre les objets. En effet, si des liens ne sont pas renseignés dans l'ontologie, alors nous ne sommes pas en mesure de les faire figurer dans les motifs. L'alternative étant qu'ils existent directement au sein des données en entrée.

Cependant, tenir compte des liens et des relations dans le processus de recherche conduit à la nécessité de manipuler des structures complexes et par conséquent coûteuses apparentées à des graphes orientés. En effet, les motifs de concepts/relations présentent non seulement une structure de graphes mais sont aussi composés de nœuds et d'arcs à différents niveaux d'abstraction.

1.2 Obstacles à la fouille des motifs cibles

Maintenant que nous avons détaillé quels sont les motifs que nous souhaitons identifier et extraire, nous pouvons aborder le problème principal quant à la mise en pratique de la fouille de ces motifs. Le problème majeur est la taille de l'espace de recherche des motifs. Cet inconvénient est directement hérité de la fouille de motifs séquentiels et de la fouille de motifs en général. Même si au sein de la littérature les auteurs ne s'accordent pas toujours sur la raison principale de la difficulté de la fouille de motifs, tous s'accordent sur le fait qu'il s'agit d'une combinaison des divers facteurs suivants :

- le coût de calcul de l'ensemble des occurrences supportant potentiellement un motif : il s'agit du calcul de l'espace des occurrences pour un motif, ces occurrences vont être celles sur lesquelles le processus d'appariement motif-occurrence va être exécuté,
- le coût de calcul de l'appariement entre un motif et une occurrence : il s'agit de l'espace de recherche dans une séquence pour un motif donné. Autrement dit de l'espace à parcourir à l'intérieur d'une occurrence spécifique,
- le nombre de motifs potentiels à tester : il s'agit de l'espace de recherche des motifs, l'ensemble des combinaisons à explorer,

Le premier de ces obstacles n'est pas particulièrement impacté par le passage aux motifs séquentiels de données liées. Le second obstacle subit directement l'utilisation de hiérarchies et l'ajout de propriétés. En effet, comme nous l'avons montré précédemment, $\lhd_\Omega$ englobe complètement $\lhd_{H_C}$ et $\lhd$. En pratique, ceci se traduit par des tests couteux de subsomption entre les éléments de la séquence et le motif à tester ainsi que par l'exploration des liens qui nécessite aussi des tests de subsomption. Ce qui fait de $\lhd_\Omega$ un problème de morphisme de graphe dirigé acyclique. Néanmoins, des représentations de séquences et de motifs judicieuses sur lesquelles nous reviendrons plus tard, sont en mesure de limiter le coût de $\lhd_\Omega$. La manipulation directe de Ω est aussi à inclure avec cet inconvénient.

Le dernier de ces obstacles, l'espace de recherche des motifs subit de plein fouet la richesse et la complexité de la structure offertes par l'ontologie. En d'autres termes, notre espace de recherche est composé de non pas une mais de deux hiérarchies distinctes en plus d'un ensemble de relations ternaires faisant intervenir à chaque fois deux classes et une propriété provenant respectivement de chaque hiérarchie. Nous avons donc une généralisation sur deux dimensions (classes et propriétés) en plus d'appliquer une variété de contraintes sur le contenu (triplets). Le coût d'une telle exploration est colossal.

A titre de comparaison, dans (Han et Fu, 1999) les auteurs ne souhaitaient pas aller dans la direction des motifs cross-niveaux (motifs généralisés contenant des items de niveaux hiérarchiques potentiellement distincts) et préfèrent s'en tenir à des motifs multi-niveaux uniquement (motifs généralisés faisant intervenir des items du même niveau hiérarchique uniquement mais plusieurs de ces niveaux sont explorés séparément) à cause du nombre astronomique de combinaisons potentielles à explorer. Sur la figure ci-dessous (Figure 1.1 p.14), nous avons illustré grossièrement la taille relative des espaces de recherche des motifs les plus connus en incluant les motifs multi-niveaux et cross-niveaux.

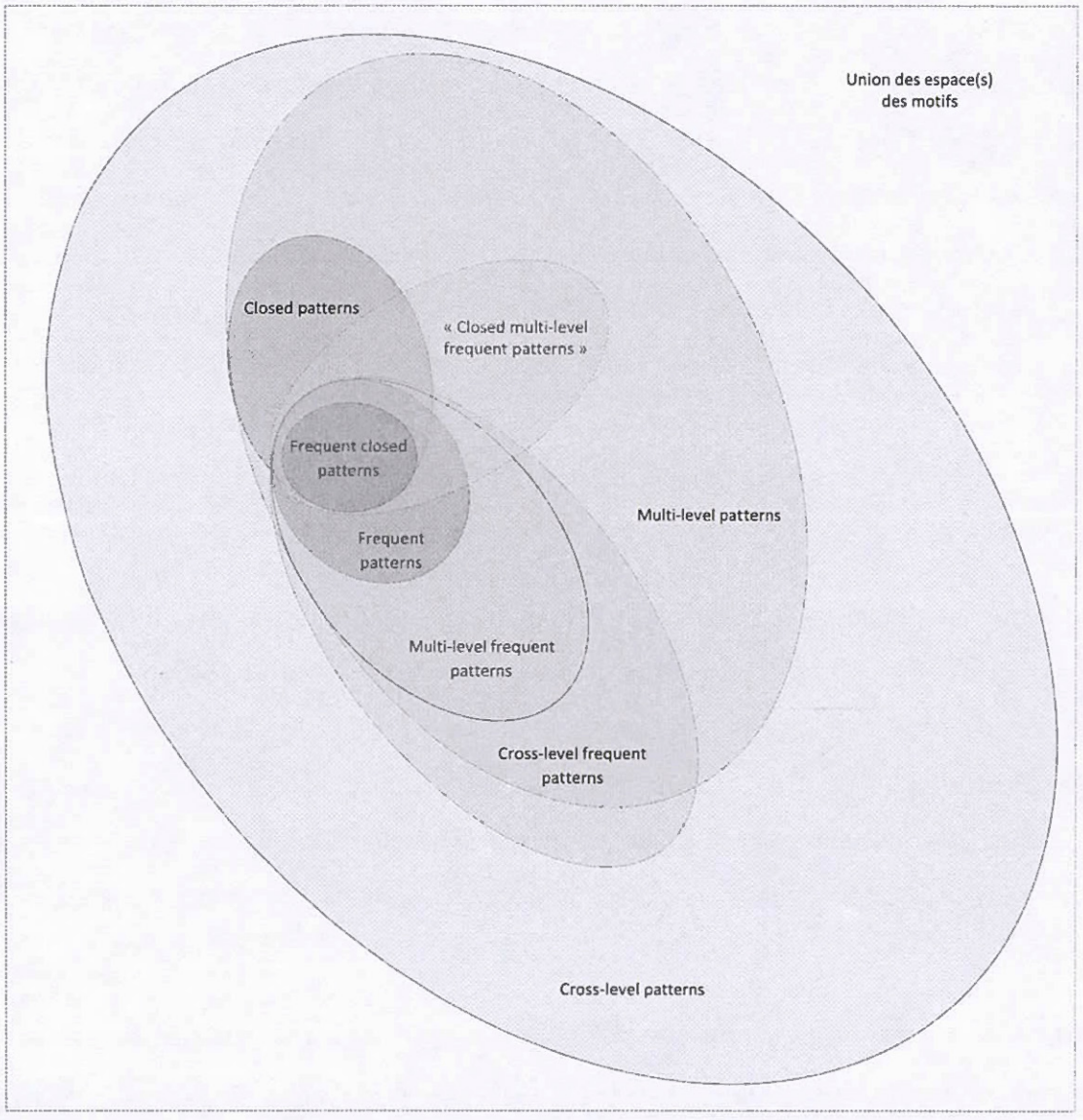

Figure 1.1: Taille de l'espace des motifs

L'espace de recherche des motifs séquentiels de données liées est donc bien supérieur à celui des motifs cross-niveaux. Nous pourrions l'approximer en le situant proche du carré de l'espace de recherche des motifs cross-niveaux tout en sachant pertinemment que habituellement le nombre de propriétés est bien inférieur à celui des propriétés. Une autre approximation serait de considérer l'espace de recherche des motifs cross-niveaux multiplié par la moyenne de propriétés compatibles entre deux classes. C'est-à-dire que

l'exploration de notre espace de recherche s'apparente au parcours de tout l'espace des motifs cross-niveaux pour chacune des propriétés possibles. Ceci sans compter que puisque une propriété connecte deux positions dans un motif, plus le motif est long, plus il faut tester de propriétés.

Sur la figure suivante (Figure 1.2 p.16), nous avons repris les espaces de recherche de la figure précédente en y ajoutant l'espace de recherche des motifs séquentiels de données liées. Dans ce travail, nous cherchons à fouiller l'espace des motifs coloré en jaune, labelisé par "Linked Data frequent patterns", autrement dit motifs de données liées fréquents. Les espaces supplémentaires par rapport à la figure précédente sont tous liés à l'ajout des propriétés. Il est aisé de constater de la difficulté de fouiller et d'explorer un espace d'une telle taille.

A l'origine, ces motifs séquentiels de données liées et XpMiner (Adda *et al.*, 2007) l'algorithme permettant de les extraire, ont été développés pour permettre d'effectuer de la recommandation web (Adda *et al.*, 2007; Adda, 2008) sur un portail de e-tourisme. Nous l'avons par la suite appliqué sur des données synthétiques de navigation dans ce portail web ainsi qu'a des données provenant de l'extraction de workflows phylogéniques.

Il est clair qu'une méthode top-down est nécessaire car une approche bottom-up passe à coté d'une partie de l'information en se contentant de généraliser. Cependant, une approche top-down subit directement l'épaisseur de la hiérarchie. En effet, il est nécessaire de fouiller chaque niveau hiérarchique pour arriver jusqu'en bas. Il serait souhaitable de disposer d'une technique permettant de prendre des raccourcis dans la hiérarchie. Une technique naïve effectuerait des sauts dans la hiérarchie en se basant sur le support des itemsets singletons à tous les niveaux. Autrement dit, si on connaît les fréquences des itemsets de taille une à tous les niveaux, nous serions en mesure d'observer les variations de fréquences lors de généralisations ou spécialisations. A partir

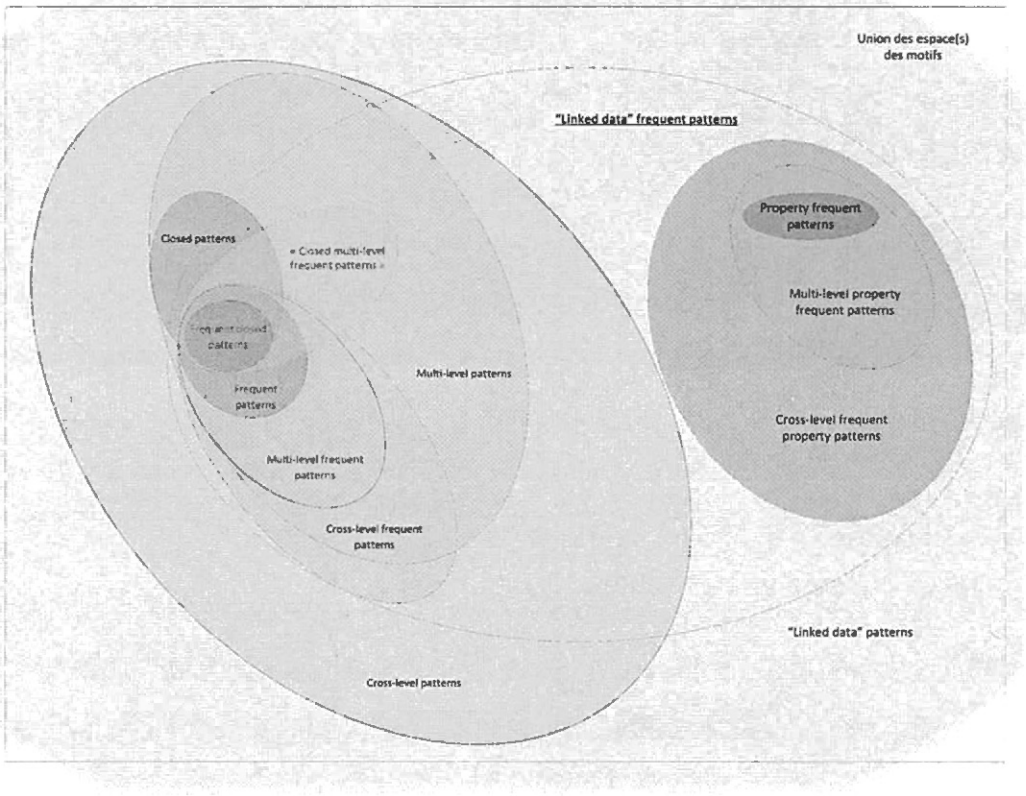

Figure 1.2: Nouvelle taille de l'espace des motifs

de là, il est envisageable de calculer des estimations de sauts réalistes tout en sachant que si les sauts sont trop grands (zone infréquente) il est possible de remonter via un procédé dichotomique et trouver la bonne frontière.

En outre, il pourrait aussi servir à valider ou enrichir une ontologie grâce au principe selon lequel si l'on trouve des motifs sans propriétés alors il existe certainement un lien sémantique entre certains des éléments du motif.

Quoi qu'il en soit, dans l'immédiat, il est vital de développer une approche efficace parcourant raisonnablement toute l'ontologie.

CHAPITRE II

ÉTAT DE L'ART : FOUILLE DE MOTIFS SÉQUENTIELS

Dans cette seconde partie, nous allons proposer un état de l'art des approches de fouille de motifs séquentiels fréquents. Cette étude est motivée par le fait que les motifs séquentiels de données liées, malgré leur complexité, restent des motifs séquentiels. Par conséquent les approches de fouille propres à ces derniers sont à privilégier pour leur recherche. Ainsi être en mesure de repérer les approches les plus efficaces nous permettra de concevoir une approche de fouille de motifs séquentiels de données liées fréquents encore plus efficace.

Pour débuter cet état de l'art, nous proposons un diagramme (Figure 2.1) qui reprend les algorithmes que nous avons étudié pour ce travail. Notre état de l'art va se concentrer sur les approches proposant des avancées algorithmiques relatives aux réductions des divers espaces de recherche ainsi qu'aux techniques de comptage de support. Ainsi, les algorithmes de fouille de motifs séquentiels fermés ou maximaux sont pas considérés puisque leurs principaux apports ne contribuent pas directement à ces deux branches. Nous ne tenons pas compte non plus des approches incrémentales parce que leur façon d'aborder le problème de fouille est trop éloignée de nos considérations, que ce soit en termes de parcours, de structures ou, plus globalement, de performances.

Cette figure permet de constater à la fois l'évolution du travail des auteurs ainsi que l'influence d'une approche sur l'autre (parfois cela est mentionné de façon explicite et

parfois l'étude de l'algorithme et la chronologie montre que il y a une influence certaine d'un travail sur l'autre).

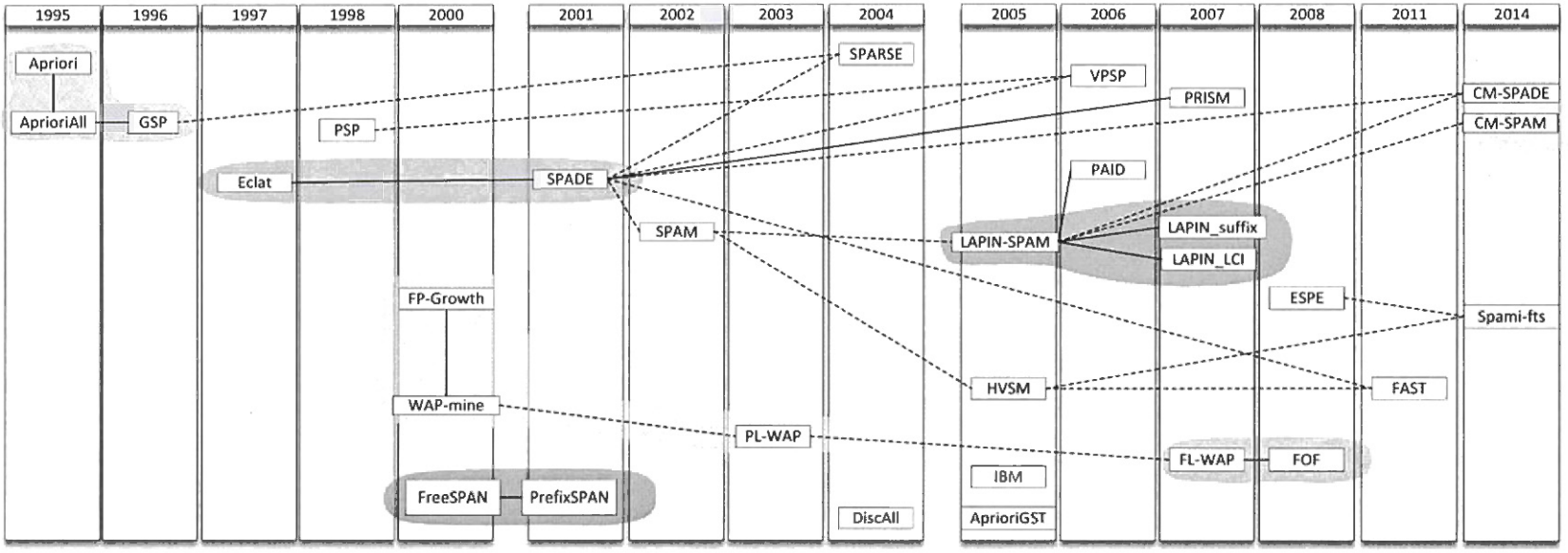

Figure 2.1: Carte des algorithmes et des relations entre eux

Nous avons découpé la période de 1995 à 2016 en trois générations représentant les trois grandes périodes de la fouille de motifs séquentiels. Ces trois périodes sont séparées par les publications les plus importantes selon nous : GSP (Srikant et Agrawal, 1996), SPADE (Zaki, 2001) et LAPIN(-SPAM) (Yang et Kitsuregawa, 2005). Nous faisons aussi figurer les ancêtres de certaines approches qui proviennent de la fouille d'itemsets fréquents lorsque l'approche en question est un portage direct. Toutes ces approches sont regroupées lorsqu'elles sont très proches. Dans cette figure, les traits pleins représentent une approche par les mêmes auteurs et un trait pointillé est utilisé lorsqu'une approche est une évolution directe d'une autre.

2.1 Première génération (1995-2000)

Dans cette première partie, nous allons aborder uniquement GSP (Srikant et Agrawal, 1996) et PrefixSpan (Pei *et al.*, 2001). Il s'agit des deux approches à la fois les plus connues même si elles sont parmi les plus anciennes. Les autres approches PSP (Masseglia *et al.*, 1998), FreeSpan (Han *et al.*, 2000a) et Wap-mine (Pei *et al.*, 2000) ne seront pas détaillées puisque dans le cas de FreeSpan, PrefixSpan l'améliore grandement et dans le cas de PSP il s'agit d'une approche qui n'a été reprise qu'une seule fois par la suite. En outre, Wap-mine constitue une approche intéressante mais son successeur PL-WAP (Lu et Ezeife, 2003) en reprend les principes. Nous détaillerons donc PL-WAP dans la seconde partie de notre état de l'art de la fouille de motifs séquentiels fréquents.

GSP, Srikant *et al.* 1996

Dans (Agrawal et Srikant, 1995) et (Srikant et Agrawal, 1996), les auteurs introduisent le problème de la fouille de motifs séquentiels fréquents. Ils proposent plusieurs algorithmes inspirés par Apriori (Agrawal *et al.*, 1994) dont le principal est GSP. Ce dernier recherche les motifs niveau par niveau dans toutes les séquences de la base de façon

horizontale et utilise les k-fréquents du niveau précédent comportant des $k-2$-préfixes identiques pour générer les candidats du niveau suivant $k+1$.

GSP est le premier à proposer l'intégration de hiérarchies ainsi que la gestion de contraintes sur les motifs. Il donnera naissance aux deux branches qui sont la fouille avec hiérarchies et la fouille avec contraintes. Nous reviendrons sur la gestion des taxonomies dans la partie dédiée à la fouille avec ontologies. Les contraintes qu'il propose d'intégrer sont d'ordre structurelles, elles permettent de restreindre les motifs en donnant un espacement maximal (ou minimal) entre deux éléments consécutifs d'une séquence. GSP est aujourd'hui dépassé en termes de performance.

GSP fonctionne à l'aide de séquences dites contiguës. Une séquence est contiguë par rapport à une autre si elle est obtenue en enlevant un item dans l'un des itemsets. De plus cette relation est transitive (si s' est contiguë par rapport à s et que s'' est contiguë par rapport à s' alors s'' est contiguë vis à vis de s). Ceci permet de définir un ordre de décomposition implicite et ainsi de structurer l'espace des candidats/motifs. Cette notion de contiguïté est utilisé majoritairement lors de la phase d'élagage. Tout comme dans Apriori, il est nécessaire que toutes les séquences contiguës de taille $k-1$ soient fréquentes pour que le candidat soit conservé.

Nous ne détaillerons pas ici les techniques de comptage de support. Ce qu'il faut en retenir c'est que les comptages nécessitent d'effectuer une passe complète sur la base. Les techniques proposées dans l'article permettent d'effectuer une passe par niveau et ainsi limiter le nombre total de passes nécessaires. Les auteurs regroupent les candidats dans un arbre de hachage pour regrouper les candidats et ainsi effectuer moins de passes sur la base. Malgré cela, le processus de calcul de support reste très coûteux à cause de la représentation horizontale des séquences utilisateur.

PrefixSpan, Pei *et al.* 2001

(Pei *et al.*, 2001) approfondissent les idées proposées dans FreeSpan (Han *et al.*, 2000a). La limitation principale de FreeSpan est le coût de gestion des bases de séquences projetées. En effet, dans FreeSpan, une séquence peut s'étendre à n'importe quelle position. Il doit vérifier toutes les combinaisons possibles, ce qui est coûteux lorsque les motifs deviennent trop longs. La projection se base uniquement sur les préfixes fréquents d'un motif. PrefixSpan travaille uniquement avec les préfixes et projette uniquement les suffixes. Dans chaque base projetée, les extensions correspondent aux items fréquents locaux. Les projections d'une séquence A sont des sous-séquences A' par rapport à un préfixe B. PrefixSpan propose trois types de projections : niveau-par-niveau, bi-niveau et pseudo-projection. Les deux premières sont les mêmes que celles proposées dans FreeSpan. La troisième, la pseudo-projection est une projection conçue pour fonctionner directement en mémoire à l'aide de curseurs sur les 1-projections plutôt que de les reconstruire physiquement à chaque étape récursive. Ce fonctionnement réduit drastiquement la quantité d'opérations à effectuer lors de la construction d'une projection. En effet, aucune séquence n'a besoin d'être copiée, que ce soit en mémoire ou directement sur le disque (ce qui a un coût encore plus grand). En outre, puisque les séquences ne sont pas physiquement répliquées (ou du moins des portions postfixes de séquences) le gain en espace mémoire est considérable.

Les auteurs comparent PrefixSpan avec FreeSpan (projection alternative) et GSP. Les auteurs utilisent un jeu de données synthétique généré avec le générateur de (Srikant et Agrawal, 1996) contenant 1000 items, 10000 séquences d'en moyenne 8 itemsets de taille 8. Il s'agit donc d'un jeu de données relativement dense. Prefixspan1 et PrefixSpan2 sont très proches. FreeSpan est un peu moins efficace alors que GSP est distancé de très loin.

Les auteurs expérimentent avec le nombre et le coût des opérations I/O nécessaires. Ils tentent d'utiliser des pseudo projections directement sur le support de stockage et comparent ainsi les coûts I/O des techniques de projection sur disque. Pour cela, ils

utilisent un jeu de données d'un million de séquences. Ils observent clairement que PrefixSpan-2 (bi-niveau) bat PrefixSpan-1 (niveau-par-niveau) sur une telle charge. Ils observent aussi que comme soupçonné, les coûts de pseudo-projection directement sur le disque sont trop importants car plus du double d'opérations est nécessaire entre pseudo-projection et projection physique si tout s'effectue directement sur le disque. Malgré ces différences de performance, les deux versions de PrefixSpan (projection physique) évoluent toutes les deux de façon linéaire même si PrefixSpan-2 creuse l'écart lorsque la charge augmente.

2.2 Seconde génération (2001-2005)

Cette seconde partie introduit les approches de type verticales, plus consommatrices en mémoire mais beaucoup plus efficaces que celles mentionnées précédemment. Nous avons fait le choix de séparer ces approches de celles présentées plus haut car celles-ci s'appuient sur une représentation alternative de la base de séquences : une base verticale.

Cette base verticale s'apparente à un index inversé sur les séquences. En effet, ceci permet très rapidement de déterminer le support d'un motif à partir du support et des positions de chaque item le composant. Au sein de cette famille d'algorithmes, nous allons présenter SPADE, SPAM, PL-WAP et HVSM. Les autres approches, DISC-all, SPARSE et IBM, ne seront mentionnées seulement que dans cette introduction.

DISC-all (Chiu *et al.*, 2004) constitue une approche complètement différentes des autres. Elle fonctionne selon des indexations et des partitionnements récursifs qui permettant de trouver l'ensemble des séquences fréquentes sans explicitement compter les supports des candidats. Toutefois, l'indexation s'apparente à des tris de partitions. Le fonctionnement de DISC-all ne s'apparente en rien aux approches utilisées dans notre travail, nous ne l'aborderons pas plus en détail. SPARSE (Antunes et Oliveira, 2004)

est une approche hybride entre GSP et SPADE mais qui ne propose pas de nouveaux concepts. Nous détaillerons seulement SPADE. IBM (Savary et Zeitouni, 2005) est seulement une technique intéressante de compression de base verticale.

SPADE, Zaki et al. 2001

SPADE (Zaki, 2001) est le premier représentant de la famille de ECLAT (Zaki *et al.*, 1997), celle des algorithmes verticaux. SPADE signifie « Sequential PAttern Discovery using Equivalence classes » ce qui se traduit par découverte de motifs séquentiels en utilisant des classes d'équivalences.

Premièrement, SPADE propose la première approche verticale appliquée à la fouille de motifs séquentiels fréquents, ce qui change complètement la donne en termes de performances. Deuxièmement, SPADE propose une approche basée sur un hyper-treillis des séquences et un partitionnement récursif de celui-ci selon classes d'équivalence. Ceci pour diviser le problème de fouille, potentiellement trop volumineux pour le résoudre directement en mémoire, en une multitude de sous-problèmes pouvant être résolus indépendamment et rapidement.

SPADE construit une représentation semi-verticale des séquences : on associe à chaque item/atome la liste des séquences qui le contiennent sous forme de couples d'identifiants entiers : identifiant de séquence (*sid*) et position de l'événement/item (*eid*) dans la séquence. SPADE peut reconstruire n'importe quelle k-séquence en associant ses k atomes/items entre eux. De plus, il est possible d'obtenir le support exact de cette séquence en effectuant un join "temporel" des id-lists des atomes. En effet, il suffit d'aggréger une par une les id-lists des atomes en respectant leur ordre dans la séquence en tenant compte du fait que les *sid* doivent être identiques et que *eid* doit être : soit identique à l'*eid* du précédent s'ils sont tous deux dans le même itemset ("equality join"), soit plus grand si ce n'est pas le cas ("temporal join"). Remarquons que l'on peut retrouver n'importe quelle k-séquence en combinant les listes de deux de ses

$(k-1)$-sous-séquences. Les auteurs proposent une approche en profondeur d'abord ainsi qu'une approche en largeur d'abord. Sachant que plus une séquence grandit en taille, plus sa id-list diminue alors plus les séquences à joindre sont grandes, plus l'intersection devient simple à calculer (Srikant et Agrawal, 1996).

Concernant le découpage en sous-problèmes, les auteurs proposent de considérer l'espace des motifs séquentiels comme un hyper-treillis dans lequel l'opération "join" entre deux motifs séquentiels ne renvoie plus uniquement un seul résultat mais un ensemble de super-motifs séquentiels. En réalité, c'est la relation de "sous-sequence" qui définit un hyper-treillis de plusieurs dimensions car un atome/item peut être ajouté de plusieurs façons différentes à une séquence et ainsi donner plusieurs nouvelles séquences. Les auteurs défissent les opérations de "meet" et "join" sur l'hyper-treillis. Toutefois, nous ne proposons ici que le $join$ car seul celui-ci est utile à la compréhension de SPADE. Simplement, le $join$ de plusieurs séquences correspond à l'ensemble des super-séquences communes minimales à toutes ces séquences. Par exemple, $join(\{a\},\{b\}) = \langle\{a,b\}\rangle$, $\langle\{a\},\{b\}\rangle$ et $\langle\{b\},\{a\}\rangle$.

La structure de l'hyper-treillis des séquences nous mène assez intuitivement vers une approche de type bottom-up (en accord avec la représentation du treillis, les atomes sont en bas), comme tous les algorithmes présentés jusque ici, que l'on peut aisément combiner avec une heuristique de type Apriori, à savoir qu'une séquence ne peut être fréquente que si toutes ses sous-séquences sont fréquentes. Or, en réalité, il s'avère que la quantité de mémoire vive disponible n'est pas toujours suffisante pour résoudre l'intégralité du problème. En effet, parcourir le treillis directement par niveaux demande beaucoup trop de mémoire. Il faut donc le découper en sous-treillis qui eux peuvent être parcourus directement. C'est ici qu'intervient la notion de classe d'équivalence, basées sur des préfixes communs. Deux séquences font partie de la même classe d'équivalence si elles partagent un $(k-1)$-préfixe commun. Nous pouvons remarquer qu'il n'y a pas d'implémentation physique de l'hyper-treillis. En réalité, il s'agit d'un arbre lexicogra-

phique.

De l'aveu des auteurs eux mêmes, dans la pratique, la partitionnement initial ($k = 1$) est suffisant, le partitionnement récursif n'étant pas nécessaire à moins de travailler sur des grandes quantités de données. En outre, les auteurs proposent deux stratégies de parcours des classes d'équivalences la première en largeur d'abord (BFS) et la seconde en profondeur d'abord (DFS). L'avantage de BFS par rapport à DFS est qu'elle dispose de plus d'informations pour l'élagage au niveau k puisque l'on connaît toutes les $(k-1)$-classes avant d'attaquer les k-classes (car l'élagage nécessite plusieurs classes d'équivalence du même niveau $k-1$). Cependant, DFS requiert moins de mémoire puisqu'il ne nécessite de conserver en mémoire qu'une classe courante et sa classe mère (pour pouvoir passer ensuite à ses autres classes sœurs).

Pour chaque candidat généré par SPADE via $join$, une première étape de pré-élagage effectue l'élagage de Apriori, c'est à dire qu'il s'assure que toutes les séquences de taille $k-1$ soient fréquentes. Si c'est le cas, alors il calcule l'intersection ("temporal" ou "equality join") des id-lists. Si le candidat est fréquent, il est ajouté à la nouvelle classe d'équivalence (préfixe).

Au niveau des expérimentations, SPADE (BFS uniquement) a été seulement comparé avec GSP sur des jeux de données réels et synthétiques. Les avantages de SPADE par rapport à GSP sont les suivants : grande diminution des coûts E/S, il n'est pas nécessaire de rechercher les séquences à proprement parler dans toute la base de séquences mais uniquement dans l'index inversé ce qui permet d'éviter la majeure partie du travail de GSP. La seule opération nécessaire est de croiser des listes qui rapetissent avec l'augmentation de la profondeur de la classe d'équivalence. L'autre remarque c'est que même si les auteurs ne le mentionnent jamais de façon explicite, les classes d'équivalence de SPADE correspondent intuitivement aux notions de projection (FreeSpan, PrefixSpan) et de bases conditionnelles (WAP-mine). Tous ces algorithmes se basent

sur la notion de réduction progressive/récursive de l'espace des motifs, ce que GSP, PSP et AprioriAll ne font pas du tout.

SPAM, Ayres *et al.* 2002

L'apport majeur de SPAM (Ayres *et al.*, 2002) à la fouille de motifs séquentiels fréquents est la représentation de l'index inversé proposé par SPADE (Zaki, 2001) à l'aide de vecteurs de bits pour être en mesure de calculer rapidement les intersections nécessaires. L'autre apport intéressant est un changement dans la définition des k-séquences. En effet, les auteurs proposent de définir la taille des séquences non plus en fonction du nombre d'items présents mais en fonction du nombre d'itemsets.

SPAM reprend l'arbre lexicographique de préfixes proposé dans SPADE. Cet arbre est exploré en profondeur d'abord. La différence par rapport à SPADE est qu'au lieu de définir des classes d'équivalence, les auteurs définissent deux opérateurs d'extension de motif : I-Step et S-Step. Le premier, I-Step (pour "itemset-extension") permet d'étendre le dernier itemset de la séquence avec un item. En accord avec la définition présentée au début, la taille de la séquence ne change donc pas. Le second, S-step (pour "sequence-extension") permet d'étendre une séquence avec un nouvel itemset singleton. Même si l'arbre reste le même que dans SPADE, le parcours de celui change un peu. En effet, plutôt que d'ajouter indistinctement des super-séquences étendues via I-Step ou S-Step à une classe d'équivalence, SPAM génère et teste d'abord toutes les S-Step puis dans un second temps les I-Step. Cependant, même s'il est construit dans un ordre différent, l'arbre des candidats reste le même, les classes d'équivalence sont manipulées de façon implicite.

Pour être en mesure de comprendre les bonnes performances de SPAM, il faut se pencher sur la façon dont l'index inversé est géré. Ici, contrairement à SPADE il n'est aucunement question de listes de paires (sid, eid) mais d'un vecteur de bits par item et par séquence. Il s'agit d'une représentation complètement verticale. Chaque vecteur

de bits à pour taille la taille de la séquence. Si l'item est présent dans l'itemset à une position i ($i > 0$) alors le bit à cette position devient 1. Dans le cas contraire, il reste à 0. À partir de cette représentation basée sur les itemsets, effectuer une I-step revient à simplement intersecter les vecteurs de bits des deux items à considérer. Cette opération nous permet de trouver la co-occurence d'items dans un itemset spécifique. D'où l'intérêt de modifier la définition des séquences pour faire correspondre leur taille à la taille des itemsets et pas au nombre d'items distincts (alignement). Nous ferons aussi remarquer que si les vecteurs de bits sont implémentés à l'aide de types primitifs (ex : int16, int32) alors il est possible d'intersecter en une seule opération autant d'itemsets que le type contient de bits et ainsi traiter beaucoup plus d'informations avec autant de ressources. L'alternative principale étant de boucler sur un tableau de booléens ce qui est à la fois un gâchis de mémoire en plus de n'être pas particulièrement efficace. A partir de là, le comptage de support est extrêmement simple, si le vecteur de bits en sortie n'est pas rempli de zéros (différent de zéro si implémente dans un entier) alors la séquence supporte le motif.

Pour ce qui est de la S-step, le fonctionnement est un peu moins judicieux. En effet, le calcul de la S-step cherche à savoir si oui ou non il existe un item compatible plus "à droite" dans la séquence. Or, par nature, une telle opération n'est pas calculable par intersection. Pour être en mesure d'appliquer la technique proposée pour la I-step, le vecteur de bits courant doit être transformé. Concrètement, il s'agit d'une transformation simple qui identifie le premier bit du vecteur (le plus à gauche, de poids fort), le met à zéro puis met le reste des bits plus à droite à un. L'explication derrière cette transformation est simple, c'est qu'il faut trouver un élément compatible strictement plus à droite alors on rend compatible toutes les positions plus à droite. A partir de là, une intersection avec le vecteur de bit du second item permet de trouver ces positions.

Pour terminer, ces deux opérations I-step et S-step sont totalement compatibles entre elles et par conséquent applicables sur les mêmes vecteurs de bits. Au niveau des per-

formances, SPAM s'avère extrêmement efficace sur les jeux de données sur lesquels les auteurs ont mené leur expérimentations. Comparativement à SPADE, SPAM est deux fois plus rapide et encore plus rapide face à PrefixSpan. Toutefois, il faut remarquer que le coût de construction des index rend SPAM mois efficace que PrefixSpan sur de très petits jeux de données. Lorsque la taille des itemsets grandit dans les séquences, l'écart de performances se fait d'autant plus sentir, ce qui est parfaitement logique puisque c'est là où la technique d'intersection de vecteurs mise en avant par SPAM est la plus efficace. En revanche, SPAM nécessite beaucoup plus de mémoire que SPADE. Les items non représentés dans une séquence sont quand même présents dans le vecteur de bits sous forme de zéros. Ainsi SPADE consomme entre x5 et x20 fois moins de mémoire par rapport à SPAM. Il est intéressant de faire remarquer que les auteurs de SPAM mettent en avant le fait que leur algorithme est extrêmement efficace sur des longues séquences. Or ce n'est tout simplement pas le cas en réalité. La contribution des auteurs s'avère judicieuse uniquement sur des séquences courtes mais avec de grands itemsets. Comme nous l'avons indiqué au-dessus, SPAM n'intègre aucun mécanisme d'élagage à priori mis à part l'arbre lexicographique et les transformations de vecteurs de bits des séquences lors des S-extensions se révèlent être l'opération la plus coûteuse de l'algorithme. Même si en théorie, ils sont très différents, dans la pratique, SPADE et SPAM sont très semblables. En effet, la différence principale est que SPAM utilise des bitmaps et est donc plus rapide (x2,5) mais consomme beaucoup plus de mémoire. Il s'agit donc de choisir l'un ou l'autre selon les circonstances.

PL-WAP, Ezeife *et al.* 2003

(Lu et Ezeife, 2003) proposent PL-WAP, une amélioration substantielle de WAP-mine (Pei *et al.*, 2000). PL-WAP signifie "Pre-Order Linked WAP-tree", autrement dit arbre WAP liée selon préordre. L'objectif premier dans PL-WAP est d'éviter la construction récursive des WAP-tree intermédiaires pour les raisons expliquées dans WAP-mine. La recherche conditionnelle dans WAP-mine est basée sur la notion de suffixe commun à

tous les préfixes. PL-WAP recherche les préfixes communs, il explore l'espace des motifs en cherchant les super-séquences fréquentes via leur préfixe commun. Si un item e est fréquent dans les suffixes (descendants) d'un motif P, alors Pe est fréquent. Ce qui est l'inverse de WAP-mine. Par exemple, pour trouver le motif $\langle a, b, c, d \rangle$ WAP-mine commence par chercher $\langle d \rangle$, puis $\langle c, d \rangle$ puis $\langle b, c, d \rangle$ puis enfin $\langle a, b, c, d \rangle$. PL-WAP commence par la séquence $\langle a \rangle$ et à l'aide de ses arbres de suffixes trouve la séquence fréquente $\langle a, b \rangle$ puis dans l'arbre de suffixe de $\langle b \rangle$ on recherche $\langle a, b, c \rangle$ puis finalement $\langle a, b, c, d \rangle$. PL-WAP étend les motifs en ajoutant des items qui apparaissent dans les suffixes de l'item en question représentés dans les nœuds dans les branches directement sous le motif en question dans le WAP-tree. Le changement principal de PL-WAP par rapport à WAP-mine concerne la construction de l'en-tête de l'arbre et l'ajout de codes uniques à chaque nœud pour accélérer les comparaisons. Contrairement à WAP-mine, les queues des items ne sont pas construites au fur et à mesure de l'insertion des séquences dans l'arbre. Elles sont construites lorsque l'arbre est complètement rempli avec toutes les séquences. Chaque nœud possède un code de position sous forme d'une chaîne binaire qui permet par la suite de retrouver le positionnement d'un nœud par rapport à un autre rapidement.

Pour chaque 1-séquence fréquente chacun des nouveaux arbres des suffixes conditionnels sont fouillés en suivant les liens de l'item en question depuis l'en-tête pour trouver la première occurrence de cet événement dans chacun des arbres. Ensuite, PL-WAP ajoute les compteurs de chacun de ces premiers nœuds au support pour le motif. Les codages binaires sont utilisés pour comparer les positions des noeuds entre eux (même branche, plus bas, plus à droite, etc.).

Le premier constat est qu'il est nécessaire de reparcourir à chaque fois tous les nœuds entre la racine de l'arbre conditionnel et l'en-tête de l'arbre. Le second constat est que PL-WAP est un mélange entre WAP-mine et PrefixSpan. Au final, le travail de fouille revient à suivre la première apparition d'un item dans chacune des sous-branches dans

chacun des arbres des suffixes de séquences.

Les auteurs ne se sont comparés qu'avec WAP-mine et GSP et les performances offertes par PL-WAP sont bien supérieures à celles des deux autres. À ce stade de l'état de l'art il est clair que GSP n'est pas l'approche la plus efficace mais la comparaison avec WAP-mine est surprenante. En effet, cette comparaison est intéressante parce que cela montre que même avec des stratégies "grossières", si les opérations d'écriture sont supprimées, des implémentations correctes d'approches plus anciennes sont en mesure de battre des algorithmes plus récents qui a priori semblent plus efficaces stratégiquement parlant (WAP-mine). Ainsi, cette approche met en évidence le coût des opérations d'écriture par rapport à une autre qui ne fait que lire et tester des conditions. À noter que (Mabroukeh et Ezeife, 2010) se comparent avec LAPIN (Yang *et al.*, 2007) et sont, selon eux, supposés le battre, ce qui n'est pas le cas dans nos expérimentations présentées à la section 5.1.

HVSM, Song *et al.* 2005

Avec leur approche HVSM (Song *et al.*, 2005) ameliorent SPAM (Ayres *et al.*, 2002). HVSM signifie "First-Horizontal-last-Vertical scanning". Avant toute chose, il ne faut pas comprendre que les deux représentations (horizontales et verticales) sont utilisée dans HVSM. Cette désignation représente les deux dimensions d'une table : les lignes (vertical) et les colonnes (horizontal) utilisée dans la fouille de séquences. Ce que cela signifie c'est que HVSM parcourt l'espace des motifs d'abord selon les colonnes (k-itemsets) puis ensuite selon les lignes (k-séquences). Ce découpage permet de réduire l'espace de recherche. En réalité, l'algorithme crée deux espaces de recherches (items dans k-itemsets et k-itemset dans k-séquences). Cette approche renforce l'idée que la fouille de séquences est une généralisation de la fouille d'itemsets. HVSM exploite quasi-pleinement la définition des séquences dans SPAM et s'avère plus rapide que SPAM. Ceci est relativement évident puisque beaucoup moins de redondances sont

présentes dans le travail à effectuer.

Le parcours reste le même que SPAM. Nous pouvons l'assimiler à deux enchaînements de SPAM, le premier uniquement sur les transactions (avec uniquement I-Steps) pour trouver tous les itemsets fréquents puis dans un second temps sur les itemsets fréquents en tant qu'items (avec uniquement des S-steps). Dans les deux cas, une représentation verticale sous forme de vecteurs de bits est utilisée. Il s'agit du premier (et pour l'instant aussi du dernier) algorithme de fouille de motifs séquentiels qui étend un motif avec un itemset plutôt qu'avec un item.

En réalité, il n'est pas nécessaire de reconstruire une deuxième base verticale puisque celle-ci est construite en fouillant les itemsets fréquents (I-steps). La différence majeure avec SPAM en termes de fonctionnement interne concerne le calcul des S-step. Évidemment, SPAM nécessite de calculer l'intégralité du nouveau vecteur à chaque fois puisqu'il mélange les extensions d'itemsets et de séquences, ce qui n'est plus le cas ici.

Les "S-step" ne sont plus faites avec des ET logiques (bitwise-AND) comme dans SPAM mais avec un comptage simple. Nous pouvons remarquer que HVSM ne trouve qu'une seule solution alors que SPAM recherche toutes les solutions. L'autre différence dans le fonctionnement interne est que HVSM effectue de l'élagage à la Apriori lors de la génération des candidats. Ainsi il progresse en largeur d'abord puisqu'il nécessite tous les k-séquences fréquentes pour générer et élaguer les candidats de taille $k+1$. Ce changement est la raison pour laquelle HVSM consomme plus de mémoire que SPAM. Le fait que SPAM utilise une technique de compression et pas HVSM ne joue pas non plus en faveur de HVSM au niveau de sa consommation mémoire et par conséquent aussi sur son temps d'exécution.

2.3 Troisième génération (2005-2016)

Dans cette troisième et dernière partie, quasiment toutes les approches sont de type verticale et introduisent le principe d'élagage précoce (Yang et Kitsuregawa, 2005). L'élagage précoce consiste à éliminer un maximum de candidats le plus tôt possible dans le processus de fouille. Les approches que nous allons détailler sont LAPIN, PAID, PRISM, FL-WAP, FOF, Spami-FTS et CM-SP(ADE/AM).

Deux autres approches FAST et VPSP ne seront pas détaillées. FAST (Salvemini *et al.*, 2011) n'apporte aucune nouvelle approche. En effet, ce travail reprend le fonctionnement en deux temps de HVSM et l'approche verticale de SPADE. Quoi qu'il en soit, FAST est très rapide d'où son nom. VPSP (Kraemer *et al.*, 2006) est une amélioration verticale de PSP.

LAPIN-(SPAM), Yang *et al.* 2005

Dans cette partie nous allons discuter de deux papiers (Yang *et al.*, 2007; Yang et Kitsuregawa, 2005) qui traitent de plusieurs algorithmes différents basés sur un nouveau principe, la notion d'élagage précoce (« early pruning »). Selon les auteurs, le problème majeur des algorithmes de fouille de séquences concernent la taille de l'espace de recherche des motifs. Ce qui est directement lié à l'incapacité à explorer l'espace des motifs séquentiels de grande taille.

Les trois algorithmes, LAPIN-SPAM, LAPIN-LCI et LAPIN-suffix partagent le préfixe LAPIN qui signifie « Last Position Induction » ou induction de dernière position. L'idée derrière cette nomenclature est d'utiliser les dernières occurrences d'items dans les séquences pour savoir le plus tôt possible qu'une combinaison n'a aucune chance d'exister et qu'il n'est pas pas nécessaire de la générer et de la tester. Par exemple, si l'on sait que l'item b n'apparaît jamais après l'item c, il n'est pas utile de générer le motif $\langle b, \ldots, c \rangle$. Il s'agit donc d'un élagage encore plus puissant que la propriété «

downward closure » d'Apriori sans toutefois être incompatible avec cette dernière.

À peu de choses près, cette technique est très proche de la branche des algorithmes fonctionnant avec des bases compressées sous forme d'arbres (WAP-mine, PL-WAP, etc.). En effet, alors que ces approches ne construisent uniquement que des candidats qui « existent » dans la base de séquences, LAPIN ne construit pas des candidats qui ne peuvent pas « ne pas exister ». Dans les deux cas, et c'est là l'intuition importante, il est nécessaire de ne considérer que des motifs qui existent au moins une fois dans la base des séquences. Autrement dit, les candidats doivent être « data-driven » plutôt que « pattern-driven ».

LAPIN-SPAM étant le premier à avoir été publié (2005) et essentiellement basé sur SPAM, que nous avons déjà présenté. Nous concentrerons l'essentiel de nos explications sur ce dernier. Son fonctionnement est identique à SPAM à ceci près qu'il intègre un mécanisme d'élagage beaucoup plus efficace.

La première étape de LAPIN-SPAM est la construction de deux tables : une pour les extensions de séquences (S-step) et l'autre pour les extensions d'itemsets (I-step). En réalité, LAPIN-SPAM ne construit directement que la table des extensions de séquences car l'autre table peut être construite par intersection à partir des positions dans un second temps. Ensuite, construit des listes de positions iid dans l'ordre ascendant pour chaque item dans chaque séquence. On recherche le $(k-1)$ "prefix border position" avec une recherche binaire. L'opération essentielle devient alors la recherche pour une position plus grande que celle du "prefix border position". Si oui, on teste/évalue le motif, sinon rien n'est fait. Selon les auteurs, même si SPAM est efficace, ses opérations de comptage de support sont trop coûteuses (ils ne distinguent pas entre I-step et S-step) surtout si l'on prend en compte la récursivité de l'algorithme.

LAPIN-LCI signifie « local candidate item-list » ce qui en réalité correspond au fonctionnement de SPADE (Zaki, 2001). L'autre approche, LAPIN-Suffix, est basée sur la

constatation que lorsque la taille de la liste d'items locaux candidats est en moyenne plus grande que les suffixes existant dans la « position list » alors il devient plus intéressant de scanner celle-ci pour trouver les extensions possibles plutôt que de continuer à travailler avec les listes locales. Ceci se fait en se déplaçant dans la liste de positions puis en prenant le premier élément dont la dernière position est plus grande que la position courante (préfixe).

Selon les auteurs, LAPIN-SPAM serait plus efficace que SPAM par un facteur de deux à trois. En outre, les deux version de LAPIN sont comparées à PrefixSpan. Elles sont toujours plus efficaces que PrefixSpan parce que l'approche de LAPIN permet un élagage de l'espace des motifs beaucoup fin que de simples bases projetées construites de façon récursive. Cependant, LAPIN-suffix est moins efficace que LAPIN-SPAM sauf pour des petits jeux de données où les suffixes dans les tables sont très restreints. En effet, plus la taille des séquences augmente dans les jeux de données, plus LAPIN-SPAM devient efficace (jusqu'à deux fois plus) au détriment de LAPIN-suffix et à fortiori de PrefixSpan (jusqu'à quasiment dix fois).

PAID, Yang *et al.* 2006

PAID (Yang *et al.*, 2006) signifie "PAssed item deduction" autrement dit il s'agit d'une approche qui "déduit" les supports des motifs à partir des informations qui lui sont fournies. Aussi, il s'agit d'une amélioration (même si publié avant) de LAPIN (Yang *et al.*, 2007) et par conséquent d'une autre approche verticale.

L'apport de cette contribution est la réutilisation des valeurs de support des motifs fréquents pour le calcul de leurs descendants. En effet, PAID propose de remplacer l'opération d'agrégation des *sid* au sein desquels un motif apparaît par une élimination des *sid* des parents du motif au sein desquels le motif n'apparaît pas. Selon les auteurs, l'étape de calcul du support d'un motif est la plus coûteuse de toutes et s'en tenir à un élagage de l'espace des *sid* n'est pas suffisant. Leur technique est inspirée par (Partha-

sarathy *et al.*, 1999) une approche incrémentale. Ceci dit, il est parfaitement logique de se tourner vers des approches incrémentielles puisqu'elles se focalisent sur les économies de parcours des espaces étant donné qu'elles doivent constamment se mettre à jour les supports (ainsi que les motifs) en fonction des nouvelles (et anciennes) séquences.

Étant construit sur LAPIN, PAID réutilise la notion de position clé dans une séquence pour un motif donné. La différence majeure est que PAID, en plus d'utiliser ces positions pour déterminer les combinaisons qui n'existent pas, s'en sert pour décrémenter directement le support du motif parent plutôt que de tout cumuler à partir de zéro. Ainsi, il n'y a plus besoin de scanner la base de données, qu'elle soit sur le disque ou en mémoire.

Le principe repose sur la recherche des positions des 2-séquences dont le 1-préfixe est le dernier item ajouté au motif fréquent grâce aux tables indexant les 2-séquences et 2-itemsets. PAID boucle alors sur la liste des 1-suffixes et si la dernière position n'est pas supérieure à celle du motif à évaluer alors PAID décrémente le support dudit motif.

D'après les expérimentations des auteurs PAID est beaucoup plus efficace que LAPIN_LCI et PrefixSpan. Plus il y a de répétitions dans les motifs, plus l'écart entre PAID et PrefixSpan augmente puisque PAID utilise des listes d'items distincts alors que PrefixSpan parcourt des bases (pseudo-)projetées contenant toutes les séquences distinctes. La différence avec LAPIN_LCI repose sur l'économie d'espace sur les *sid*. Dans ce cas, nous pourrons remarquer que SPADE tient déjà compte de cet élagage de l'espace des *sid* mais qu'il est le plus lent dans les expérimentations des auteurs. Néanmoins, SPADE n'élague que localement l'espace des motifs même s'il dispose d'une technique de calcul selon nous très efficace, il teste beaucoup trop de combinaisons, d'où ses performances laissant à désirer.

Cependant, il est important de faire remarquer que ces expériences ne montrent pas vraiment l'apport de la technique de PAID mais plutôt la différence entre des tech-

niques qui utilisent un élagage des *sids* et d'autres non. Selon nous la technique de décrémentation proposée par PAID ne constitue pas une avancée majeure puisqu'elle propose de décrémenter en parcourant la même liste qu'il faudrait parcourir avec une technique de cumul. Ainsi, la parcourir en incrémentant ou décrémentant avec la condition inverse revient plus ou moins au même.

Depuis lors, PAID a fait l'objets de travaux plus récents CC-PAID (Matsubara *et al.*, 2011) et CCDR-PAID (Matsubara *et al.*, 2012), visant à le paralléliser ainsi qu'à tirer profit du cache CPU afin d'augmenter encore plus ses performances.

PRISM, Gouda *et al.* 2007

(Gouda *et al.*, 2007) proposent PRISM, une amélioration de SPAM basée, en apparence, sur un encodage de vecteurs de bits à l'aide d'une factorisation en nombres premiers. Leur nouvelle approche PRISM signifie "PRIme-Encoding Based Sequence Mining".

Ce que PRISM permet de montrer est qu'une segmentation en blocs de la mémoire permet de pré-calculer toutes les combinaisons possibles pour un problème en apparence complexe. Ce pré-calcul permet à la fois de consommer beaucoup moins de mémoire puisqu'il n'y a pas de vecteurs de bits complets à proprement parler à maintenir (seulement des indices) et d'accélérer (un peu) les calculs.

En réalité, PRISM gère deux bases verticales (*sid* et *iid*) et les opérateurs s'appliquant dessus sont tous pré-calculés à l'aide d'une approche basée sur les treillis booléens et accessoirement la factorisation en nombres premiers "square-free", c'est-à-dire dont la puissance du multiple est toujours inférieure < 2. Si l'on considère les n plus petits nombres premiers dans l'ordre croissant comme un vecteur, alors il devient aisé de se représenter les factorisations susmentionnées sous la forme d'un vecteur de bits représentant la présence (ou l'absence) de chaque nombre premier au sein de la factorisation.

Ainsi, nous pourrons considérer ces facteurs premiers comme des atomes d'un treillis (ou les items) et les factorisations des nombres entiers comme des combinaisons de ces mêmes atomes pour ainsi construire un treillis booléen des nombres entiers. Il est donc possible d'assimiler chaque identifiant de séquence ou d'itemset dans ce treillis ainsi que toutes leurs combinaisons.

Ceci nous montre qu'à l'aide d'une représentation sous forme de treillis, il est possible de représenter l'ensemble des intersections possibles entre un nombre d'ensembles limites ainsi que leurs intersections (et unions). Il est maintenant possible d'utiliser ce treillis pour représenter tous les *sid* et *iid* via un découpage en petits blocs chacun présent dans le treillis, chaque configuration de bloc étant assignée à un nombre entier unique. Dès lors calculer une intersection entre deux listes devient une simple recherche dans une matrice aux positions représentées par les identifiants des blocs dans le treillis. Il s'agit d'une technique ad-hoc, d'autres travaux beaucoup plus poussés sur les vecteurs de bits existent tels que (Chambi *et al.*, 2015). De plus, maintenant que la famille des intersections possibles est limitée PRISM pré-calcule aussi les cardinalités de chaque bloc pour accélérer les calculs de supports ainsi les opérations de masquage pour la S-step de SPAM. Pré-calculer les cardinalités est selon nous discutable puisque les jeux d'instruction actuels sont en mesure de calculer les cardinalités de types primitifs extrêmement rapidement (ex : Intel pop_count()). Ainsi, la différence majeure par rapport à SPAM est qu'au lieu de laisser à l'implémentateur le soin d'imaginer sa propre technique de calcul d'intersection de vecteur de bits, ici elle est détaillée. Néanmoins, le parcours de l'espace de recherche dans PRISM est le même que dans SPAM (arbre lexicographique avec S-step et I-step) avec les inconvénients que nous avons déjà exprimés plus haut. Les valeurs des nœuds du treillis sont utilisées pour rechercher dans les tables (matrices[i][j]) que ce soit pour les intersections de blocs, le calcul des cardinalités des vecteurs et le calcul des transformations pour la S-Step.

Les auteurs comparent PRISM avec SPAM (Ayres *et al.*, 2002), PrefixSpan (Pei *et al.*,

2001) et SPADE (Zaki, 2001). La version de PRISM implémentée utilise des blocs de huit nombres premiers. SPADE est battu par un facteur de 4, PrefixSpan par un facteur de 10. SPAM termine toujours second et parfois même premier mais utilise trop de mémoire ce qui fait qu'il n'est pas en mesure de terminer dans certains cas.

En résumé, on peut assimiler PRISM à du SPAM utilisant des vecteurs de bits segmentés sur lesquelles les opérations (AND, MASK et CARD) sont pré-calculées pour accélérer les computations. Toutefois, nous ferons remarquer que sur 4 bits (blocs de 4 nombres premiers), les indices dans le treillis de PRISM montent jusqu'à 210 alors que sur 4 bits avec puissances de 2 l'indice maximal est 16. Donc au final si l'on se passe des nombres premiers, les matrices représentant les opérateurs peuvent être beaucoup plus compactes et la mémoire mieux exploitée. De plus, assigner une valeur à un nœud du treillis demande de calculer des multiplications (une seule fois) alors que si on utilise des vecteurs de bits ceci est natif.

FL-WAP, Yang *et al.* 2007

FL-WAP (Tang *et al.*, 2007) constitue une amélioration de PL-WAP (Lu et Ezeife, 2003) signifiant arbre WAP lié par premières occurrences ("First-Occurrence Linked WAP-tree"). La première différence concerne la façon dont les nœuds sont connectés au sein de l'en-tête de l'arbre PL-WAP. La seconde différence est que tout comme dans WAP-mine, les sous-arbres sont construits. Nous rappellerons que FL-WAP ne fouille que les séquences d'itemsets singletons comme WAP-mine et PL-WAP.

Les auteurs proposent leur propre version de l'arbre PL-WAP, l'arbre FP-WAP au sein duquel uniquement les premières occurrences de chaque item dans chaque branche sont connectées à travers l'en-tête. Ici, il s'agit donc exactement du même type d'arbre d'aggrégation que dans (Lu et Ezeife, 2003). PL-WAP utilise tous les nœuds alors que seulement certains d'entre eux sont nécessaires pour évaluer le support d'un motif. En effet, comme nous l'avons fait remarquer précédemment, par rapport à un nœud

racine, seules les premières occurrences de chaque item sont nécessaires pour calculer l'extension d'un motif par rapport à ce nœud racine.

L'inconvénient dans FL-WAP est que puisque l'arbre ne contient que les premières occurrences et qu'il est exploré en descendant alors très rapidement il n'y a plus de liens à explorer. Pour solutionner cela, les auteurs proposent de reconstruire des sous-arbres FL-WAP à l'instar de WAP-mine. Cependant, ceci élimine l'apport principal de PL-WAP qui est de ne pas reconstruire les sous-arbres.

D'après les expériences des auteurs, FL-WAP est plus rapide que PL-WAP par un facteur entre 5 et 30. Si l'on s'en tient à nos remarques concernant PL-WAP, sa capacité à ne pas reconstruire les sous-arbres ne devrait pas être vaincue par une approche mettant en avant la copie de ces mêmes sous-arbres. L'explication à ce phénomène est la suivante : PL-WAP, même si il ne recopie pas les sous arbres, évalue beaucoup trop de nœuds lors de la recherche des premières occurrences. D'ailleurs, ceci est certainement la raison pour laquelle les auteurs ont intégré un mécanisme d'encodage des nœuds plutôt que de s'en tenir à la nouvelle structure et son parcours. Plus les motifs grandissent, plus l'exploration nécessaire est profonde dans l'arbre PL-WAP. Or PL-WAP ne conserve pas les nœuds jouant le rôle de racine, il doit donc recommencer son parcours depuis le début à chaque fois. Ainsi, même si en principe, les bases projetées diminuent en taille avec l'agrandissement du motif, l'espace des nœuds à explorer dans PL-WAP augmente quand même. C'est la raison pour laquelle FL-WAP, en s'en tenant uniquement aux dites bases projetées qui diminuent, même si il doit les copier, est beaucoup plus rapide. Cependant, puisqu'il construit des arbres supplémentaires, il consomme plus de mémoire que PL-WAP.

FOF, Yang *et al.* 2008

FOF (Peterson et Tang, 2008), évolution de FL-WAP (Tang *et al.*, 2007), propose de remplacer les arbres FL-WAP par des forêts de premières occurrences ("First Oc-

curence Forests") qui correspondent simplement à des pointeurs vers des nœuds de l'arbre d'initial jouant le rôle de racines pour simuler des sous-arbres. Il s'agit alors d'une version de PL-WAP avec pointeurs sur les nœuds ce qui évite de refaire tout le travail et permet de se passer d'encodage des positions des nœuds dans l'arbre PL-WAP puisqu'ils sont ici tous "sous" le nœud courant. Autrement dit, FOF est une version de FL-WAP sans reconstruction de sous-arbres qui ne fait que suivre des pointeurs délimitant ces mêmes sous-arbres. Ceci devrait s'avérer très efficace comme le prétendent les auteurs.

Cependant, FOF n'est pas en mesure d'élaguer a priori l'espace des motifs. En effet, il cherche à étendre un motif fréquent avec tous les motifs 1-fréquents (items) alors que FL-WAP, puisqu'il copie et recalcule les sous-arbres jouant le rôle de bases projetées, est en mesure de maintenir à jour les en-têtes des arbres et donc de raffiner l'ensemble des extensions possibles. FL-WAP est donc conscient de items distincts à tout moment dans chacune des projections sans avoir à les re-parcourir intégralement comme le fait FOF. Ceci oblige donc FOF à devoir tester l'extension d'un motif par tous les items fréquents ce qui est selon nous une aberration.

Selon les auteurs, FOF battrait FL-WAP mais leurs expérimentations ne semblent pas assez conséquentes pour l'affirmer avec certitude. Selon nous, par rapport à PL-WAP il est possible que sur des jeux de données de séquences très courtes mais avec beaucoup d'items PL-WAP se révèle plus performant que FL-WAP.

Spami-FTS, Febrer-Hernández *et al.* 2014

(Febrer-Hernández *et al.*, 2014) introduisent Spami-FTS, un nouvel algorithme de fouille de motifs séquentiels basé sur l'extension des motifs à l'aide de 2-séquences fréquentes. Cet algorithme est inspiré par LAPIN (Yang *et al.*, 2007) et ESPE (Hsieh *et al.*, 2008). En deux mots, ESPE est basé sur l'énumération des 2-séquences à partir des séquences utilisateurs. Il s'agira donc ici d'utiliser les 2-séquences pour étendre les

motifs fréquents et élaguer l'espace des candidats. Il s'agit aussi d'un algorithme de type élagage précoce comme les algorithmes de la famille de LAPIN. L'idée principale repose sur le fait que jusqu'à lors, les algorithmes de type "pattern-growth" étendent dans un premier temps les motifs fréquents avec toutes les 1-séquences fréquentes (ex : PSP (Masseglia *et al.*, 1998)) ou avec les items qui ont permis de trouver des motifs fréquents au niveau précédent (ex : SPAM !(Ayres *et al.*, 2002)) ou encore en recherchant des items appartenant aux super-sequences du motif (ex : PrefixSpan (Pei *et al.*, 2001)). Dans un second temps, d'autres motifs infréquents sont utilisés pour élaguer les nouveaux candidats (souvent, pour un niveau k, tous les motifs de niveau $k-1$). Ici le principe est simplement d'utiliser les 2-séquences fréquentes comme des maillons d'une chaîne qui forment le motif. Alors, en disposant d'une structure indexant ces motifs de façon adéquate, nous pouvons étendre un motif en lui ajoutant uniquement les 2-séquences compatibles. Pour qu'une 2-séquence soit compatible avec un motif il suffit que l'item dans sa dernière position soit égal a l'item en première position de la 2-séquence. Par exemple, si le motif $\langle a, b, c\rangle$ est fréquent, cela implique que $\langle a, b\rangle$, $\langle b, c\rangle$ et $\langle a, c\rangle$ sont fréquents aussi. Supposons que $\langle c, a\rangle$ et $\langle c, c\rangle$ soient aussi fréquents. Alors nous pouvons étendre $\langle abc\rangle$ avec $\langle c, a\rangle$ et $\langle c, c\rangle$ pour rechercher $\langle a, b, c, a\rangle$ et $\langle a, b, c, c\rangle$. Remarquons que $\langle a, b\rangle$, $\langle b, c\rangle$ et $\langle a, c\rangle$ ne sont pas compatibles car ils ne commencent pas par l'item à la dernière position dans le motif à étendre.

La propriété justifiant cette stratégie est, encore une fois, la propriété de "downward closure" d'Apriori (**?**). En effet, toutes les 2-séquences infréquentes ne peuvent être contenues dans aucun autre super-motif fréquent. Toutefois en étendant les motifs avec ces séquences fréquentes les candidats peuvent aussi s'avérer infréquents et un élagage de type Apriori (avec les $k-1$-motifs) pourrait éliminer plus de candidats. Néanmoins, un tel mécanisme est beaucoup coûteux à calculer et nécessite une approche de type largeur d'abord pour fonctionner de façon optimale. Ici, même si les auteurs ne le mentionnent pas, il est possible de fonctionner à la fois en largeur d'abord, en profondeur

d'abord (pour consommer moins de mémoire) ou à l'aide d'une stratégie hybride. Les séquences et les motifs sont représentées de façon verticale à l'aide de vecteurs de bits. Les auteurs proposent un vecteur par item et motif dont les positions correspondent aux identifiants des séquences. Les positions des items et motifs dans chaque séquence sont représentées comme une liste de positions numériques. En outre, Spami-FTS intègre une nouvelle stratégie d'élagage basée sur la notion de support potentiel maximal. En effet, chaque motif dispose de son propre vecteur de bits représentant les séquences dans lesquelles il apparaît ou non. Ainsi, intersecter les vecteurs de deux motifs nous permet de connaître les séquences dans lesquelles tous deux apparaissent. Toutefois, ils peuvent apparaître dans n'importe quel ordre. Depuis SPADE(Zaki, 2001), nous savons que cette opération n'est pas suffisante pour trouver le support d'un nouveau motif. Néanmoins, cette opération est suffisante pour trouver une borne supérieure pour le support du nouveau motif. Ainsi, si la cardinalité du vecteur intersecté est strictement inférieure au support minimal le motif ne peut en aucun cas être fréquent. Il est donc possible de l'éliminer. Dans le cas contraire, l'intersection nous permet de connaître exactement dans quelles séquences nous devons chercher à comparer les positions d'occurrences des motifs pour retrouver le support exact. Puisque nous connaissons déjà les occurrences respectives des deux motifs dans chacune des séquences, il suffit de trouver au moins une position pour laquelle l'autre motif apparaît plus à droite pour pouvoir incrémenter le support du motif. Nous ferons remarquer que même si les auteurs ne le mentionnent pas, l'inverse est aussi possible. En effet, nous pouvons rechercher les séquences où une telle combinaison n'est pas possible et faire décrémenter le support potentiel maximal jusqu'à descendre en dessous de *minsup*.

Nous ferons aussi remarquer que contrairement à ce que proposent les auteurs, il est évident qu'il n'est pas nécessaire de reparcourir toutes les séquences lors de la recherche du support exact pour un motif. Parcourir seulement les séquences renvoyées par le calcul d'intersection est algorithmiquement beaucoup plus intéressant. Ce n'est à

aucun moment explicitement indiqué par les auteurs mais nous supposons que Spami-FTS fonctionne de la même façon que HVSM, c'est-à-dire qu'il trouve tous les itemsets fréquents lors de la phase de recherche des 1-séquences fréquentes puis les combine dans les phases ultérieures. PRISM et Spami-FTS semblent offrir sensiblement les mêmes comportements, bien que Spami-FTS soit un peu plus rapide, alors que LAPIN, en ayant la même stratégie d'exploration que PRISM, est loin derrière. En augmentant le nombre d'items par transactions les performances de Spami-FTS devraient augmenter sensiblement et distancer les autres algorithmes de type S-step/I-step cependant ce n'est pas le cas.

CM-SPADE/SPAM, Fournier-Viger *et al.* 2014

Le dernier algorithme que nous allons présenter est CM-SPADE/SPAM (Fournier-Viger *et al.*, 2014a). Tout comme Spami-FTS (Febrer-Hernández *et al.*, 2014) il étend les contributions de LAPIN (Yang *et al.*, 2007). Simplement, CM-SP(ADE/AM) étudie les co-occurences fréquentes entre items dans chaque séquence pour construire deux tables CMAP (signifiant CO-occurence map), une pour les 2-itemsets fréquents et une autre pour les 2-séquences fréquentes d'itemsets singletons.

Ensuite, lors des phases de génération de candidats, il suffit de ne pas générer un candidat avec un item qui n'apparaît pas dans la CMAP du dernier item ajouté au motif dépendament du type d'extension (S-step ou I-step).

Il s'agit encore une fois de la propriété de "downward closure" d'Apriori (Agrawal et Srikant, 1995). A ce stade, nous pouvons remarquer que dans le cas de séquences d'itemsets singletons, la CMAP_s équivaut à la structure qui maintient les 2-séquences fréquentes dans Spami-FTS. Il n'est donc pas nécessaire de justifier cette approche pour les extensions de séquences. Pour justifier les extensions d'itemsets, il suffit de se référer à SPAM, lorsqu'une I-extension est infréquente, il n'est pas nécessaire de continuer plus loin puis qu'aucun super-motif de cette extension ne peut être fréquent.

À partir de là, il est relativement évident que ce mécanisme fonctionne pour tous les préfixes en utilisant seulement les combinaisons possibles du dernier item du préfixe.

Les auteurs intègrent leur CMAP dans deux algorithmes déjà présentés : SPAM (Ayres *et al.*, 2002) et SPADE (Zaki, 2001).

- CM-SPAM : Dans SPAM les candidats sont générés de façon successive à l'aide d'une liste d'items compatibles (pour S-step et I-step). Lorsque le candidat est généré, dépendamment de la boucle dans laquelle nous nous trouvons (I-step ou S-step) alors l'algorithme recherche dans la CMAP correspondante. Si la séquence formée par le dernier item et le nouvel item n'est pas trouvée alors le candidat est éliminé d'office sans avoir à rechercher son support.
- CM-SPADE : dans SPADE les candidats sont crées en combinant les motifs de la même classe d'équivalence, c'est à dire des motifs qui partagent un $(k-1)$-préfixe commun. Lorsqu'un candidat est généré, dépendamment de sa catégorie (extension itemset ou séquence), à l'instar de CM-SPAM, une recherche dans une des deux CMAP est effectuée. Si la combinaison n'existe pas alors le candidat est éliminé.

Les performances des versions CM sont entre deux et huit fois plus rapides que les algorithmes d'origine. CM-SPADE semble généralement plus rapide que CM-SPAM ceci est du encore une fois au manque d'élagage de l'espace des motifs par SPAM. Ceci est relativement évident puisque SPADE est aussi plus rapide que SPAM dans les exemples proposés. Toutefois, la version de SPADE dans SPMF ne correspond pas tout à fait à la version d'origine. En effet, SPADE (spmf) des utilise vecteurs de bits pour représenter les *sid* alors que pas dans l'implémentation d'origine (du moins pas dans l'article). Ceci permet un calcul du support ainsi qu'un élagage des candidats extrêmement efficace.

2.4 Synthèse

Avant d'aller plus loin nous proposons deux observations critiques à l'ensemble des travaux présentés :

1. La fouille verticale semble être l'approche adaptée à la fouille de séquences. Ceci parce que cette technique permet de compenser le volume des séquences et des itemsets.
2. Un maximum de techniques de (pre-)élagage efficaces sont nécessaires. Les algorithmes les plus récents et prétendument les plus efficaces en font usage.
3. La prise en compte de tous les espaces de recherche (*sid*, *eid* et *items*) est vitale. En effet, des approches intéressantes comme FOF ne réduisent qu'un ou deux des trois espaces et dans la pratique leurs performances laissent à désirer.

Pour terminer cet état de l'art, nous allons récapituler toutes les approches présentées dans la table ci-dessous (Figure 2.2).

	Algorithme	Base	Candidats	Elagage a priori	Comptage support	Exploration	Structures remarquables
1995	AprioriAll	Horizontale	join, generer-et-tester	Apriori	Recherche dans liste	Par niveau k	Arbre de hachage pour les candidats
1996	GSP	Horizontale	join, generer-et-tester	Apriori	Recherche dans liste	Par niveau k	Arbre de hachage pour les candidats
1998	PSP	Horizontale	join, generer-et-tester	Apriori	Recherche dans liste	Par niveau k	Arbre de prefixe pour les candidats + listes de positions
2000	FreeSpan	Projection horizontale	Pattern-growth	Aucun	Recherche dans liste	Projection recursive, partout	Listes ou F-matrix pour sequences
2000	Wap-mine	Arbre d'aggregation	Pattern-growth	Aucun	Parcours arbre	Projection recursive, prefixe	Arbre d'aggregation des sequences avec supports cumules
2001	SPADE	Verticale	Join local, generer-et-tester	Si join potentiel max plus petit que minsup	Intersection sets + Recherche dans liste	Profondeur d'abord locale (tous les fils d'abord)	Ensembles de paires de (sid, iid) pour sequences
2001	PrefixSpan	Projection horizontale	Pattern-growth	Aucun	Recherche dans liste	Projection recursive, suffixe	Listes ou F-matrix pour sequences
2002	SPAM	Verticale	Pattern-growth	Aucun	Intersections vbits	Profondeur d'abord locale	Vecteurs de bits pour sequences
2003	PL-Wap	Arbre d'aggregation	Pattern-growth	Aucun	Parcours arbre	Projection recursive, suffixe	Arbre d'aggregation des sequences avec supports cumules sans reconstruction
2004	Disc-All	Horizontale	Enumeration sequences	Apriori	Tri recursif de partion	Par k-partition puis niveau	Partitionnements + Re-index. seq via k-min seq.
2005	HVSM	Verticale	Join, generer-et-tester	Aucun, pas indiqué	Recherche dans vbit	Par niveau k, itemsets d'abord	Matrice de bits pour chaque motif
2005	IBM	Verticale	Join, generer-et-tester	Apriori	Recherche dans matrice	Par niveau k	Matrice de bits compréssée globale
2005	LAPIN-SPAM	Verticale	Join, generer-et-tester	Dernières positions motifs	Recherche tables + intersection vbits	Profondeur d'abord locale	Tables des dernieres positions motifs + Vecteurs de bits
2006	PAID	Verticale	Join, generer-et-tester	Dernières positions	Recherche table + Decrementation	Profondeur d'abord locale	Tables des dernieres positions motifs
2006	VPSP	Verticale	Join, generer-et-tester	Apriori	Intersection sets + Recherche dans liste	Par niveau k	Aucune
2007	LAPIN-LCI	Verticale	Join, generer-et-tester	Dernières positions motifs	Intersection sets + Recherche dans liste	Profondeur d'abord	Tables des dernieres positions motifs
2007	FL-Wap	Arbre d'aggregation	Pattern-growth	Aucun	Parcours arbre	Projection recursive, suffixe	Arbre d'aggregation des sequences avec supports cumules sans reconstruction
2007	PRISM	Verticale	Join, generer-et-tester	Aucun	Joins pré-calculés entre blocs	Profondeur d'abord locale	Matrice de blocs pré-calculés
2007	LAPIN_suffix	Verticale	Join, generer-et-tester	Dernières positions motifs	Recherche dans liste	Projection recursive	Tables des dernieres positions motifs
2008	FOF	Arbre d'aggregation	Pattern-growth	Aucun	Parcours arbre	Projection recursive, suffixe	Forêts d'aggregation des sequences avec supports cumules sans reconstruction
2011	FAST	Verticale	Join, generer-et-tester	Aucun, pas indiqué	Croisement de listes a deux dimensions	Par niveau k, itemsets d'abord	Aucune
2014	Spami-FTS	Verticale	Chainage de 2-frequents	2-sequences frequentes	Intersection sets + Recherche dans liste	Itemsets par niveau k, sequences profondeur d'abord locale	Arbre de compatibilite des 2-sequences
2014	CM-SPADE	Verticale	Join/Local + CM	2-seq, 2-itemsets, liste locale	Intersection vbits + Recherche dans liste	Profondeur d'abord locale	Matrices de co-occurrence + Vecteurs de bits (sid) + Listes de positions (iid)
2014	CM-SPAM	Verticale	Join/Local +CM	2-seq, 2-itemsets, liste locale	Intersections vbits	Profondeur d'abord locale	Matrices de co-occurrence + Vecteurs de bits

Figure 2.2: Comparaison des caractéristiques des approches

Cette figure tient compte uniquement des caractéristiques importantes selon nous : type d'exploration, représentation de la base, méthode de calcul du support, méthode(s) d'élagage(s) a priori et l'utilisation d'une structure spécifique. Les autres caractéristiques telles que la notion de compression, de partitionnement ou la distinction memory-based/storage-based ne sont pas indicatives selon nous puisqu'elles sont directement liées à l'implémentation et non pas à l'algorithmique derrière chaque approche.

CHAPITRE III

ETAT DE L'ART : FOUILLE AVEC ONTOLOGIES

Ces dernières années de plus en plus d'algorithmes de fouille ont étés conçus afin d'être en mesure d'intégrer des connaissances du domaine. En majorité écrasante, ces algorithmes optent pour l'intégration d'une taxonomie (hiérarchie de concepts). Cependant, ils s'attaquent à une multitude de domaines parmi lesquels la fouille de règles d'associations, de motifs séquentiels et même de (sous-)graphes (Berendt, 2006).

Nous présentons dans cette section une synthèse de la littérature de la fouille des motifs en utilisant une couche sémantique représentée sous une forme taxonomique ou ontologique en tant que graphe acyclique direct des concepts et des relations.

(Anand *et al.*, 1995) est le premier travail au sein duquel est discuté l'intégration de connaissances du domaine dans le processus de fouille de données. Ce framework permet de trouver des règles proches aux règles d'associations via des mécanismes d'induction.

(Han et Fu, 1995) puis (Han et Fu, 1999) proposent un système capable de découvrir des règles d'associations multi-niveaux. La stratégie proposée par les auteurs est de type descendante. Contrairement à des règles d'associations telles qu'elles sont formulées dans (Agrawal et Srikant, 1995), ici les règles extraites sont en mesure d'utiliser différents niveaux de granularité dans une hiérarchie et ainsi gagner en utilité. Il est

cependant important de préciser qu'à cause de considérations d'ordre combinatoire de telles règles n'utilisent qu'un seul niveau de la hiérarchie. En effet, comme mentionné au chapitre 1, si l'on considère des combinaisons d'éléments de plusieurs niveaux le nombre total d'ensembles à évaluer devient colossal et par conséquent très difficile à réaliser en pratique. Cette approche se veut donc être un compromis pragmatique entre la richesse de l'information pouvant être extraite et les ressources nécessaires pour la mener à bien.

GSP (Srikant et Agrawal, 1996) suppose que la hiérarchie ne représente qu'un seul type de relation entre les items soit la relation taxonomique (*est_un*). Il s'agit de construire des motifs de différents niveaux hiérarchiques en remplaçant les items par leur ancêtre taxonomique. Cependant, cette approche reste couteuse en terme d'espace mémoire et calcul puisque la représentation de la base de transactions devient énorme avec le profondeur de hiérarchie utilisée. En effet, rajouter les items ancêtres dans toutes les transactions est impensable en pratique car cela revient à multiplier la taille de la base de données par un nombre pouvant atteindre la profondeur maximale de la hiérarchie.

(Fortin et Liu, 1996) font remarquer que les extractions de règles d'associations construisent beaucoup trop de règles, ce qui crée un problème de gestion. Selon eux ceci s'explique par le fait qu'avec des jeux de données volumineux, pour trouver des règles relativement intéressantes, il est nécessaire d'opter pour un seuil de support (et confiance) très bas et ainsi extraire une multitude d'observations supplémentaires, certes fréquentes mais ni intéressantes ni exploitables.

(Fortin et Liu, 1996) proposent un framework orienté objet pour être en mesure d'extraire des règles d'associations multi-niveaux, composées avec les éléments d'un niveau unique (Srikant et Agrawal, 1996), de plusieurs niveaux et même de plusieurs hiérarchies différentes. Selon eux, il est nécessaire de fouiller des règles inter-niveaux puisqu'elles sont encore plus informatives et utiles que les règles construites à partir

d'un seul niveau.

(Chen *et al.*, 2003) proposent d'utiliser des règles d'association même niveau uniquement ici (donc multi-niveau). L'enjeu principal selon les auteurs est que certains jeux de données sont trop creux pour pouvoir faire émerger des règles intéressantes avec de grands supports. Ils ne peuvent que représenter des informations (ex : intérêts clients) trop abstraites pour être exploitables. Simplement, ici il s'agit de remplacer les objets présents dans les tuples par leurs ancêtres dans la hiérarchie de concepts offerte par l'ontologie et ainsi de généraliser l'information à moindre coût et ainsi diminuer le nombre d'items (et donc l'espace de recherche des motifs). Ceci est différent de GSP puisque l'on remplace complètement les items, on ne les cumule pas. Les auteurs exploitent une ontologie en tant que hiérarchie de concepts pour préparer les données à la fouille d'associations. En remplaçant les occurrences de bas niveau avec les occurrences de haut niveau, nous pouvons observer que le support des règles/itemsets augmente et donc plus de règles peuvent être trouvées (pour un support fixe).

Dans (Plantevit *et al.*, 2006), les auteurs présentent une approche multi-dimensionnelle de la fouille de motifs séquentiels généralisés, les motifs séquentiels h-généralisés multidimensionnels. L'idée ici est d'exploiter les dimensions et leurs hiérarchies présentes naturellement dans les entrepôts de données.

Leur approche HYPE (HierarchY Pattern Extension) est une extension de M2SP (Plantevit *et al.*, 2005) un de leurs précédents travaux. À notre connaissance, il s'agit du premier travail qui tente d'unifier la fouille de séquences multidimensionnelles avec les hiérarchies sur les dimensions et les items et au sein d'entrepôts de données. HYPE est un algorithme par niveaux comme Apriori ainsi les motifs de niveau i sont combinés pour construire les candidats de niveau $i+1$ en sachant que sur n dimensions deux motifs combinables doivent partager $n-2$ items.

(Berendt, 2006) propose la prise en compte d'une taxonomie pour le processus de

fouille en modifiant les mesures de fréquence/support à grâce à l'utilisation une nouvelle notion d'intérêt contextuel ("context-induced interestingness"). Il s'agit avant tout d'un cadre de fouille pour retrouver les motifs fréquents qui prend en considération la généralisation entre les nœuds de graphes. Il s'agit d'intégrer les taxonomies (hiérarchies) de concepts dans le processus de recherche de sous-structures fréquentes à partir d'une base de graphes correspondant à des transactions.

Leur algorithme utilise une approche de type Apriori mais étend un motif de taille $k-1$ avec un motif de taille 1 pour faire une taille k (via arcs). Il fonctionne à la fois en profondeur d'abord et largeur d'abord mais à cause de considérations relatives à la quantité de mémoire disponible les auteurs ont opté pour une fouille en profondeur d'abord.

Selon (Zhou et Geller, 2007), il est intéressant d'utiliser une hiérarchie pour modifier les valeurs de support des tuples et ainsi mettre en évidence certains motifs. Les auteurs se basent sur l'opération de "raising" introduite dans (Chen *et al.*, 2003) à ceci près qu'ici les auteurs proposent d'effectuer directement le "raising au niveau k" qui appliquée à un tuple qui permet récupérer l'union des ancêtres de niveau k de tous les items. Elle porte le même nom que dans (Chen *et al.*, 2003) mais ses entrées et sorties sont différentes. Dans (Chen *et al.*, 2003), chaque item subit le *k-raising* et renvoie un ensemble de nouveaux tuples (introduisant au passage des erreurs de support) alors qu'ici c'est un tuple qui subit le *k-raising* et qui renvoie un autre tuple ainsi aucun élément n'est pris en compte plusieurs fois.

Il s'agit donc d'un remplacement des items d'un tuple qui permet une augmentation du support (plus d'itemsets fréquents, pas forcement plus de règles) et dans certains cas une augmentation aussi de la confiance. Cette nouvelle façon d'appliquer le principe de raising permet de toujours avoir un support plus grand pour les itemsets sans avoir à comptabiliser plusieurs fois un même tuple.Leurs résultats montrent que le niveau

minimum de généralisation se trouve dans une profondeur moyenne de l'ontologie.

Dans (Adda *et al.*, 2007), l'espace des séquences à explorer devient l'espace des hiérarchies et des triplets d'une ontologie. Ceci permet d'extraire des motifs séquentiels composés de classes et de propriétés issues de l'ontologie. Les motifs sont donc semblables à des graphes dirigés acycliques avec des liens labellisés. La spécialisation s'effectue ici à la fois sur les classes et les propriétés. Les auteurs proposent l'algorithme xPMiner. Cette méthode applique une stratégie de parcours descendante en profondeur d'abord de l'espace de recherche des motifs à l'aide de quatre opérateurs : *AddConcept*, *SpeConcept*, *AddRelation* et *SpeRelation* permettant de :

- *AddConcept* permet l'ajout d'un concept racine à la fin de la séquence de concepts.
- *SpeConcept* permet la spécialisation du dernier concept de la séquence de concepts uniquement si aucune propriété ne s'applique déjà sur lui.
- *AddRelation* permet l'ajout d'une propriété entre deux concepts distincts (un sujet et un objet) de la séquence. Néanmoins, l'objet de la propriété ne peut être que le dernier concept de la séquence de concepts.
- *SpeRelation* permet la spécialisation d'une propriété. A l'instar de *SpeConcept*, la propriété à spécialiser ne peut que avoir pour objet le dernier concept de la séquence.

Notre travail dans ce mémoire se situe dans la continuité de cette méthode. En effet, l'utilisation de classes et des propriétés d'une ontologie fait croitre l'espace de recherche des motifs de sorte que le nombre de combinaisons à explorer devient colossal, ce qui se ressent directement sur les performances de l'outil. Nous reprendrons le fonctionnement de cette approche dans le chapitre suivant.

(Rabatel *et al.*, 2010; Rabatel *et al.*, 2014) considèrent que le contexte des motifs est un apport non négligeable à la fois pour répondre à des requêtes de type "existe t-il un comportement spécifique à tel groupe d'utilisateurs ?" mais aussi pour trouver

mettre en parallèle des règles positives et négatives sur le même contexte telles que "un phénomène A est observé sur les jeunes utilisateurs mais pas sur des utilisateurs plus âgés". Dans un troisième temps cette information permet aussi de mettre en évidence des comportements spécifiques à un ou plusieurs contextes versus des comportements observables de façon générique. À notre connaissance, il s'agit du premier travail à considérer la fouille de contextes à partir de motifs séquentiels. Les travaux s'en rapprochant le plus sont les travaux sur la fouille de motifs séquentiels multidimensionnels. Selon les auteurs la limitation principale de la fouille de motifs séquentiels (et implicitement de tous les autres) est la notion de fréquence qui ne distingue pas les catégories d'utilisateurs.

Les auteurs proposent de représenter les dimensions dans une taxonomie. En d'autres termes, la fouille de motifs séquentiels contextuels correspond à une fouille de motifs séquentiels multi-dimensionnels (Pinto *et al.*, 2001) dont les dimensions sont généralisées. Leur algorithme est basé sur PrefixSpan (Pei *et al.*, 2001). Il extrait les séquences qui sont fréquentes dans au moins un contexte (nœud de la hiérarchie des dimensions). Ensuite le reste des motifs contextuels séquentiels sont extraits à partir de l'ensemble précédent.

L'objectif de (Mabroukeh, 2011) est très proche de celui dans (Adda *et al.*, 2007). Cette propose un système de recommandation web basé sur une fouille d'usage des logs web assisté par une ontologie en exploitant les relations entre objets (*is-a*, *has-a*, *part-of*). Ce travail augmente l'efficacité du système en résolvant les problèmes de démarrage à froid, "sparsity" et de sur-spécialisation du contenu et le compromis complexité-precision. Cette méthode enrichit les web logs avec de l'information sémantique grâce à une nouvelles mesure de similarité basée sur l'indice de Jaccard. De plus, la matrice de probabilité de transition des modèles de Markov construits depuis les logs est aussi utilisée dans l'approche. Cette thèse montre qu'intégrer la sémantique dans les séquences permet d'élaguer l'espace de recherche des séquences et par conséquent d'accélérer le

mécanisme de fouille. Néanmoins, leur intégration ne considère que des séquences a priori sémantiquement valides, c'est-à-dire qui sont explicables par le domaine tel que modélisé actuellement. Autrement dit, jamais la fouille ne sera en mesure de découvrir de nouvelles associations inédites puisque celles-ci ne sont pas composées d'éléments similaires. Automatiquement, cette approche se prive d'une partie (à nos yeux essentielle) de l'information.

(Ramezani *et al.*, 2014) considère une nouvelle classe de règles d'associations, les règles d'associations multi-relations. Chaque règle est constituée d'une entité et de plusieurs relations. Elles permettent d'indiquer des relations indirectes entre les entités. Leur algorithme MRAR permet d'extraire ces règles à partir de graphes dirigés contenant des arcs labellisés. Ces graphes peuvent être construits à partir de bases de données relationnelles ou des données du web sémantique.

MRAR est basé sur Apriori, cependant il n'utilise plus le concept de transaction mais de chaîne d'items ("ItemChain"). Il cherche d'abord les 1-ItemChains fréquentes puis les 2-ItemChains fréquentes jusqu'aux k-ItemChains. À partir de k-ItemChains fréquentes (avec $k > 2$) l'algorithme construit les règles d'association multi-relations. Une chaîne d'items est constituée par les points communs (entités et relations) entre deux entités jusqu'à une entité qui n'a plus de liens sortant (endpoint). Une chaîne d'items est construite par rapport à un endpoint et deux entités.

CHAPITRE IV

MÉTHODE D'APPARIEMENT PAR GESTION EXPLICITE DES TÂCHES DE RAFFINEMENT

4.1 Introduction

Dans ce chapitre nous allons revenir sur les limitations de la méthode d'appariement de motifs/séquences dans xPMiner (Adda *et al.*, 2006; Adda *et al.*, 2007) puis proposer un nouvel algorithme plus efficace. Cette nouvelle approche a été publiée dans (Halioui *et al.*, 2017) et fait partie du travail de thèse (Halioui, 2017). Cette nouvelle approche de la mise en correspondance des motifs recherchés et des séquences utilisateurs, devenue le goulot d'étranglement lors de l'exécution de la fouille, nous permet ainsi être en mesure d'augmenter grandement les performances de l'outil.

Dans un premier temps, nous allons décrire le fonctionnement à haut niveau de xPMiner ce qui va permettre d'introduire et illustrer les principes de l'appariement entre un motif séquentiel de données liées et une séquence utilisateur. Nous présenterons les limites de l'approche xPMiner ainsi que nos intuitions pour l'améliorer. Ainsi, de ces intuitions découlera le fonctionnement de notre nouvelle approche.

Avant d'aller plus loin, nous allons fixer un cadre pour tous les exemples proposés par la suite. Pour cela, nous proposons une sequence utilisateur s type composée d'instances (Figure 4.1) ainsi que l'ontologie complète Ω qui la supporte (Figures 4.2 et 4.3).

Figure 4.1: Exemple de séquence utilisateur (instances)

Cet exemple utilise le thème du Tourisme. L'ontologie Ω offre des classes et des instances permettant de représenter une visite au sein d'un portail d'e-tourisme (ex : Ville, Hôtel, etc.). Les propriétés permettent de marquer l'appartenance d'un lieu touristique à une ville ainsi que d'indiquer un potentiel rabais proposé pour la réservation d'un hôtel et d'une visite sur un lieu touristique.

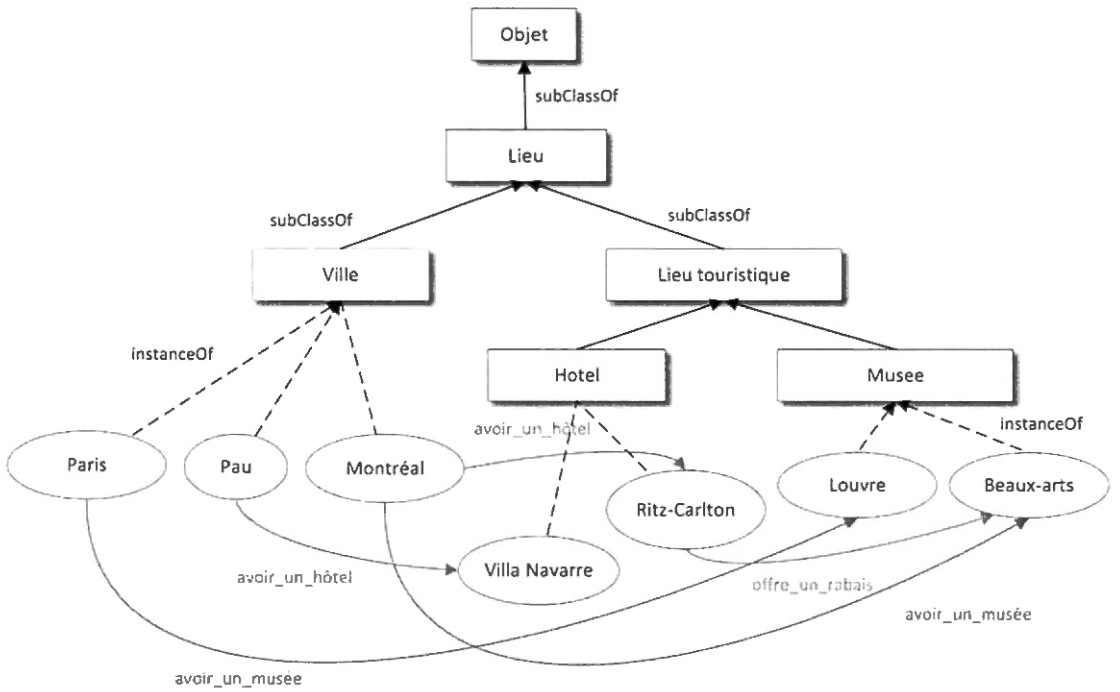

Figure 4.2: Ontologie Ω avec ses classes, instances et triplets

Dans la figure 4.2 les classes sont représentées par des rectangles tandis que les instances sont sous la forme d'ellipses. Les instances sont liées à leurs classes respectives par une flèche en pointilles et liées entre elles via des flèches de couleurs, chaque couleur représentant un type de propriété.

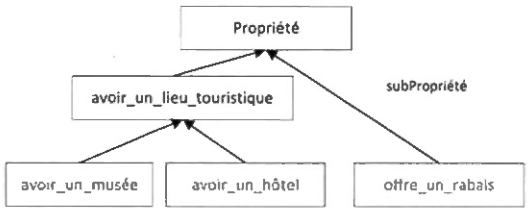

Figure 4.3: Propriétés de l'ontologie Ω

Tout d'abord, via Ω, cette sequence s va être transformée en séquence de classes. Autrement dit, les instances qui composent s vont être remplacées par leurs classes. Ensuite, les triplets vont être conservés mais vont faire intervenir les classes plutôt que les instances. Cette transformation est visible aux figures 4.4 et 4.5.

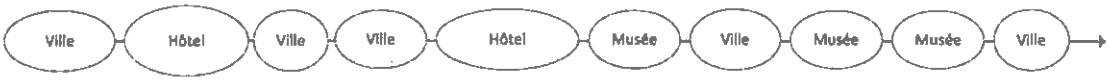

Figure 4.4: Exemple de séquence utilisateur (classes)

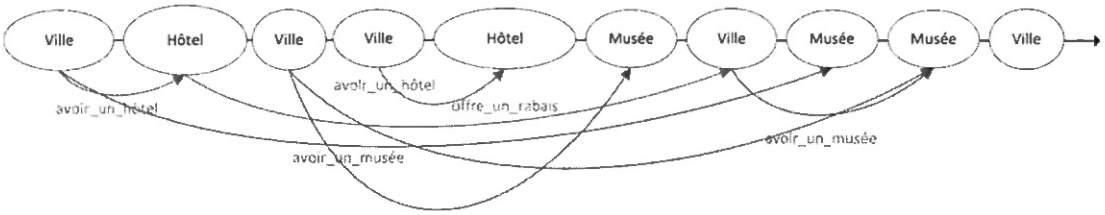

Figure 4.5: Exemple de séquence utilisateur enrichie (classes et triplets)

Cette sequence utilisateur enrichie (figure 4.5) avec les classes et les triplets va être réutilisée dans toute la suite de ce travail.

4.1.1 xPMiner : Fonctionnement global

xPMiner utilise l'approche communément appelée "générer et tester", c'est-à-dire qu'il génère un ensemble de motifs candidats puis chacun d'entre eux est testé. En fonction de son support, ce motif est conservé ou pas. À l'origine, ce choix a été fait car il permet de libérer de la mémoire après le traitement complet d'une branche pour les traitements

suivants. À l'instar de SPAM (Ayres *et al.*, 2002) et de SPADE (Zaki, 2001), la fouille de motifs séquentiels réalisée par le framework xPMiner est une fouille en profondeur d'abord. Au départ, le premier candidat est un motif vide que l'on va étendre au fur et à mesure de notre exploration. C'est-à-dire que nous allons tester les extensions des motifs et conserver les extensions fréquentes. Ensuite, pour chaque motif fréquent, nous relançons une étape de "générer et tester", ceci tant que des extensions sont fréquentes.

Cependant, comme nous l'avons indiqué précédemment, nos motifs séquentiels de données liées ne sont pas uniquement composés d'objets, mais aussi de généralisations de ces objets et de relations entre ces objets. C'est la raison pour laquelle notre espace de motifs contient quatre dimensions au lieu d'une seule : les classes racines, la spécialisation de ces classes, les propriétés dont ces classes sont domaines et codomaines et les spécialisations de ces propriétés. Comme c'est le cas dans (Adda *et al.*, 2007), les extensions de motifs sont calculées à l'aide de quatre opérations canoniques : *AddConcept*, *AddRelation*, *SpeConcept*, *SpeRelation*.

4.1.2 xPMiner : Principes d'appariement motif-séquence

Pour être en mesure d'appareiller un motif avec une séquence utilisateur, il est nécessaire de trouver une correspondance entre toutes les classes présentes dans le motif, toutes les propriétés présentes dans le motif ainsi que tous les triplets formés par les classes et le propriétés dans le motif. En outre, l'ordre d'apparition des classes et des triplets doit être préservé dans la séquence utilisateur.

La procédure d'appariement d'une classe revient à rechercher dans toute la sequence quels sont les classes qui lui correspondent. L'opération permettant d'affirmer ou infirmer la correspondance n'est pas une égalité mais plutôt une relation de subsomption, c'est-à-dire qu'il ne faut pas tester l'égalité entre deux caractères mais une relation d'inclusion $\lhd$ dans une hiérarchie de classes. En d'autres termes, nous cherchons à savoir

si la classe à tester est une sous-classe non-stricte de la classe dans le motif. Pour cela, nous disposons d'une opération $isConceptEqualOrDescendant(a, b)$ où a représente la classe à tester et b représente la classe dans le motif, qui fait appel à notre représentation de l'ontologie Ω. Cette opération est la raison pour laquelle nous devons impérativement disposer d'une représentation optimisée de l'ontologie étant donné qu'il s'agit de l'opération à la fois la plus atomique et fréquente de notre algorithme.

En ce qui concerne les propriétés, il s'agit du même problème si ce n'est qu'elles se comportent comme des contraintes sur les classes qu'elles sont supposées lier. C'est à dire que lorsqu'une classe dans la séquence correspond à la classe dans le motif, il faut aussi s'assurer que les instances qui correspondent aux positions actuellement identifiées possèdent les propriétés imposées par le motif, c'est-à-dire en tant que sujet ou objet d'un triplet.

Néanmoins, alors que l'appariement d'une classe ne demande qu'un seule opération, l'appariement d'une relation requiert au moins quatre opérations distinctes :

- trouver un triplet dont la propriété corresponde,
- trouver un triplet dont le domaine corresponde,
- trouver un triplet dont le codomaine corresponde,
- s'assurer qu'il s'agit du même triplet dans les trois cas,

En effet, ceci correspond à la définition d'un triplet, composé de deux ressources qui sont ici des classes jouant respectivement le rôle de domaine et codomaine de la propriété.

Spécifiquement, il s'agit ici d'une recherche de morphisme de graphe dirigé acyclique. En effet, les motifs ainsi que les séquences s'apparentent à des graphes dirigés acycliques dans la mesure où ils sont composés principalement par une chaîne orientée (classes) jouant le rôle de colonne vertébrale du motif, sur laquelle prennent racine les triplets eux-mêmes orientés (propriétés). L'orientation de ces triplets est dans le même

sens que l'orientation de la chaîne et par conséquent cette structure ne comporte aucun cycle. Ainsi, l'ensemble formé par la combinaison de la chaine des classes et des triplets enracinés sur celle-ci forme une graphe orienté acyclique.

Si un des concepts du motif n'est pas appareillé, alors l'algorithme en conclut que l'appariement de la sequence actuelle avec le motif courant est impossible et s'arrête là. Dans le cas contraire, l'algorithme va pour chaque propriété du motif tenter de trouver un appariement avec les propriétés de la sequence.

En outre, nous devons, comme nous l'avons fait précédemment pour les classes, nous assurer que la relation en question soit bien la propriété ou une sous-propriété de la propriété indiquée dans le motif. Pour cela, nous devons à nouveau nous référer à la représentation de l'ontologie et nous assurer que $isPropertyEqualOrDescendant(a, b)$, avec a la propriété à tester et b la propriété dans le motif, est vrai. Éventuellement, si toutes ces conditions sont satisfaites pour tous les classes et toutes les propriétés d'un motif sur une séquence, nous sommes en mesure d'affirmer que notre séquence s'appareille avec le motif.

xPMiner effectue cet appariement en deux temps. En effet, dans un premier temps il recherche un appariement pour toutes les classes puis dans un deuxième temps, il recherche un appariement pour les propriétés. Ainsi, l'appariement de xPMiner reste extrêmement coûteux et par conséquent lent à l'exécution.

En outre, la stratégie de parcours employée dans xPMiner permet de réduire grandement le nombre de séquences à traiter avec chaque motif candidat mais ne réduit que partiellement l'espace de recherche des motifs au vu des techniques mises en avant dans le chapitre 2. De plus, même si xPMiner réduit l'espace de recherche des séquences en descendant, il ne réduit pas l'espace à l'intérieur de chaque séquence. En effet, xPMiner recommence complètement le procédé d'appariement pour chaque séquence à chaque nouveau motif. Et selon nous, c'est là sa limitation majeure.

À partir de là nous avons axé notre travail sur la recherche d'une représentation du motif à appareiller qui nous permet de faire un minimum d'opérations de recherche, c'est à dire éviter les opérations inutiles. Nous avons donc opté pour la construction d'une nouvelle représentation de notre motif qui tient compte des étapes à effectuer pour être en mesure de trouver une correspondance. Cette structure est assimilable à une séquence de taches représentées chacune par un bloc et correspondant à l'application de nos opérateurs canoniques *AddConcept*, *SpeConcept*, *AddRelation*, *SpeRelation*.

4.2 OntoPattern : Un nouveau mécanisme d'appariement

L'appariement des motifs candidats fils construits à partir d'un motif fréquent ne nécessite pas obligatoirement de recommencer complètement le processus. En effet, il est possible de réutiliser l'appariement du motif parent comme base et ainsi réduire le nombre d'opérations nécessaires à un appariement. L'intuition est qu'un motif candidat est en tout point identique à son parent fréquent, à l'exception de l'ajout fait par l'application d'un des opérateurs canoniques. Il est logique que son appariement soit aussi très proche de celui de son parent.

Dans (Halioui *et al.*, 2017), nous avons conçu une nouvelle méthode d'appariement entre les motifs et les séquences qui tire parti de ces observations. Ainsi, pour être en mesure d'accélérer le processus d'appariement par rapport à xPMiner il est nécessaire de tenir compte des résultats de l'appariement du motif parent du motif courant pour effectuer notre correspondance. C'est-à-dire, travailler par extension de l'appariement du motif parent plutôt que de reconstruire une structure d'appariement en entier. En effet, l'approche par niveaux k nous permet de garantir qu'un seul élément du motif va être modifié ou ajouté d'un niveau sur l'autre.

Intuitivement, nous considérons le processus d'appariement comme une suite d'opérations chacune liée à son contexte (ou niveau). Ainsi, pour appareiller un motif candidat,

Figure 4.6: Exemple de motif avec sa structure d'appariement complétée

il est possible de, soit reconstruire l'appariement depuis le début ou, de copier l'appariement de son parent puis appliquer une séquence de taches grandement réduite correspondant seulement à l'appariement de la différence entre le motif candidat et son parent. Dès lors, faire le choix de la copie de l'appariement précédent et ne travailler que sur les différences entre le motif et son parent semble être beaucoup plus judicieux.

Toutefois, repenser l'appariement en termes de taches successives complémentaires nous a forcé à découper nos opérations d'appariement en quatre primitives de bas niveau : $trouveClasse$, $trouveDomaine$, $trouveCodomaine$ et $verifieTriplet$. Comme leur nom l'indique, elles vont être respectivement chargées de trouver une prochaine solution lors d'un appariement de classe, lors de la recherche du domaine pour une propriété, lors de la recherche du codomaine pour une propriété et de la vérification que la solution pour le domaine et pour le codomaine font bien partie du même triplet.

L'utilisation de chacune de ces primitives va être représentée sous la forme d'un bloc au sein d'une structure représentant le processus d'appariement (figure 4.6). Cette structure doublement chainée représente le parcours de l'appariement, chaque bloc correspondant à une des quatre nouvelles primitives ainsi qu'aux résultats de celle-ci. Un curseur permettant de marquer l'avancement du travail d'appariement est présent au sein de la structure et les résultats de chaque primitive sont conserves au sein de chaque bloc, ceci permettant facilement de poursuivre le travail d'appariement. Une instance de cette structure est construite pour l'appariement de chaque sequence (sid). En cas d'appariement fructueux, elle est conservée pour être réutilisée pour l'appariement des motifs fils si celui-ci est établi comme fréquent. Dans le cas contraire, elle est immé-

diatement détruite.

Toutefois, l'inconvénient principal de cette approche est qu'elle introduit le principe de "retour en arriere". En effet, en rajoutant une propriété au motif, il est possible que les solutions déterminées pour ses classes "domaine" et "codomaine" soient invalides. Autrement dit, qu'il n'existe pas un triplet supportant cette nouvelle contrainte entre les deux instances utilisées comme solution. Dans ce cas, il est nécessaire de "défaire" les appariement précédents pour retrouver une nouvelle solutions à la fois pour la classe "domaine" mais aussi pour la classe "codomaine". La structure n'est pas défaite à proprement parler, mais il est nécessaire de suivre des liens pour revenir à l'étape/bloc à modifier. Étant donne qu'il faut revenir sur des appariement passés, nous avons baptise ce phénomène le "retour en arrière". Un exemple de retour en arrière est visible à la figure 4.7. Sur cette figure, les "retours en arrière" possibles sont représentés sous forme de flèches grises en pointillés.

Ces retours en arrière recherchent des solutions à chacune des étapes passées du processus d'appariement tant qu'ils n'aboutissent pas à une solution ou qu'ils ne déterminent pas qu'aucune solution n'est possible. Par exemple, il est aisé de déterminer qu'il n'existe plus aucun appariement possible pour une classe à droite de la solution actuelle. Mais il est beaucoup plus difficile de déterminer la même chose pour une propriété étant donné qu'elles peuvent s'étendre sur une grande partie d'une séquence. En effet, dans le cas où plusieurs solutions pour une propriété se chevauchent, c'est-à-dire que le codomaine de la première solution se situe à droite du domaine de la seconde solution, il est ardu d'anticiper que la première solution ne permettra pas d'appareiller une classe après le codomaine car trop à droite. Tandis qu'opter pour la seconde solution dont le codomaine est plus à gauche aurait permis de trouver un appariement. Ainsi, opter pour la première solution aurait conduit à calculer un retour en arrière lors de l'operation *AddConcept* suivante. Il faut maintenant imaginer que ce cas peut se produire avec un motif de grande taille comportant plusieurs propriétés. Ceci peut

entrainer la recherche de nouvelles solutions pour une grande partie du motif tout en n'ayant aucune garantie de réussite.

En résumé, ce nouvel algorithme d'appariement fait le choix de ne calculer qu'une seule solution par extension puis, au prix de retours en arrière coûteux, tente de défaire une partie des appariements précédents pour trouver de nouvelles solutions dans la partie inexplorée (à droite) de la séquence.

Une solution plus judicieuse ne permettrait pas de retour en arrière explicite et serait en mesure d'indiquer si il existe au moins une prochaine solution avant d'entreprendre des ré-appariements. Nous souhaiterions alors disposer d'une approche efficace en mesure de calculer toutes les solutions, ou du moins une partie, pour un nouvel opérateur.

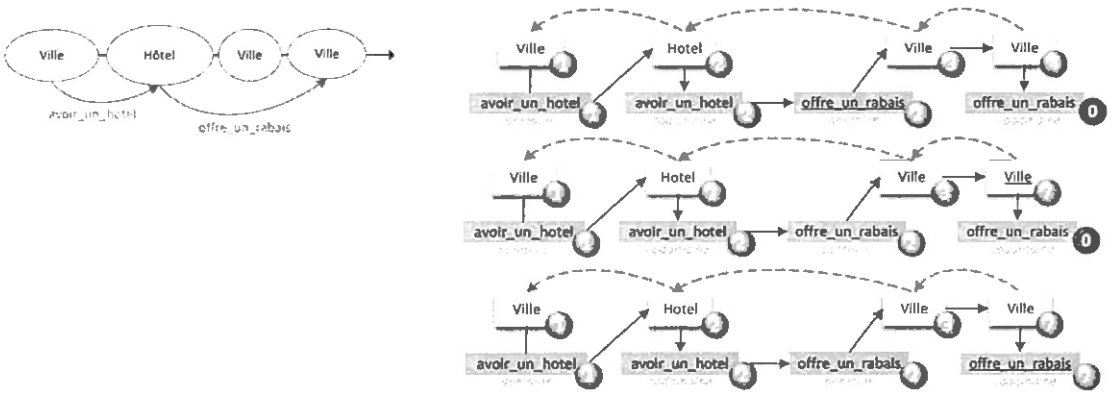

Figure 4.7: Exemple d'appariement avec "retour en arrière"

Quoi qu'il en soit, mis à part les inconvénients dûs aux "retours en arrière", le nouveau mécanisme d'appariement apporte des gains non-négligeables en termes de performances. Avant de proposer une toute nouvelle approche d'appariement nous allons d'abord évaluer ce travail en le confrontant avec xPMiner.

4.3 Comparaison

Dans cette partie nous allons comparer les performances de l'ancienne version de l'algorithme et de la nouvelle afin d'observer les gains de performance obtenus avec la nouvelle technique d'appariement. Pour ce faire, nous allons proposer une comparaison de plusieurs mesures que nous avons effectuées sur les quatre versions de l'algorithme, xPMiner-B qui correspond à notre outil tel qu'il était lors de notre prise en main de celui-ci, xPMiner-C qui correspond à son état après avoir amélioré la représentation interne de l'ontologie, OP-2014 qui correspond à son état avec le nouvel algorithme d'appariement et OP-2014S qui correspond à son état avec le nouvel algorithme rappariement en conservant les résultats entre les niveaux. En d'autres termes, OP-2014S correspond à l'ensemble des améliorations présentées dans la section précédente. OP-2014 correspond au même algorithme mais qui reconstruit complètement les structures d'appariement entre chaque extension de motif. Autrement dit, une version intermédiaire entre l'ancien xPMiner et la nouvelle approche OP-2014S. Les résultats de cette expérimentation sont visibles sur la figure 4.8.

Les mesures sur XPMiner-B et XPMiner-C ont été réalisées sous Mac OSX avec un processeur Intel Core i7 cadencé à 3.40Gz, 16Go de RAM 64bits. Le reste des expérimentations ont été réalisées sous Windows 8.1 avec un processeur Intel Core i7 cadencé à 2.20Gz, ce qui est considérablement inférieur à l'équipement dont nous disposions pour les premières expérimentations. La différence de performance entre les deux séries de mesures est d'autant plus importante. Notre objectif dans cette comparaison est simplement de montrer qu'avec la nouvelle version de l'algorithme, même avec un équipement inférieur, les temps d'exécution sont bien meilleurs. Sur cette comparaison, nous voyons bien la différence au niveau des performances, dès que l'on réduit le seuil, autrement dit, dès que l'on augmente le nombre de résultats découverts de façon importante, la différence est de plus en plus importante. Par exemple, avec

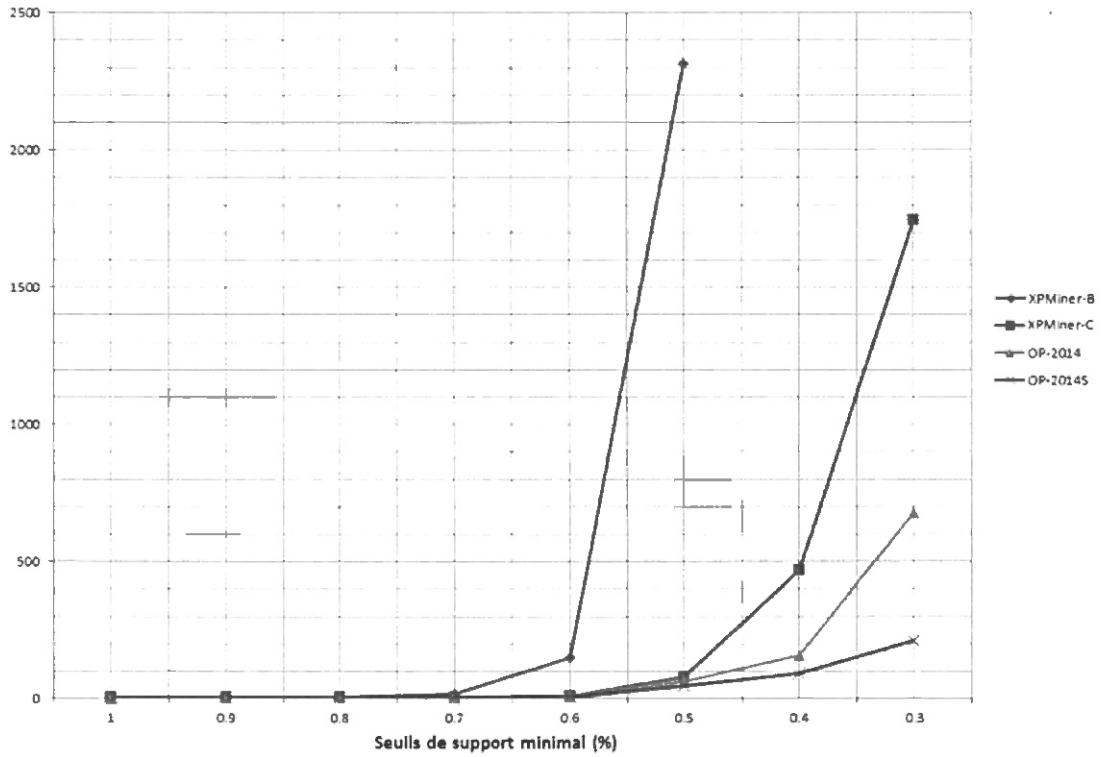

Figure 4.8: Expérimentations comparatives avec xPMiner

un seuil de 60%, la nouvelle version est plus de 20 fois plus rapide entre XPMiner-B et XPMiner-C. Lorsque nous avons effectué les tests sur notre poste de travail, nous avons atteint des performances environ sept fois plus rapides avec un seuil de 70%. En effet, 12.416s avec la version optimisée et 1.34s avec la version originale. A partir de là, nous en avions conclu que le projet d'amélioration était un succès. Mais, bien évidemment, d'autres optimisations restaient encore possibles afin d'accroitre encore les performances.

Entre XPMiner-C et OP-2014, nous pouvons constater pour un seuil de 50% que nous avons un gain 25% supplémentaire. Pour la même valeur de seuil, si l'on considère XPMiner-C et OP-2014S, nous avons un temps d'exécution réduit de moitié, ce qui est une avancée non négligeable. Toujours sur ce seuil de 50%, si l'on considère XPMiner-B et OP-2014S, le temps d'exécution à été réduit d'un facteur de cinquante. Il faut

remarquer que si nous avions effectué des mesures sur des supports moins élevés, les différences auraient été encore plus prononcées.

Néanmoins, sur des grands seuils, peu de différences se font sentir. Sur XPMiner, nous avons optimisé les opérations les plus atomiques liées à l'ontologie (test d'instanciation, compatibilité des propriétés) appelées en permanence. Nous avons donc gagné peu de temps mais finalement sur des milliers d'appels nous gagnons beaucoup de temps. En revanche, dans le reste des implémentations nous avons travaillé la stratégie de parcours, c'est-à-dire que nous allons plus rapidement à l'essentiel. De plus dans la dernière version, nous avons conservé les informations des opérations précédentes donc plus le traitement est long, plus nous gagnons de temps.

Maintenant que nous avons montré le gain de performance de notre nouvelle version, nous allons montrer le coût relatif à la fouille de chaque niveau de l'ontologie. Malgré les gains annoncés au dessus, nous ne sommes pas encore en mesure de descendre sur des supports bas (inférieurs à 20%) et comme le montre la figure 4.9, un seuil minimal de 30% demande un temps de calcul de plus d'une heure. En effet, comme nous l'avons indiqué en début de ce travail, selon nous le problème majeur n'est pas dans la méthode d'appariement mais dans la taille de l'espace de recherche des motifs. Bien évidement les deux sont liées et grâce à nos améliorations nous maintenant mieux mettre en évidence le coût de cet espace de recherche en faisant varier le niveau d'entrée dans l'ontologie. Si l'algorithme commence par fouiller les classes à partir d'une profondeur p alors l'espace de recherche n'inclut pas toutes les classes situées au dessus et diminue d'autant. Ainsi, en faisant varier le niveau p nous allons montrer les variations de temps d'exécution de l'algorithme à support minimal constant. Cependant, nous n'allons faire varier que le niveau pour la hiérarchie des classes puisque le nombre de celles-ci est très supérieur au nombre de propriétés. Nous faisons figurer avec cette première figure une seconde (figure 4.10) qui indique le nombre de motifs fréquents restants avec chaque seuil de support et chaque niveau initial de l'ontologie. Ce que montrent cette paire de

figures est que comme nous l'avons annoncé le coût de calcul est directement liée au nombre d'éléments présents dans l'ontologie et plus particulièrement à la hauteur des hiérarchies. Plus on élimine de niveaux dans la hiérarchie des classes, plus l'algorithme performe rapidement. Cependant, de plus en plus de motifs sont mis de coté. Le gain entre une profondeur de départ à la racine et une profondeur de départ au niveau en dessous est particulièrement frappant.

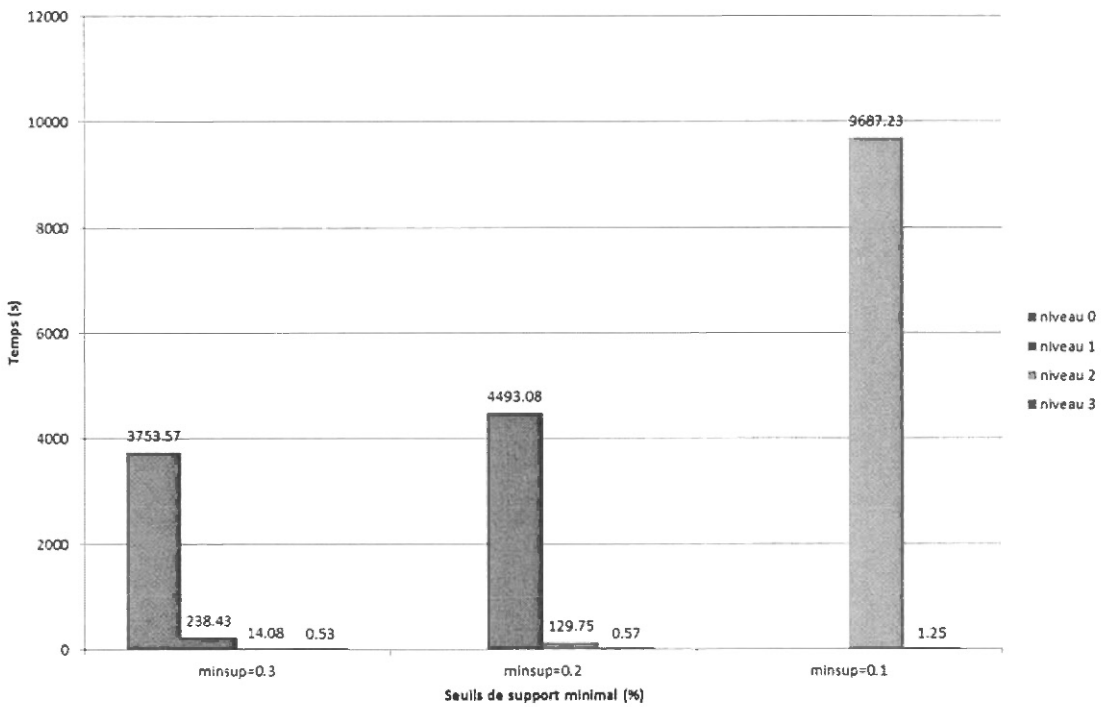

Figure 4.9: Coûts du calcul par rapport à la profondeur de l'ontologie

Ces résultats montrent bien que les abstractions successives de l'ontologie rajoutent des items à considérer qui augmentent la quantité de combinaison possibles et par conséquent la taille de l'espace de recherche des motifs. Ceci renforce l'idée selon laquelle l'élagage de l'espace de recherche des motifs est la tâche la plus critique de tout le processus de fouille.

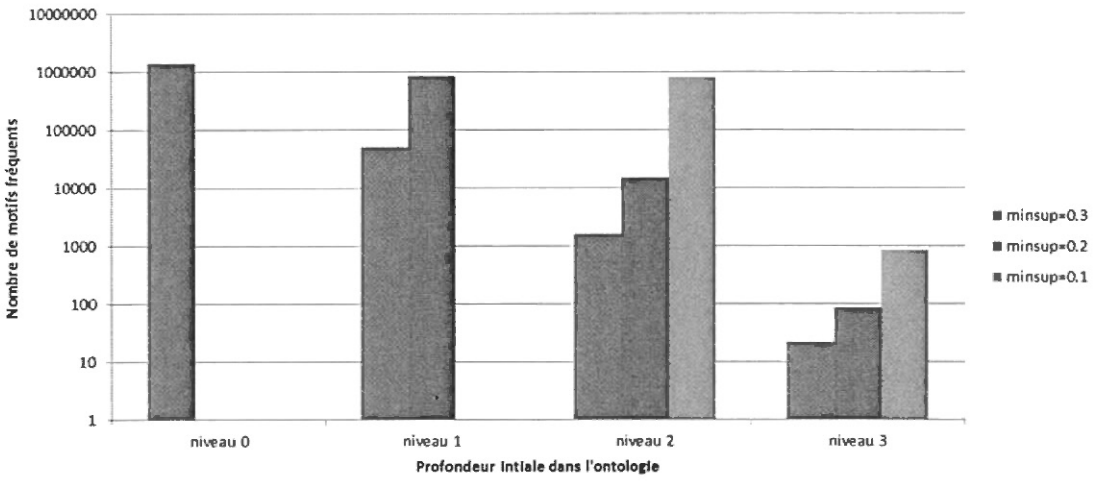

Figure 4.10: Nombre de motifs par rapport à la profondeur de l'ontologie

4.4 Conclusion

Malgré l'amélioration substantielle, le nouveau mécanisme d'appariement présenté reste peu exploitable dans le contexte d'une navigation sur le web traditionnelle étant donné qu'il faut tout d'abord indiquer dans notre ontologie quelle page est liée à quels concepts. Sur un plan plus large, la fouille de graphes est un domaine encore peu exploité relativement à ce qu'elle pourrait apporter. C'est pourquoi de nouvelles techniques de fouille adaptées à ces motifs sont à développer.

Notre nouvelle solution permet de solutionner le problème d'appariement de façon relativement compréhensible et intuitive. Il est aisé de constater que cette nouvelle version de l'outil offre des performances bien supérieures à l'ancienne version. Néanmoins, ces performances sont loin d'être satisfaisantes et selon nous il est possible de faire beaucoup mieux en appliquant les techniques les plus efficaces provenant de la fouille de motifs séquentiels. À cette fin, le chapitre suivant va proposer une comparaison des meilleurs algorithmes de fouille séquentiels puis tirer des conclusions sur quelles sont les approches les plus efficaces. Forts de ces remarques, nous exploiterons ces techniques au sein d'une nouvelle version de nos mécanismes d'appariement.

CHAPITRE V

NOUVELLE MÉTHODE DE FOUILLE BASÉE SUR LE CODAGE VERTICAL ET LE PRÉ-ÉLAGAGE

Dans ce dernier chapitre, nous allons dans un premier temps présenter nos expérimentations sur les différents algorithmes de fouille de motifs séquentiels existants en 2016. Dans une deuxième partie, nous allons proposer une nouvelle version beaucoup plus efficace de notre algorithme de fouille de motifs séquentiels de données liées conçue en tirant parti des stratégies efficaces présentes dans les algorithmes de fouille de motifs séquentiels.

5.1 Expérimentations sur les outils de fouille de motifs séquentiels

Maintenant, la question est dans quelle mesure nous sommes certains de la véracité de ces observations. Pour cela nous allons devoir évaluer la performance réelle de ces algorithmes à travers une expérimentation et ainsi compléter les informations manquantes dans la figure 5.1 ci-après.

Cette figure montre les performances supposées entre les algorithmes tel qu'affiché par les différents auteurs après notre état de l'art. Chaque flèche pleine représente la "supériorité" d'une approche sur une autre selon une publication. En effet, chaque publication est représentée ici par une couleur spécifique. Les flèches en pointilles indiquent des suppositions de notre part, puisqu'aucune publication n'existe pour comparer di-

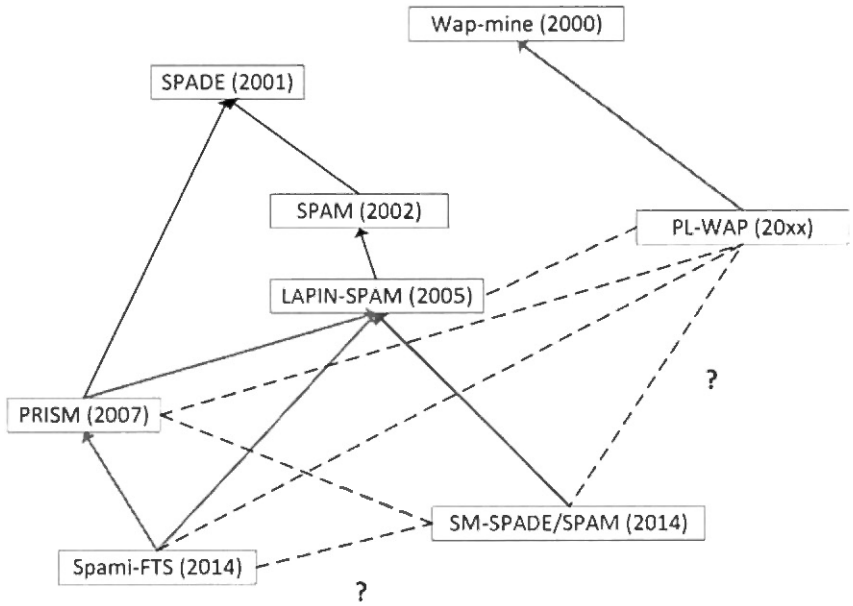

Figure 5.1: Synthèse des performances entre algorithmes de fouille de séquences

rectement les approches. Ainsi, les deux approches les plus efficaces semblent être CM-SPADE/SPAM (Fournier-Viger *et al.*, 2014a), PL-WAP (Lu et Ezeife, 2003) et Spami-FTS (Febrer-Hernández *et al.*, 2014). Notre objectif ici va être de vérifier ces hypothèses puisqu'il n'existe à notre connaissance aucun travail récent qui s'en charge.

Dans le paragraphe suivant, nous allons indiquer de quelles implémentations nous disposons pour réaliser nos expérimentations. La majorité des implémentations originales d'algorithmes de fouille de motifs (séquentiels) sont réalisées en C/C++ tandis que nos travaux sont implémentes majoritairement en Java. En outre, certaines implémentations proviennent du projet SPMF (Fournier-Viger *et al.*, 2014b). Celles-ci étant aussi implémentées en Java, elles serviront de comparaisons en priorité. Il va de soi que des implémentations en Java ne sont pas en mesure de rivaliser avec des implémentations en langage C, que ce soit au niveau des temps d'exécution ou de la consommation en mémoire. Cependant, il s'agit des implémentations telles qu'elles ont été pensées par les concepteurs des différents algorithmes. Nous tenons donc à les inclure en tant

qu'objectifs à atteindre plutôt qu'en tant que compétiteurs directs.

Nous utilisons les implémentations suivantes provenant de SPMF : SPAM, PrefixSpan, SPADE, LAPIN, CM-SPAM et CM-SPADE. Wap-mine et PL-WAP proviennent de leurs auteurs et intègrent aussi une version de GSP en guise de comparaison. Il s'agit d'implémentations en C++. FLWAP et FOF proviennent aussi de leurs auteurs et sont aussi des implémentations C++. La version de SPAM d'origine en C est utilisée.

Les versions d'origine de SPADE et PRISM ne compilent plus, le code ayant plus de 10 ans. Nous avons cependant opte pour l'utilisation de SPADE dans R qui correspond au code original en C intégré au sein du framework R via le package "aRuleSequences". En réalité, il s'agit ici de cSPADE mais cela correspond à SPADE si nous n'utilisons aucune contrainte. Les auteurs de LAPIN, LAPIN-SPAM et PAID nous ont fourni leurs implémentations mais tout comme pour celle du dessus, le code est trop ancien.

Nous ne disposons pas d'implémentations de tous les algorithmes présentés dans l'état de l'art mais à nos yeux cet ensemble est suffisamment représentatif. De plus nous disposons de la majorité des algorithmes les plus récents et présentés comme les plus efficaces dans l'état de l'art. En ce qui concerne les implémentations réalisées par nos soins, nous nous sommes efforcés de réaliser des implémentations efficaces du mieux possible.

Les auteurs de Spami-FTS ont refuse de faire parvenir une implémentation de leur algorithme. Nous avons donc développé nous mêmes une version fonctionnelle et améliorée de Spami-FTS que nous désignerons par la suite par Spami-FTS++. Notre implémentation réutilise donc des idées de Spami-FTS et de FAST. Elle integre aussi de nouveaux mécanismes d'élagage a priori de l'espace de recherche sur lesquels nous ne reviendrons pas dans l'immédiat.

Nous avons écarté PRISM d'emblée puisque il ne s'agit pas d'une réelle amélioration

liée à l'algorithmique de fouille de séquences mais plutôt d'une utilisation judicieuse de pré-traitement. Les performances de PRISM sont indubitablement indexées sur celles de SPAM. Cependant, grâce à la réduction de coût en mémoire et l'accélération due au pré-calcul des opérations entre et sur les vecteurs de bits il est difficile de se prononcer sur ses performances réelles. Quoi qu'il en soit, PRISM ne modifie pas la complexité algorithmique de SPAM. Ainsi, avec des jeux de données suffisamment grands, tout comme SPAM, ses performances viendront à faiblir considérablement. De plus, comme nous l'avons indiqué dans notre état de l'art n'importe quel algorithme de fouille basé sur des vecteurs de bits pourrait exploiter les techniques présentées dans PRISM. Nous n'avons pas été en mesure de récupérer une implémentation de DISC-all donc nous ne l'inclurons pas dans nos expérimentations. Wap-mine étant toujours plus lent que PL-WAP nous l'avons écarté de nos figures.

Nous disposons de plusieurs versions distinctes de SPAM, de SPADE et de PrefixSpan. Il va être intéressant de constater et d'analyser les écarts de performances être ces versions. Cependant, elles ont été réalisés par différentes personnes dans différents langages, ainsi nos mesures vont autant refléter la différence entre chacun de ces plateformes que la différence de qualité des implémentations.

Nos expérimentations vont porter sur deux des jeux de données les plus connus et utilisés au sein des diverses publications portant sur la fouille de motifs séquentiels fréquents, à savoir SIGN et Gazelle/BMS2. Nous allons aussi utiliser deux autres jeux de données synthétiques générés avec SPMF. Nous ne nous limiterons donc pas à des jeux de données de séquences contenant des itemsets singletons uniquement. Des détails sur le contenu de chacun des jeux de données sont proposés dans la figure 5.1.

Gazelle/BMS2 [1] est un jeu de données utilisé dans la compétition KDD-CUP 2000 qui contient 77,512 séquences avec 3340 items. Il s'agit de séquences de clics web, autre-

1. http ://www.kdd.org/kdd-cup/view/kdd-cup-2000

ment dit d'itemsets singletons. Sign [2] est un jeu de données d'utterances de langage des signes qui contient 730 séquences d'itemsets singletons.

Jeu de données	Nbr. séq.	Moyenne trx/séq.	Nbr d'items	Nbr d'items/trx
Gazelle/BMS2	77000	4,62	3340	1
Sign	730	51,9	264	1
Synth6	1000	25	250	2
Synth8	1000	25	250	10

Tableau 5.1: Détail des jeux de données utilisés

Nos quatre jeux de données nous permettent de couvrir l'essentiel des extrémités du spectre des cas possibles : beaucoup de courtes séquences d'événements, un petit nombre de grandes séquences d'événements, des séquences de paires et des séquences de longues transactions.

La première expérimentation (figure 5.2) porte sur Gazelle et représente le cas typique de fouille de séquences, c'est à dire un grand nombre de courtes séquences utilisateurs comportant des itemsets singletons (événements).

Les trois meilleures approches ici sont CM-SPAM, cSPADE, FL-WAP et PrefixSpan. cSPADE est légèrement plus efficace que CM-SPAM mais il est tout à fait normal qu'une implémentation C soit la plus performante du lot. Toutefois, cela ne signifie pas que cSPADE est généralement plus efficace que CM-SPAM. En effet, la version de SPADE Java se comporte bien puisqu'elle bat LAPIN jusqu'à un certain stade, mais finit par exploser. cSPADE semble très efficace sur des séquences creuses alors que la version dans SPMF se comporte beaucoup moins bien. CM-SPADE et LAPIN réagissent aussi favorablement, cependant sur le plus petit seuil de support, leurs perfor-

2. http ://cs-people.bu.edu/panagpap/Research/asl_mining.htm

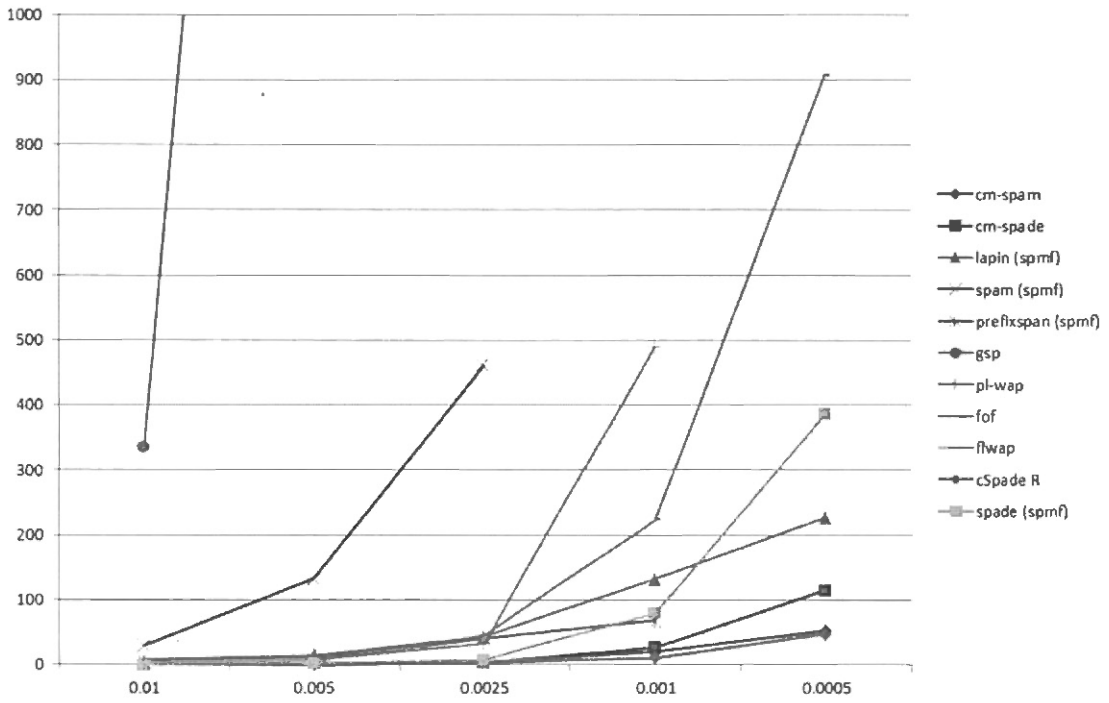

Figure 5.2: Comparaison des approches sur Gazelle/BMS2

mances commencent à s'écarter de SPAM et cSPADE.

Tout d'abord la performance de SPAM (spmf) est surprenante. En effet, celui-ci est de loin le moins efficace mis à part GSP. La version d'origine de SPAM n'est pas capable de traiter ce jeu de donnes (limite de 64 transactions par séquence). Évidement, GSP étant leur ancêtre à tous, il est celui qui se comporte le moins bien et explose très rapidement. Mis à part GSP, les premiers à exploser sont dans l'ordre SPAM, PL-WAP puis FOF. Il est tout à fait normal que FOF batte PL-WAP puisque celui-ci est sont ancêtre. Cependant, FLWAP très bon aussi, mais seulement par rapport à PL-WAP et FOF. Nous ne pouvons donc pas conclure que la famille d'algorithmes issus de WAP-mine est efficace.

Concernant SPAM, ce jeu de données nous permet de bien constater la différence entre SPAM et SPADE qui bien qu'ils partagent le même arbre lexicographique lors de l'ex-

ploration des motifs, leur façon d'élaguer l'espace de recherche d'un candidat et de générer les candidats sont fondamentalement différentes et avantagent SPADE. En effet, SPAM (en plus de transformer le vecteur de bits) intersecte les vecteurs de bits d'un motif fréquent avec le vecteur d'un item alors que SPADE intersecte le vecteur d'un motif fréquent avec le vecteur du motif fréquent qui localement utilise l'item en question. L'intersection calculée par SPADE est plus simple à calculer puisque le second vecteur comporte beaucoup moins d'entrées. L'autre aspect concerne l'espace de recherche d'un candidat : SPAM travaille sur toutes les séquences de la base alors que SPADE ne travaille que sur l'intersection des séquences (*sid*) supportant les parents du candidat. De plus, SPADE effectue un élagage à priori avec cette intersection ce que ne fait pas SPAM. Ce qui est plus surprenant est que cette tendance s'inverse dans CM-SPADE et CM-SPAM. Ceci est certainement dû à une subtilité d'implémentation au sein de SPMF.

Les performances de PrefixSpan sont relativement surprenantes puisqu'au support le plus bas, il arrive a battre tous les autres algorithmes exceptes cSPADE et CM-SPAM. Notre supposition est que cela est dû à la pseudo-projection de PrefixSpan qui permet de simplement maintenir des curseurs sur les bases projetées calculées au début plutôt que de devoir tout copier à chaque nouveau fréquent.

Nous ferons remarquer que ces résultats ne correspondent que très partiellement à ceux avancés sur le site de SPMF [3] et publies dans (Fournier-Viger *et al.*, 2014a). Quoi qu'il en soit, ces résultats ne semblent pas suffisants pour conclure que les techniques verticales et d'élagage précoce sont plus efficaces que les autres sur de longues séquences creuses.

La seconde de nos expérimentations (figure 5.3) utilise Sign, elle cible des données beaucoup plus denses et plus longues avec moins de volume mais toutefois plus difficile

3. http ://www.philippe-fournier-viger.com/spmf/index.php ?link=performance.php

à fouiller. Ce jeu comporte des séquences utilisateurs tronquées après 64 transactions pour que SPAM original soit en mesure de le traiter. Ceci ne modifie que très peu de séquences. Dans cette expérience nous allons aussi utiliser une version de Spami-FTS+ développée par nos soins volontairement handicapée dans sa capacité à calculer des intersections entre les espaces des séquences des motifs mais qui en contrepartie est complétée avec une nouvelle méthode d'élagage des candidats. Il va être intéressant d'observer si disposer d'une meilleure façon de parcourir l'espace de recherche mais une méthode de calcul du support inefficace permet de rester compétitif par rapport avec la technique qui à date semble alimenter les algorithmes les plus efficaces (CM-SP).

Ici, nous n'utilisons pas GSP puisqu'il est évident qu'il est incapable de constituer un concurrent sérieux pour n'importe laquelle des implémentations utilisées ici. Nous ne faisons pas non plus figurer les performances de cSPADE sur ce dataset car il n'a pas été en mesure de terminer en dessous de quelques heures pour un support de 10%.

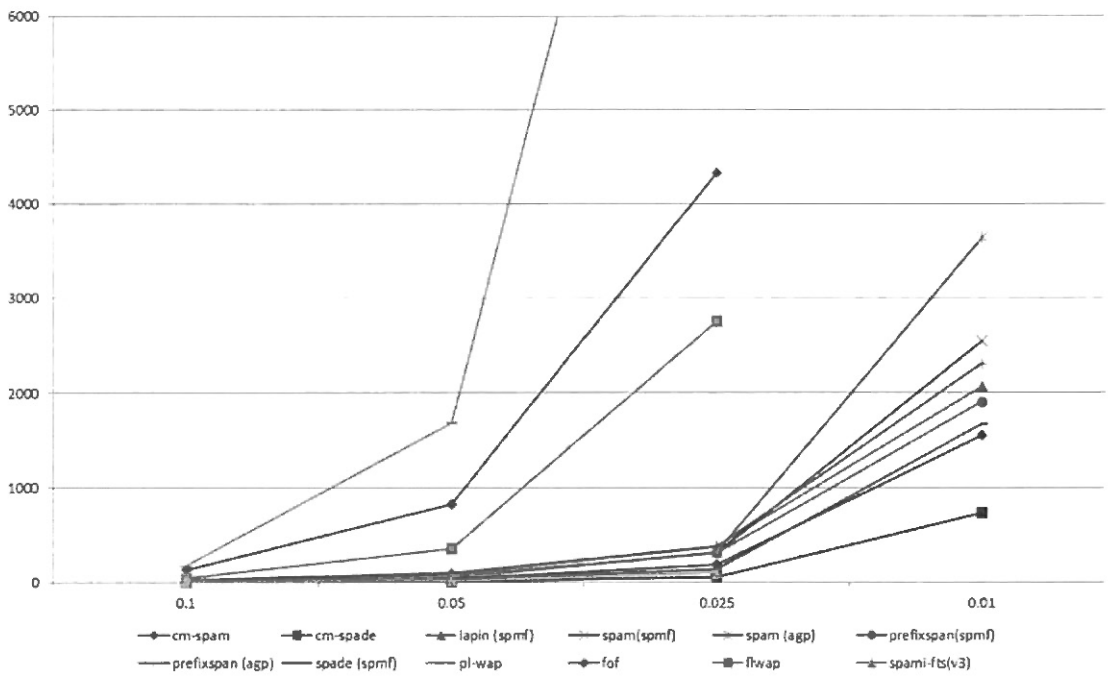

Figure 5.3: Comparaison des approches sur Sign

La première observation est que cette fois, la famille des arbres d'agrégation (PL-WAP, FL-WAP, FOF) est très clairement distancée par les approches verticales. Étrangement, FL-WAP se comporte mieux que FOF dans les deux expérimentations alors qu'il est suppose être l'ancêtre de FOF et être battu par ce dernier.

Malgré être implémentés en C++, les résultats de PL-WAP et FOF sont relativement décevants. En effet, même les implémentations en Java sont plus rapides. Selon nous, après avoir étudié les sources de ces implémentations, le manque de performances est plus liés à la qualité des implémentations qu'à la qualité des algorithmes.

Encore une fois, PrefixSpan se comporte de façon inattendue. En effet, ce dernier bat SPAM et LAPIN sur l'ensemble des seuils de supports. Cependant, sa version alternative PrefixSpan(agp) ne parvient pas à battre LAPIN. Nous pouvons supposer que sur des seuils de support plus bas, LAPIN reprend le dessus sur les deux versions de PrefixSpan.

Le quatuor gagnant ici est constitué de SPADE, CM-SPADE, CM-SPAM et Spami-FTS+. CM-SPADE et Spami-FTS+ sont clairement au dessus de toutes les autres approches. Dans cette expérience nous pouvons commencer à constater l'efficacité des approches verticales et d'élagage précoce même si les performances de LAPIN laissent à désirer. En outre, en observant les performances de notre approche par rapport à toutes les autres, la réduction de l'espace de recherche semble, comme nous en avions fait l'hypothèse, une composante au moins aussi importante, si ce n'est plus, qu'une capacité de calcul efficace du support d'un motif.

La troisième expérience (figure 5.4) porte sur un jeu de données synthétiques comportant cette fois des itemsets. En effet, ce jeu de données ne comporte que des 2-itemsets. Il va nous permettre de montrer l'impact de l'ajout d'une seconde dimension dans l'espace des motifs à explorer.

Nous n'utilisons ici que les approches ayant fait leur preuves dans les deux expérimentations précédentes et celles capables de gérer des itemsets non-singletons.

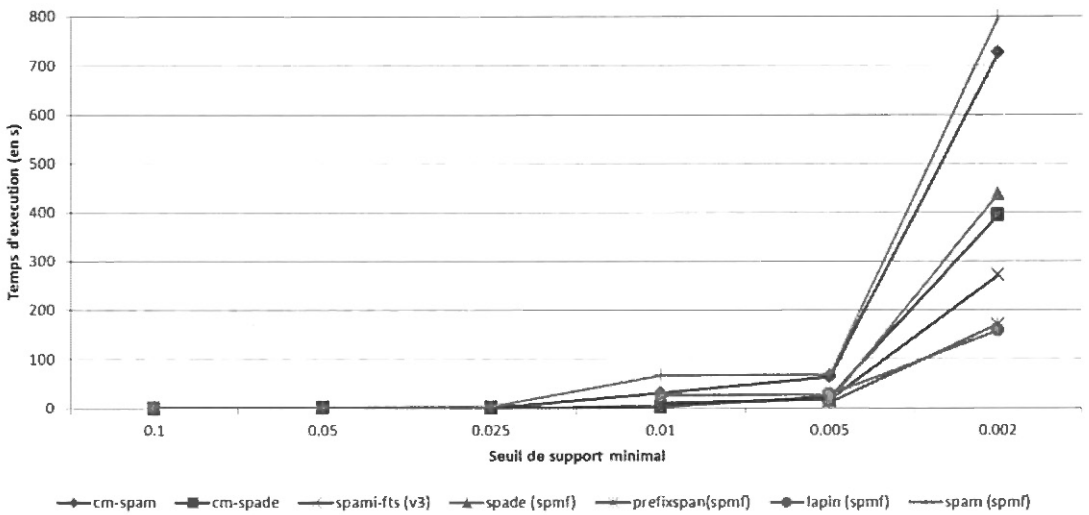

Figure 5.4: Comparaison des approches sur le premier jeu de données synthétiques

La première observation est la proximité des résultats entre les versions CM et non-CM de SPADE et SPAM. En effet, ceci s'explique par la densité du jeu de données, la grande majorité des 2-séquences possibles sont fréquentes et donc la matrice de co-occurrence n'est que très peu utile. Cependant, encore une fois, SPADE performe beaucoup mieux que SPAM.

Ici, notre version de Spami-FTS offre des performances sensiblement supérieures à CM-SPAM et CM-SPADE ainsi que leurs version d'origine dans SPMF. Cependant, LAPIN et PrefixSpan sont ici les meilleures approches. Concernant LAPIN, il ne peut s'agir que d'une question d'implémentation puisque CM-SPADE est une version algorithmiquement plus avancée de LAPIN. En toute logique deux implémentations de qualité équivalente devraient respecter le fait que CM-SPADE est plus efficace. En ce qui concerne PrefixSpan, nous supposons que encore une fois la pseudo-projection lui permet de rester très compétitif.

Ces résultats montrent que rajouter uniquement une seconde dimension dans les données n'est pas suffisant pour faire exploser les temps de calcul de toutes les approches.

La dernière de nos expérimentations (figure 5.5) utilise un jeu de données particulièrement difficile à fouiller, de loin plus dense que tous les autres. En effet, chaque transaction comporte ici 10 items différents à explorer et chaque sequence est composée de 25 de ces transactions. Il est relativement évident que l'espace de recherche des motifs est considérablement plus vaste que celui des autres jeux de données utilisés au-dessus. Ceci va nous permettre de distinguer les approches efficaces parce que leur implémentation est de qualité de celles efficaces parce que algorithmiquement judicieuses.

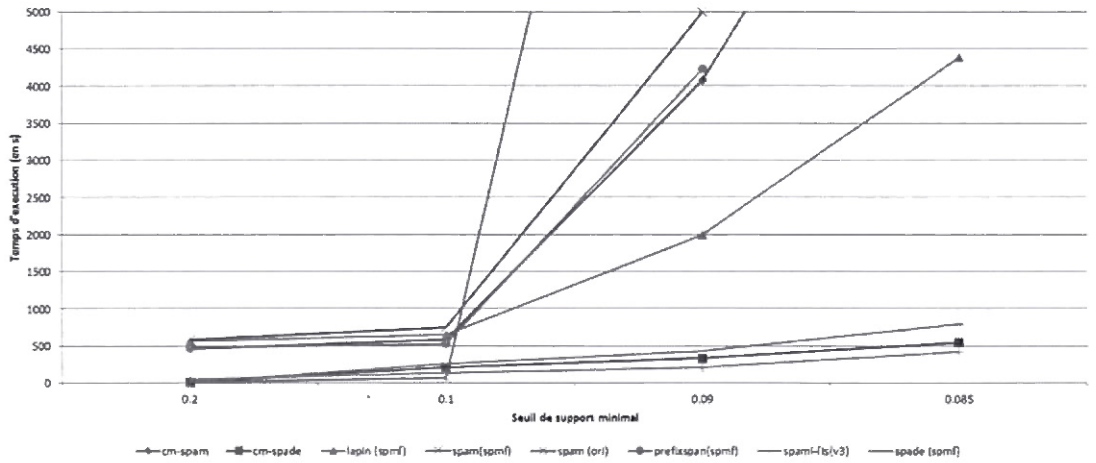

Figure 5.5: Comparaison des approches sur le second jeu de données synthétiques

Ce que l'on remarque au premier abord est qu'au seuil de support le plus élevé nous pouvons distinguer deux groupes : les approches extrêmement rapides toutes verticales (CM-SPADE, SPADE, SPAM original, etc.) et les approches beaucoup plus lentes comprenant LAPIN, PrefixSpan ainsi que SPAM (spmf) et CM-SPAM.

La plus rapide de toutes sur le premier seuil de support est SPAM original. Il s'agit d'une implémentation très efficace, la seule restante de langage C. La plus lente de toutes est SPAM de SPMF. Ceci montre très bien que la qualité de l'implémentation

constitue l'essentiel des performances. Cependant, si l'on compare les performances de ces deux implémentations, sur un seuil inférieur à 10% la tendance s'inverse. Ceci est certainement dû au besoin d'utiliser un swap puisque SPAM consomme beaucoup trop de mémoire.

Encore une fois, les version CM de SPADE et SPAM sont plus efficaces que leur versions d'origine.

Si l'on compare ces résultats avec l'expérimentation précédente, ici il est clair que PrefixSpan subit de plein fouet l'augmentation de taille de l'espace des motifs. De même, LAPIN malgré le fait qu'il apparient à la famille des approches verticales d'élagage n'est pas en mesure de compenser l'augmentation de l'espace des motifs.

Indubitablement, SPADE, CM-SPADE et notre version de Spami-FTS sont ici les approches les plus efficaces. A cause de la densité des donnes, la matrice de co-occurrence ne semble pas aider outre-mesure les approches CM-SP. Cependant, plus le support baisse, plus l'écart entre SPADE et CM-SPADE semble se creuser en faveur de CM-SPADE. Les techniques d'élagage précoce utilisées dans notre version semblent plus efficaces que la seule matrice de co-occurrence des approches CM-SP. Néanmoins, l'écart de performance entre Spami-FTS et CM-SPADE semble diminuer au fur et a mesure que le support diminue, ce qui indiquerait que sur des seuils encore plus bas, CM-SPADE l'emporterait. Ceci semble parfaitement logique puisque plus le seuil de support minimal est bas, plus de motifs sont fréquents et plus il y a de motifs à combiner et la technique d'intersection inefficace utilisée dans notre approche n'est plus en mesure d'être compensée par les techniques d'élagage de l'espace des motifs. Quoi qu'il en soit, notre approche est en mesure de battre l'ensemble de toutes les approches sur un jeu de données extrêmement difficile à fouiller avec lequel les autres approches à l'exception de CM-SPADE montrent clairement leurs limites.

Ainsi à travers nos quatre expérimentations nous avons pu mettre en avant les approches

les plus efficaces dépendament de la complexité de différents espaces de motifs. Nous avons pu montrer que les approches verticales ne montrent leur plein potentiel que dans les cas extrêmes, où les autres approches ne sont pas en mesure de compenser la quantité de motifs à tester. De plus, les approches de type élagage précoce semblent être les plus efficaces avec les approches verticales. Cependant, sur deux jeux de données beaucoup plus creux, les approches verticales ne sont pas nécessairement à privilégier. Toutefois, il convient de prendre cette observation avec un peu de recul, puisque nous avons aussi montré que la qualité des implémentations joue un rôle majeur dans les performances observées et que seuls les cas extrêmes permettent de distinguer clairement une hiérarchie algorithmique dans les implémentations.

Globalement, si l'on tient compte de la différence de performances relatives entre les implémentations en C et les implémentations Java, notre version de Spami-FTS++ est celle qui se comporte le mieux dans chacune des expérimentations proposées. Ceci semble montrer que nos intuitions concernant l'importance des méthodes d'élagage précoce de l'espace de recherche sont effectivement correctes.

5.2 Proposition d'une nouvelle solution

Notre nouvelle solution va être un algorithme de type vertical puisque il s'agit de l'approche la plus efficace comme l'ont montré nos expérimentations.

Notre objectif va être d'appliquer une approche de fouille de motifs séquentiels verticale à nos motifs séquentiels de données liées. Comme nous l'avons montré dans notre état de l'art, il existe deux branches dans cette famille d'algorithmes :

- ceux basés sur SPADE (LAPIN-SPAM, CM-SPADE, etc.) qui représentent les séquences positions numériques,
- ceux basés sur SPAM (HVSM, CM-SPAM, etc.) qui représentent les séquences sous forme de vecteurs de bits,

Les premiers sont plus adaptés aux travaux sur de longues séquences composées de petits itemsets ou d'imtesets singletons tandis que les seconds sont plus adaptés à travailler sur séquences plus courtes mais composées de grands itemsets. Or, selon nous, les spécificités de nos motifs sont très proches d'une fouille de motifs séquentiels incluant des itemsets de taille supérieure à un. En effet, si l'on considère les deux items dans un 2-itemset comme une co-occurrence d'items alors il est possible d'assimiler l'ajout de propriétés à une double co-occurrence, c'est-à-dire une 2-séquence de 2-itemsets dont un item est commun aux deux itemsets.

Par exemple, le 2-motif séquentiel de données liées suivant $\{\langle Ville, Hôtel\rangle, \{avoir\_un\_hôtel(0,1)\}\}$ peut être relaxé pour former la 2-séquence de 2-itemsets suivante $(\{Ville, avoir\_un\_hôtel\}, \{Hôtel, avoir\_un\_hôtel\})$. Ainsi, l'item $avoir\_un\_hôtel$ présent dans le premier itemset peut représenter une solution pour le domaine de la propriété tandis que le second peut représenter une solution pour le codomaine de la propriété. Toutefois, ceci n'est possible que si ces deux solutions font bien partie du même triplet au sein de données à fouiller. En d'autres termes, ceci montre que la recherche d'un triplet compatible avec un motif est relativement proche des problématiques existant au sein de la littérature de fouille de motifs séquentiels fréquents couverte au chapitre 2 avec une contrainte d'unicité supplémentaire.

Ce parallèle nous permet de mettre en avant la ressemblance entre nos opérateurs d'ajout de propriétés et les opérateurs déjà bien connus dans le domaine de la fouille de motifs séquentiels et à fortiori de fouille d'itemsets. Dès lors s'orienter vers une solution inspirée par une approche de fouille de motifs séquentiels composés d'itemsets non-singletons telle que SPAM semble tout indiqué.

Toutefois, notre fouille de motifs n'est pas unidirectionnelle comme peut l'être une fouille de motifs séquentiels classique. L'ajout de classe nécessite de parcourir l'espace restant dans une séquence tandis que l'ajout d'une propriété nécessite de potentiel-

lement reparcourir une partie de l'espace des séquences déjà écarté lors d'ajouts de classes précédents (plus à gauche). En outre, il faut aussi être en mesure de récupérer les résultats de tous les opérateurs d'ajout de classe (et spécialisation) appliqués au motif depuis le début (motif vide). Comme pour notre approche précédente, nous ne pouvons donc pas écarter une notion d'historique manipulable pour chaque motif.

L'inconvénient majeur de notre précédente approche est le nombre de ré-appariements nécessaires pour étendre un motif lors de l'ajout (ou spécialisation) de propriétés. Intuitivement, pour éviter d'avoir à rechercher des nouvelles solutions à des opérateurs déjà appliqués, il suffit de modifier ces opérateurs pour calculer non plus une seule solution mais plutôt toutes les solutions.

Néanmoins, calculer toutes les solutions plutôt qu'une seule a des grandes chances d'alourdir le processus ainsi que rajouter beaucoup de calculs pour ne pas améliorer sensiblement les performances. A titre de rappel, il s'agissait de l'approche employée par xPMiner, même si elle était effectuée à chaque fois au complet et en deux temps.

Une approche basée sur SPAM serait en mesure d'apporter une solution à toutes ces contraintes. En ce qui concerne l'ajout de propriétés, si elles se comportent bien comme un "double-itemset" les opérations proposées par SPAM semblent être l'approche idéale. De même, puisque SPAM fonctionne à l'aide de vecteurs de bits sur lesquels les opérateurs sont intégralement appliquées, SPAM semble une solution appropriée au dernier point.

Restent les opérateurs de spécialisation qui se comportent comme des filtres sur les résultats des opérateurs précédents. Qu'ils soient calculés avec une approche à l'aide de vecteurs de bits ou de positions numériques ne change pas grand chose étant donné qu'ils s'appliquent à chaque fois sur un espace de plus en plus restreint. Le choix de notre nouvelle approche ne tient pas compte de ces opérateurs sachant que l'essentiel du travail de spécialisation sera déjà pris en charge par la base verticale (inférence). Ce

point sera détaillé par la suite (sous-section 5.2.1 p. 93).

Notre nouvel algorithme suit un fonctionnement quasiment identique à SPAM. Il alterne les phases d'ajout de nouvelle classe et d'ajout de nouvelle propriété ainsi que leurs spécialisations respectives. Par rapport à la version d'origine de SPAM, cet algorithme comporte deux changement majeurs.

Le premier est que contrairement à SPAM ici il est nécessaire de conserver les appariements successifs d'un motif, autrement dit son "historique" pour être en mesure de rajouter des propriétés entre le dernier élément et l'une des positions antérieures dans le motif. Ainsi il s'agit d'un SPAM "avec mémoire" puisqu'il est capable de restituer les positions exactes des correspondances dans chacune des séquences utilisateur pour chaque motif. Bien sur, cette fonctionnalité fait que cet algorithme consomme encore plus de mémoire vive que la version d'origine de SPAM ce qui était déjà son point faible. Toutefois, il serait possible en cas de manque de mémoire de maintenir la totalité des informations nécessaires au fonctionnement de l'algorithme directement sur le disque (y compris les deux bases verticales).

Le deuxième point qui diffère de SPAM est que notre algorithme (tout comme le précédent) associe à chaque motif fréquent les identifiants de séquences dans lesquelles il apparait (sid).

Pour chacun des candidats formes à partir d'un motif établi comme fréquent, ne seront explorées que les séquences restantes, c'est-a-dire appareillés avec le motif parent. Il s'agit donc d'une réduction importante de l'espace de recherche des séquences pour un motif. A noter que cet élagage était déjà utilisé dans xPMiner.

Tout comme SPAM chacun des motifs fréquents à étendre est associé avec la liste des items localement fréquents. SPAM maintient deux listes, une pour les extensions de séquences et une pour les extensions d'itemsets. Nous maintenons ici quatre listes dis-

tinctes, une pour les extensions de classes, une pour les extensions de propriétés, une pour les spécialisations de classes et la dernière pour les spécialisations de propriétés.

Le dernier point à souligner concerne l'implémentation. SPAM, que ce soit dans (Ayres *et al.*, 2002) ou dans l'implémentation d'origine est un algorithme complètement récursif. Il ne possède donc pas de pile d'exécution explicite. Nous avons fait le choix d'utiliser une pile explicite à des fins de parallélisation. En effet, les opérateurs d'ajout et de spécialisation sont indépendants entre eux. Rien ne nous empêche de paralléliser le traitement global de la pile d'exécution. Nous ne traiterons pas ce sujet dans ce travail, nous nous en tiendrons à préciser que l'allocation de l'espace de travail global est conçue pour gérer différents threads en concurrence notamment pour empêcher les problèmes liés à une mauvaise gestion des différents niveaux du cache CPU tels que le "false sharing" (Jin *et al.*, 2005). Nous sommes tout à fait conscients qu'une telle approche comporte un impact, certes relativement minime, sur les performances globales de l'outil mais en plus de faciliter le contrôle du flot d'exécution permet aussi de tester différentes stratégies de parcours (profondeur d'abord ou largeur d'abord). Pour terminer, notre algorithme est implémenté en Java, tout comme les versions précédentes de xPMiner.

Nous n'allons pas détailler le fonctionnement interne de l'espace de travail puisque ceci relève uniquement de l'implémentation et ne constitue en rien une avancée algorithmique. Ce qu'il faut comprendre est que cet espace de travail est une quantité de mémoire vive pré-allouée et de taille fixe qui nous permet de successivement appliquer tous nos opérateurs sur un contexte de motif fréquent sans jamais nécessiter plus de mémoire. En plus de contrôler complètement et fortement limiter la quantité de mémoire utilisée par notre implémentation, cette technique à pour effet supplémentaire d'améliorer considérablement la gestion du cache du processeur et ainsi augmenter d'autant plus les performances de notre approche.

Même si précédemment nous avons montré l'efficacité de techniques de fouille verticales avancées (CM-SP, Spami-FTS, LAPIN) notre démarche n'est pas de les appliquer ici. En effet, nous souhaitons concevoir une première approche verticale qui résout efficacement le problème de la fouille de motifs séquentiels de données liées fréquents à travers l'application des opérateurs existants à un mode vertical. De plus, nous souhaitons mettre l'accent sur une méthode plus avancée d'élagage de l'espace de recherche que celle utilisée auparavant qui reste compatible avec les techniques proposées dans LAPIN et Spami-FTS mais celles-ci pourront être appliquées lors de travaux futurs. À titre de preuve, nous rappellerons que notre implémentation personnelle de Spami-FTS+ utilise une approche de type SPAM pour la recherche de tous les itemsets fréquents et le calcul des 2-séquences fréquentes.

Avant d'aborder le fonctionnement des nouvelles bases verticales et des nouveaux opérateurs, nous mettrons en avant l'algorithme de parcours global de notre nouvelle approche basée sur SPAM.

Nous utiliserons les opérateurs suivants : $\oplus$ pour représenter l'extension d'un motif avec une nouvelle classe et $\oplus_{pos}$ pour représenter l'extension d'un motif avec une propriété. Les ajouts s'effectuent toujours à la fin. Sauf pour les propriétés où pos correspond à la position du domaine dans le motif. L'opérateur $\otimes$ sera utilisé pour représenter l'opération de bitwise-anding entre deux vecteurs de bits. L'opérateur $\odot$ sera utilisé pour représenter l'opération de bitwise-oring entre deux vecteurs de bits. Pour une classe c ou une propriété p, $c_{\downarrow}$ et $p_{\downarrow}$ représentent respectivement toutes les classes filles directes de c et toutes les propriétés filles directes de p.

L'algorithme *OntoPattern-SPAM* (Algorithme 1 p.91) propose un fonctionnement à très haut niveau de notre nouvelle approche. Étant donné qu'il s'agit d'une approche de type vertical, les bases verticales sont construites au démarrage de l'algorithme (lignes 4 et 6). Les bases verticales seront détaillées dans la sous-section 5.2.1 suivante.

Algorithm 1: Ontopattern-SPAM : calcule tous les motifs séquentiels de données liées fréquents

Ontopattern-SPAM : sup_{min} : minimum support ;
seq : séquences utilisateurs ;
Ω : ontologie ;

$vb_C \leftarrow vbClasses(seq)$ ▷ Construction de la base verticale (classes)
$genVbClasse(vb_C)$ ▷ Généralisation dans vb_C(classes)
$vb_P \leftarrow vbProprietes(seq)$ ▷ Construction de la base verticale (propriétés)
$genVbProprietes(vb_P)$ ▷ Généralisation dans vb_P (propriétés)
$\mathscr{F}_{\Omega.P|\sigma} \leftarrow calculeToutesProprietesFrequentes(vb_P, min_{sup})$
$\mathscr{F}_{\Omega.C|\sigma} \leftarrow calculeToutesClassesFrequentes(vb_C, min_{sup})$
$Pile \leftarrow \emptyset$ ▷ Initialisation de la pile de tâches
forall $c \in \mathscr{F}_{\Omega.C|\sigma}$ **do**
    $allctx \leftarrow generateContexts(c, \mathscr{F}_{\Omega.C|\sigma}, \mathscr{F}_{\Omega.P|\sigma})$ ▷ Génère tous les contextes des classes fréquentes (1-motifs fréquents)
    c-step$(c, 2, allctx, Pile)$ ▷ Calcule toutes les extensions 2-fréquentes, ajoute dans la pile
    $c_{\downarrow} - step(c, 1, allctx, Pile)$ ▷ Calcule toutes les spécialisations 1-fréquentes, ajoute dans la pile
while $Pile \neq \emptyset$ **do**
    $tmp \leftarrow Pile.pop()$
    $p_{\downarrow} - step(tmp.m, tmp.ctx, Pile)$
    $c_{\downarrow} - step(tmp.m, tmp.ctx, Pile)$
    $p - step(tmp.m, tmp.ctx, Pile)$
    $c - step(tmp.m, tmp.ctx, Pile)$

En suivant, l'algorithme va évaluer quelles sont les propriétés fréquentes $\mathscr{F}_{\Omega.P|\sigma}$, c'est-à-dire les 0-motifs comportant une propriété (ligne 8). Cependant, des 0-motifs ne constituent pas des motifs valides puisqu'ils ne comportent pas de classes . Ainsi, nous les qualifierons de proto-motifs composes de une propriété.

Autrement dit, quelles sont les propriétés qui apparaissent dans plus de sup_{min} séquences. Il est inutile d'utiliser dans la suite une propriété à ce stade établie comme infréquente. Ce calcul est trivial grâce à la base verticale : pour une propriété p, il suffit simplement de boucler sur toutes les entrées dans la base verticale et de compter le nombre de séquences pour lesquelles la propriété apparait au moins une fois.

La même opération est effectuée pour les classes (ligne 9) pour calculer $\mathscr{F}_{\Omega.C|\sigma}$. Pour une classe c, il suffit simplement de boucler sur toutes les entrées dans la base verticale et de compter le nombre de séquences pour lesquelles la classe apparait au moins une fois.

Dans un second temps (ligne 12), pour chaque 1-motif fréquent, l'algorithme va construire un contexte ctx à partir duquel nous allons pouvoir exécuter nos quatre opérations.

Nos quatre opérateurs nécessitent en entrée, en plus d'un motif fréquent m, l'ensemble des classes encore localement fréquentes[4] ainsi que des propriétés et leurs spécialisations respectives et les listes de transformations (historique) contenant les vecteurs de bits propres aux sid appareillés avec ce motif. Cet ensemble auquel il faut ajouter la nature de la dernière opération sera par la suite désigné sous le nom de contexte ctx d'un motif.

Maintenant que nous avons introduit la notion de contexte ctx d'un motif, nous allons décrire l'opération $ctx.hist(sid, pos)$ qui sera utilisée dans chaque algorithme. Cette

4. Pour rappel, dans SPAM, les items dits localement fréquents sont l'ensemble des items dont l'ajout au motif préfixe commun a permis de trouver un nouveau motif fréquent.

opération nous permet de récupérer un état antérieur du vecteur d'appariement. Autrement dit, récupérer le vecteur d'appariement d'un des motifs "ancêtres" du motif courant. Cette opération prend deux éléments en entrée : un identifiant de séquence *sid* ainsi qu'une position dans l'historique. L'opération courante est à la position 0, ainsi la dernière opération se trouve à la position -1.

Avec ces contextes nous allons pouvoir commencer à appliquer nos quatre opérateurs. Toutefois, étant donne que seuls des 1-motifs sont construits à ce stade, seules les opérations c-step et $c_{\downarrow}$-step sont possibles.

À noter que les bases verticales contiennent aussi les informations sur les positions des classes et propriétés qui apparaissent indirectement dans les séquences, c'est-à-dire les éléments "ancêtres" des éléments dans les séquences. Ceci nous permet de supprimer directement de l'ontologie les classes/propriétés infréquentes incluant leurs spécialisations infréquentes. Celle-ci sera utilisée pour les opérations de spécialisation $c_{\downarrow}$-step et $p_{\downarrow}$-step.

À partir de la ligne 15, nous disposons de tous les 1-motifs (classes et spécialisations), de tous les 2-motifs sans spécialisations ainsi que des propriétés fréquentes. De là, l'algorithme va utiliser une pile de taches dans laquelle sont exécutées les quatre opérations à partir d'un contexte de motif. Chaque opérations étant en mesure d'ajouter des nouvelles opérations à exécuter.

5.2.1 Bases verticales et généralisation

La nouvelle version de l'algorithme fait partie de la famille des algorithmes de fouille de motifs séquentiels verticaux. En revanche, puisqu'il doit être en mesure de manipuler à la fois des classes et des propriétés il doit comporter deux bases verticales : une pour les classes et une autre pour les propriétés. Aussi, dans ces deux bases verticales sont précalculées respectivement toutes les positions des éléments non-feuilles dans les deux

hiérarchies (classes et propriétés).

Avant d'entrer dans le vif du sujet, nous tenons à préciser qu'il s'agit ici de réelles bases de données verticales. En effet, celles-ci sont conçues pour être en mesure de fonctionner telles quelles directement hors mémoire vive. Il ne s'agit donc pas de structures de données en mémoire conçues pour offrir le meilleur compromis entre performance et coût de stockage. Il s'agit d'une représentation directe des séquences utilisateurs sous forme de vecteurs de bits exprimant la présence ou l'absence d'un item pour une position donnée. En effet, même les séquences pour lesquelles un vecteur est vide (car un item n'apparait pas) sont représentées. Ce comportement est souhaité puisqu'il permet à la fois d'accéder rapidement à l'information désirée et permet l'utilisation de ces deux bases telles quelles directement depuis un périphérique de stockage. La raison principale derrière ce choix est qu'une représentation verticale des séquences utilisateur est en règle générale beaucoup plus coûteuse en termes de stockage qu'une représentation horizontale. Ainsi, nous souhaitons permettre à notre algorithme de fonctionner tel quel, mais depuis le disque, dans l'éventualité où la quantité de données à analyser s'avérerait trop élevée pour demeurer entièrement en mémoire vive. D'autant plus que cette stratégie permet de conserver et recharger aisément nos deux bases verticales entre deux utilisations de l'algorithme.

Les mécanismes lies à la base verticale des classes sont très proches de ce que propose SPAM. C'est-à-dire qu'il s'agit pour chaque classe et chaque séquence de construire un vecteur de bits qui représente sa présence ou absence dans la séquence en question. Nous ne reviendrons pas sur ce point puisqu'il est déjà expliqué dans l'état de l'art au chapitre 2. Dans le cas présent, ce vecteur permet d'accéder à l'aide d'un sid et d'une classe c aux positions auxquelles cette classe apparait (ou pas) dans ledit sid. Nous désignerons le vecteur de bits correspondant à une classe c pour un sid fixe par vdb_c^{sid}.

La base verticale des propriétés est construite afin de permettre d'accéder grâce à un

couple sid, pos et une propriété p aux positions auxquelles dans cette séquence, il existe un triplet (x, p, z) tel que z apparaisse à la position pos. Autrement dit, à partir d'une position pos quelles sont les positions x compatibles en tant que domaine pour la propriété p.

Pour une propriété p dans un sid fixe, nous utiliserons $vdb_p^{sid,pos}$ avec pos la position en tant que objet du triplet (codomaine). En effet, la vecteur de bits d'une propriété pour une séquence indique les positions ou il est domaine à partir d'une position jouant le rôle de codomaine qui est fixe. Ainsi, $vdb_p^{sid,pos}$ représente toutes les positions dans une sequence donnée où cette propriété apparait avec un codomaine en position pos.

La représentation utilisée dans les illustrations (figure 5.7) est une représentation simplifiée en ceci qu'elle inclut aussi le bit de la position codomaine (souligné). En réalité, cette position n'est pas présente dans la base verticale. Quoi qu'il en soit, il ne s'agit que d'un changement mineur puisque la seule altération algorithmique est que lors des intersections de vecteurs pour la recherche des positions valides en tant que domaine, il ne faut pas prendre en compte le dernier bit puisque celui-ci représente la position du codomaine qui est déjà connu et fixe. De même, lors de la recherche des positions codomaines compatibles, plutôt que de retourner à chaque fois dans la base verticale des propriétés nous réutilisons les vecteurs de bits des 1-motifs de propriétés fréquents. Cette technique nous permet de calculer les positions codomaines en $O(((|vec| - 1)/blocksize) + 1)$ plutôt qu'en $O(card(vec))$. Ici, vec représente le vecteur de bits de la taille de la sequence à tester. Sachant que la majorité des séquences sont encodées dans une primitive nous sommes en mesure de filtrer les positions en $O(1)$ plutôt qu'en $O(card(vec))$ ce qui constitue une petite amélioration.

Une des caractéristiques souhaitées pour la nouvelle base verticale était qu'elle soit en mesure de nous éviter de devoir faire appel à la représentation en mémoire de l'ontologie pour faire des tests de subsomption coûteux que ce soit lors de ajouts ou des

spécialisations de classes ou de propriétés.

Ainsi, la nouvelle base verticale rajoute directement dans les vecteurs de bits des classes/propriétés parentes les occurrences de leurs filles. Dès lors les phases de spécialisation ne nécessitent plus qu'une intersection de vecteurs de bits pour confirmer ou infirmer l'appariement actuel et les phases d'ajout peuvent directement utiliser le vecteur de bits de la classe à ajouter sans avoir à tester chacune des classes ou propriétés filles.

Simplement, pour une classe cette généralisation se calcule avec une union entre elle-même et l'union des vecteurs de bits de toutes ses classes filles directes :

$$vdb_c = vdb_c \odot \left(\bigodot_{c' \in c_\downarrow} vdb_{c'} \right)$$

De même, pour une propriété, la généralisation s'effectue par couple sid, pos, autrement dit par propriété et par position de codomaine. Elle suit le principe proposé au dessus :

$$vdb_p^{sid,pos} = vdb_p^{sid,pos} \odot \left(\bigodot_{p' \in p_\downarrow} vdb_{p'}^{sid,pos} \right)$$

Sur la figure 5.7 nous pouvons voir une séquence utilisateur contenant des classes et des triplets. Les quatre classes en présence sont ici *Ville*, *Montréal*, *Pau*, *Hôtel* et *Musée* tandis que les trois propriétés sont *avoir_un_hôtel*, *avoir_un_musée* et *avoir_un_hôtel*. Les exemples illustrant la suite de cette partie feront tous référence à cette base verticale.

Nous allons maintenant présenter les quatre opérateurs $AddConcept$, $SpeConcept$, $AddRelation$ et $SpeRelation$ qui pour rappeler leur ressemblance avec SPAM ont été renommés respectivement c-step pour $AddConcept$, $c_\downarrow$-step pour $SpeConcept$, p-step pour $AddRelation$ et $p_\downarrow$-step pour $SpeRelation$.

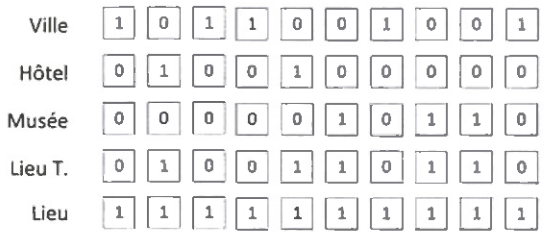

Figure 5.6: Nouvelle base verticale des classes

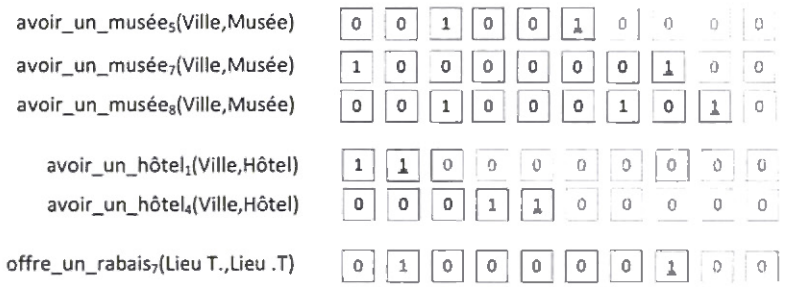

Figure 5.7: Nouvelle base verticale des propriétés

5.2.2 Ajout de de classe, *c*-step

Cet opérateur est chargé de calculer les appariements entre l'ensemble des motifs construits en appliquant l'opérateur *AddConcept* sur un motif fréquent m et les séquences dont l'appariement avec m à été possible. Contrairement à l'approche d'appariement précédente, nous pouvons voir qu'ici qu'à l'instar de SPAM, toutes les extensions sont évaluées au sein d'un même opérateur. Ce choix est justifié par le fait que l'ensemble maximal de séquences pouvant appareiller avec succès un motif construit par extension est l'ensemble des séquences supportant le motif parent.

De tous nos nouveaux opérateurs celui-ci est le plus proche des opérateurs originaux proposés dans SPAM. Il s'agit d'une *s*-step mais comportant quelques améliorations.

L'algorithme 2 (p.99) présente la *c*-step. Pour chaque séquence (*sid*) présente dans le contexte, cet opérateur tente d'étendre le motif fréquent avec chacune des classes présentes dans le contexte. Préalablement à l'application de l'extension avec chaque

classe, il faut appliquer la transformation $\_^{\triangleright}$ telle que proposée dans SPAM (ligne 13). Qui est, pour rappel, une recherche du premier bit à 1 dans le vecteur en partant de la gauche, la mise à 0 de celui-ci puis la mise à 1 de tous les bits plus à droite restants. Nous utiliserons la notations proposée dans PRISM puisque SPAM ne propose pas de notation spécifique à cette opération et que cette notion nous semble relativement intuitive par rapport à sa fonction.

La seconde boucle *For* (ligne 20) est un nettoyage des extensions localement fréquentes. En effet, elle permet de supprimer des contextes des nouveaux motif fréquents fils toutes les classes qui n'ont pas permis d'étendre le motif courant et qui sont par l'heuristique d'Apriori désormais inutilisables dans quelconque extension de notre motif.

Dès lors (ligne 24), il ne reste qu'à construire un nouveau contexte de motif pour chaque nouveau motif fréquent puis l'ajouter au début de la pile pour qu'il puisse être étendu directement en suivant puisque nous fonctionnons en profondeur d'abord. La figure 5.8 ci-dessous illustre le fonctionnement de cet opérateur en ajoutant la classe B à la classe A selon l'exemple de base verticale présenté au dessus 5.7.

Avant de passer à la partie suivante, nous souhaitons préciser qu'en réalité l'opération de transformation $\_^{\triangleright}$ préalable à $\otimes$ ne concerne en réalité que le bloc "pivot" du vecteur de bits. C'est-à-dire le bloc qui contient le premier bit à 1. En effet, puisque le premier vecteur est transformé à partir de cette position en une séquence de 1 et qu'il est ensuite intersecté à avec un autre vecteur de bits, la solution constituée par les bits à 1 du second vecteur après la position "pivot". Dans notre implémentation, nous utilisons l'opération de transformation $\_^{\triangleright}$ telle que présentée dans SPAM puis dans PRISM.

Nous avons fait le choix de boucler sur les séquences d'abord plutôt que sur les items, parce que de cette façon il n'y a pas besoin de conserver plusieurs transformations de séquences en même temps. En effet, si on boucle sur les items puis sur les séquences

Algorithm 2: c-step : étend un motif fréquent avec toutes les classes possibles

```
1  c-step : ctx : contexte d'un motif fréquent de taille k − 1 ;
2  m : motif ;
3  vdb : base verticale des classes ;
4  k : taille du motif à évaluer ;
5  stack : pile d'exécution globale ;
6  frequents : ensemble des motifs fréquents ;
7  sup_min : support minimum ;
8
9  sids_par_classe ← {}                          ▷ Conserve les sid pour chaque classe
10 bitvect_par_classe ← {}   ▷ Conserve les vecteurs d'appariement pour chaque classe
11                                                ▷ Pour chaque séquence du contexte
12 forall sid ∈ ctx.sids do
13     ctx.hist(sid, −1) ← ctx.hist(sid, −1)^▷    ▷ Transformation provenant de SPAM
14                          ▷ Pour chaque classe non marquée comme infréquente localement
15     forall c ∈ ctx.ζ do
16         tmp ← vdb_c^sid ⊗ ctx.hist(sid, −1)    ▷ Intersection avec la base verticale
17         if tmp ≠ ∅ then
18             sids_par_classe[c].add(sid)
19             bitvect_par_classe[c][sid] ← tmp
20 forall c ∈ ctx.ζ do
21     if |sids_par_classe[c]| < sup_min then
22         ctx.ζ ← ctx.ζ − {c}                    ▷ Si infréquent, élimine la classe localement
23                                                ▷ Pour chaque classe restante
24 forall c ∈ ctx.ζ do
25     m' ← m
26     m'.ζ ← m.ζ ⊕ c                             ▷ Construit le motif fréquent
27     sids ← sids_par_classe[c]
28     vect ← bitvect_par_classe[c]
29     ctx' ← createContext(m', k + 1, sids, ctx.ζ, ctx.θ, Ω.getSpeClass(c), ∅)
30     forall sid ∈ sids do
31         ctx'.hist(sid, 0) ← vect[sid]          ▷ Copie l'historique
32     ctx'.lastOp ← ADDCLASS
33     stack.addfirst(ctx')
34     frequents.push(m', k, |sids|)
```

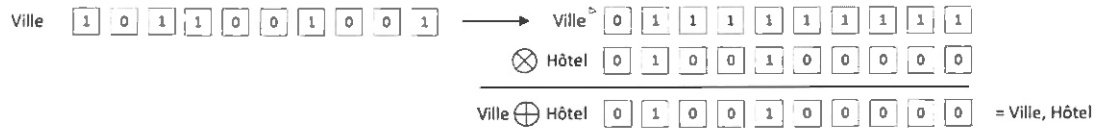

Figure 5.8: Vecteur pour l'ajout de classe *Hôtel* au motif composé par *Ville*

alors il faut à chaque fois changer le vecteur de bits servant de base à l'appariement.

La raison principale est que de cette façon nous pouvons identifier après le test de chaque sequence (avec tous les items) quels sont les nouvelles extensions fréquentes et ainsi mettre en commun les allocations nécessaires à la création de nouveaux contextes pour chacun de ces nouveaux motifs fréquents. L'inconvénient majeur de ce fonctionnement est qu'il demande plus de mémoire vive que le fonctionnement par item d'abord. Cependant, ceci est solutionné lors de la création de l'espace de travail après avoir calculé les classes et propriétés fréquentes. Si l'espace de travail a pu être crée correctement, alors le calcul d'une s-step ne nécessitera jamais plus de mémoire que celle qui a été allouée à l'espace de travail (Bien sûr, excepté la mémoire nécessaire aux contextes des nouveaux motifs mais il en va de même pour le fonctionnement par item d'abord).

5.2.3 Spécialisation de classe, $c\downarrow$-step

La spécialisation de classe est l'opération la plus simple de toutes. Cet opérateur est une version simplifiée de l'opérateur d'ajout de classe. Il s'applique uniquement si la dernière opération sur le motif était un ajout de classe ou une spécialisation de classe. Il s'agit d'un filtre sur le vecteur de bits de la dernière opération dans la liste.

L'algorithme 3 (p.102) relatif à la $c_{\downarrow}$-step commence par boucler sur chaque sequence présente dans le contexte puis chacune des classes filles de la dernière classe testée dans le motif.

Si le contenu du vecteur après intersection entre le vecteur de bits de la classe et le dernière élément de l'historique dans le contexte du motif reste non-vide, alors la séquence correspond au motif (ligne 15). Il suffit d'effectuer cette opération avec toutes les classes filles de la dernière classe ajoutée au motif et encore fréquente.

Cet algorithme est extrêmement proche de celui de la c-step. La différence principale est que contrairement à la c-step, ici pas besoin de nettoyage car un motif fréquent fils ici ne va pas contenir ses classes sœurs de la classe ajoutée. En effet, si un nouveau motif est fréquent ses spécialisations possibles ne sont que les classes filles de la classe constituant l'extension fréquente puisque aucune nouvelle classe n'est ajoutée par la suite.

Les autres différences ont qu'une spécialisation de classe n'ajoute pas de nouvelle classe, elle remplace la dernière classe par une autre. Ainsi, la taille k des motifs évalués ne change pas.

La figure 5.9 illustre la spécialisation de la classe E en C dans le motif ABE selon la base verticale présentée plus haut. Nous ferons remarquer qu'il s'agit du même résultat que si on avait étendu AB directement avec la classe C comme proposé par la figure 5.10. Néanmoins, l'avantage de le faire de cette façon est qu'il n'y pas de $\_^{\triangleright}$ à calculer les candidats de chaque classes filles. En outre, cette approche diminue le nombre d'opérations nécessaires pour parcourir le même espace de recherche puisque si E est infréquent localement alors C ne sera jamais testé. Ce n'est en rien une nouvelle stratégie, celle-ci était déjà exploitée dans xPMiner.

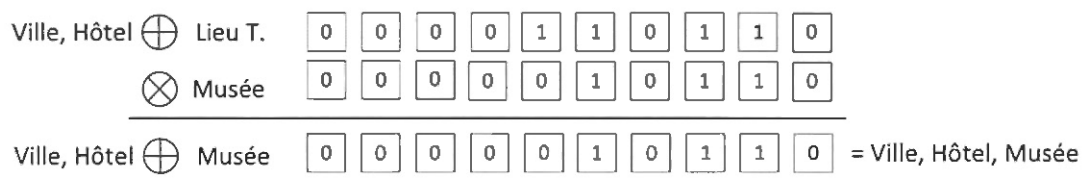

Figure 5.9: Vecteur pour la spécialisation de classe *Lieu.T* en *Musée*

Algorithm 3: $c\downarrow$-step : calcule toutes les spécialisations de classe d'un motif

$c\downarrow$-step : ctx : contexte d'un motif fréquent de taille $k-1$;
vdb : base verticale des classes ;
k : taille du motif à évaluer ;
$stack$: pile d'exécution globale ;
$frequents$: ensemble des motifs fréquents ;
sup_{min} : support minimum ;

$sids\_par\_classe \leftarrow \{\}$
$bitvect\_par\_classe \leftarrow \{\}$
▷ Pour chaque séquence du contexte
forall $sid \in ctx.sids$ **do**
  ▷ Pour chaque spécialisation de classe dans le contexte
  forall $c \in ctx.\zeta_{\downarrow}$ **do**
    $tmp \leftarrow vdb_c^{sid} \otimes ctx.hist(sid, -1)$ ▷ Intersection avec la base verticale
    if $tmp \neq \emptyset$ **then**
      $sids\_par\_classe.add(sid, item)$
      $bitvect\_par\_classe[c][sid] \leftarrow tmp$
forall $c \in ctx.\zeta_{\downarrow}$ **do**
  if $|sids\_par\_classe[c]| < sup_{min}$ **then**
    $ctx.\zeta_{\downarrow} \leftarrow ctx.\zeta_{\downarrow} - \{c\}$
forall $c \in ctx.\zeta_{\downarrow}$ **do**
  $m' \leftarrow m$
  $m'.\zeta \leftarrow m.\zeta \ominus m.\zeta[|m.\zeta| - 1] \oplus c$ ▷ Construit le motif fréquent
  $sids \leftarrow sids\_par\_item[c]$
  $ctx' \leftarrow createContext(m', k, sids, ctx.\zeta, ctx.\theta, \Omega.getSpeClass(c), \emptyset)$
  $ctx'.lastOp \leftarrow SPECLASS$
  $vect \leftarrow bitvect\_par\_classe[c]$
  forall $sid \in sids$ **do**
    $ctx'.hist(sid, -1) \leftarrow vect[sid]$ ▷ Copie l'historique
  $stack.addfirst(ctx')$
  $frequents.push(m', k, |sids|)$

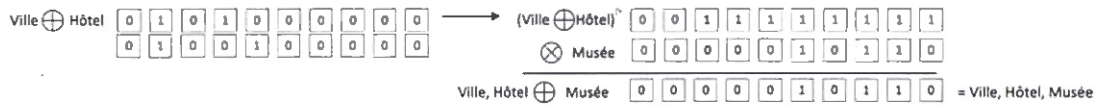

Figure 5.10: Vecteur pour l'ajout de classe direct *Musée* sans spécialisation

5.2.4 Ajout de propriété, p-step

Cet opérateur *p-step* est chargé de calculer les appariements entre l'ensemble des motifs construits en appliquant l'opérateur $AddRelation$ sur un motif fréquent m et les séquences dont l'appariement avec m à été possible. Conceptuellement, l'appariement entre un nouveau motif produit par $AddRelation$ et une sequence revient à :

1. vérifier qu'il existe au moins un triplet dont le codomaine correspond à au moins une des solutions identifiées pour la classe qui va jouer le rôle de codomaine ;
2. vérifier qu'il existe, parmi les triplets restants, au moins un triplet dont le domaine correspond à au moins une des solutions identifiées pour la classe qui va jouer le rôle de domaine ;
3. vérifier qu'il reste, parmi les solutions existantes à chacune des classes/propriétés présentes dans le motif entre la position domaine et la position courante (codomaine) au moins une solution pour chaque appariement ;

À première vue, certaines de ces trois vérifications peuvent prêter à confusion. Surtout la troisième qui peut sembler redondante voire contre-intuitive. Pour présenter la p-step nous allons donc procéder de la façon suivante : d'abord présenter les intuitions soutenant ces trois actions puis, par la suite, détailler ces mêmes actions sous forme de conditions plus formelles.

Tout d'abord, nous pouvons remarquer que chacune des trois vérifications prend en entrée le résultat de la condition précédente. En effet, chacune restreint l'ensemble des solutions restantes pour l'appariement. Chaque vérification étant alors assimilable à un filtre sur les vecteurs d'appariement du motif. Dès lors, nous pouvons voir que ces

actions peuvent intuitivement se représenter sous la forme d'intersections entre vecteurs d'appariements. Il en va de même pour la combinaison de ces trois actions.

Ainsi, rajouter une propriété à un motif agit de façon semblable à une contrainte sur les positions des appariements antérieurs. Ceci a pour effet de réduire (non-strict) l'ensemble des positions satisfaisant l'appariement des classes jouant le rôle de domaine et de codomaine.

Réduire l'ensemble des positions possibles pour une classe codomaine n'a aucune incidence sur les appariements antérieurs du motif, c'est-à-dire autres que ceux concernant la dernière classe appareillée. Il suffit qu'il en existe au moins une de valide pour poursuivre l'appariement. Néanmoins, cette réduction des positions sur l'appariement de la classe domaine peut invalider les appariements effectués sur les classes (et propriétés) présentes dans le motif entre les positions domaine et codomaine. C'est la raison pour laquelle la troisième condition est nécessaire pour qu'une sequence supporte un motif généré par $AddRelation$. Par la suite, nous désignerons cette réduction de positions par le terme "filtrage" puisqu'elle permet de supprimer des positions rendues invalides ou incompatibles avec la nouvelle contrainte sur le motif.

Tout ceci rend cet opérateur de loin le plus complexe. En effet, il nécessite de parcourir toutes les combinaisons de positions possibles entre la dernière position du motif (codomaine) et toutes les précédentes (domaines), fait intervenir directement l'ontologie et doit en plus effectuer des filtrages si l'extension est fréquente. De plus comme dans xPMiner, avant d'ajouter une propriété au motif il doit vérifier qu'elle n'est pas déjà présente aux mêmes positions.

Ainsi, pour faciliter la compréhension de cet opérateur, nous avons scindé son fonctionnement en deux parties : *p-step* (Algorithme 4 en p.106) et $apparie\_prop$ (Algorithme 5 en p.109). Le premier algorithme *p-step* effectue les opérations à plus haut-niveau des appariements ($sid \times p, \forall sid \in ctx.sids \wedge p \in \Theta$) alors que le second $apparie\_prop$ dé-

taille l'appariement entre un motif auquel une propreté p est ajoutée via $AddRelation$ et une séquence sid. Les trois vérifications mentionnées au-dessus seront explicitées avec le second algorithme $apparie\_prop$.

Tout d'abord l'opérateur *p-step* va récupérer la position cod dans le motif jouant le rôle de codomaine à partir de laquelle les nouvelles propriétés vont être testées (ligne 9). En suivant, la variable $cursor_{dom}$ est initialisée à une valeur de -2 (ligne 10). Cette variable représente un curseur nous permettant d'interagir avec l'historique des appariements présent dans le contexte ctx du motif. Pour rappel, nous avons indiqué précédemment que le dernier appariement se situe à la position -1 de l'historique. Ainsi, la variable $cursor_{dom}$ pointe vers le pénultième appariement.

Ces deux positions vont être les positions initiales pour tester l'ajout des propriétés au motif, cod représentant la position codomaine dans le motif et $cursor_{dom}$ représentant la position domaine dans l'historique. Il n'y a pas d'équivalent de $cursor_{dom}$ pour la position codomaine car celle-ci étant fixe, elle à toujours pour valeur -1.

La boucle principale de *p-step* itère sur les positions possibles en tant que domaine sur le motif délimitées par l'intervalle $]cod, 0]$. Pour chaque couple de classes formé par ζ_{pos} et ζ_{cod} nous recherchons les propriétés compatibles, c'est-à-dire les propriétés dont ζ_{pos} et ζ_{cod} sont respectivement des domaines et des codomaines valides selon l'ontologie Ω. Ici, Ω est assimilable à une structure de type table de hachage montée en mémoire interrogeable en $O(1)$. Il s'agit d'une structure de notre conception. Toutefois, celle-ci faisant déjà partie des anciennes versions du processus d'appariement, nous ne détaillerons pas son fonctionnement ici.

À la ligne 13, Θ représente l'ensemble de propriétés composé des propriétés racines compatibles avec les deux classes présentement testées et des propriétés encore présentes dans le contexte du motif. $ctx.\theta$ est l'ensemble des propriétés localement compatibles présentes dans l'en-tête du contexte de la même façon que pour les classes. Cet

Algorithm 4: p-step : étend un motif fréquent avec toutes les propriétés possibles

```
1  p-step : m : motif fréquent de taille k − 1 ;
2  ctx : contexte de m ;
3  vdb : base verticale des classes ;
4  k : taille du motif à évaluer ;
5  stack : pile d'exécution globale ;
6  frequents : ensemble des motifs fréquents ;
7  sup_min : support minimum ;
8
9  cod ← |ctx.motif.ζ|
10 cursor_dom ← −2                                  ▷ Curseur dans l'historique ctx.hist
11 forall pos ∈ ℕ ∩ ]cod, 0] do
12                                                  ▷ Sélection des propriétés à tester
13     Θ ← Ω.getRootProps(ζ_pos, ζ_cod) ∩ ctx.θ
14     if ∅ ≠ ctx.lastOp ∩ {ADDPROP, SPEPROP} then
15         Θ ← Θ − {(r, x, y) ∈ m.θ, x = pos ∧ y = cod}
16     sids_par_prop ← {}
17     if ∅ ≠ Θ then
18                                                  ▷ Calcul du support des motifs
19         forall sid ∈ ctx.sids × p ∈ Θ do
20             teste_propriete(sid, p, cursor_dom, ctx, vdb, k, stack, sids_par_prop)
21                                                  ▷ Évaluation du support des motifs
22         forall p ∈ Θ do
23             if |sids_par_prop[p]| < sup_min then
24                 Θ ← Θ − {p}
25                                                  ▷ Traitement des motifs fréquents
26         forall p ∈ Θ do
27             p' ← m ⊕_pos p
28             sids ← sids_par_prop[p]
29             ctx' ← createContext(p', k, sids, ctx.ζ, Θ, ∅, Ω.getSpeProps(p))
30             if ∅ ≠ calculeRaffinage(ctx.hist, sid, cursor_dom) then
31                 ctx'.lastOp ← ADDPROP
32                 stack.addfirst(ctx')
33                 frequents.push(p', k, |sids|)
34     cursor_dom ← cursor_dom − 1
```

ensemble Θ est alors l'ensemble des propriétés racines compatibles n'ayant jamais été évaluées comme infréquentes dans toute la généalogie du motif.

Si la dernière opération n'est pas pas directement liée aux propriétés, autrement dit s'il s'agit d'un ajout ou d'une spécialisation de classe, alors l'algorithme peut commencer à tester chacune des propriétés restantes. Dans le cas contraire, il est possible que certaines des propriétés aient déjà été ajoutées au motif entre les positions couramment étudiées. Dans ce cas, il est nécessaire de supprimer ces propriétés de la liste des propriétés à tester (ligne 15). En pratique, tout comme dans xPMiner, utiliser un ordre total entre les propriétés permet de s'assurer qu'un triplet ne soit pas dupliqué dans le motif.

sids_par_prop représente une structure de type associatif entre un *pid* (identifiant de propriété) et un ensemble de *sid* (identifiant de séquence). Cette structure va être remplie par la procédure *apparie_prop* (ligne 20) et va permettre à l'algorithme de distinguer entre un ajout de propriété résultant en un nouveau motif fréquent ou pas (ligne 24). Tout comme pour les classes et les spécialisations de celles-ci l'algorithme procède par *sid* puis par *pid*. La procédure *apparie_prop* va être détaillée par la suite, nous allons directement poursuivre par le traitement des propriétés fréquentes après appariement (ligne 29). L'opération $\Omega.getSpeProps$ permet de récupérer les spécialisations directes d'une propriété p.

Dans cette dernière étape, l'algorithme doit matérialiser les nouveaux motifs établis comme fréquents et construire leur contexte. Cependant, une dernière vérification est requise sous la forme d'un filtrage supplémentaire *calculeRaffinage* (ligne 30). Ce filtrage fait conceptuellement partie de *apparie_prop* que nous allons décrire en suivant. Il s'agit de la troisième vérification mentionnée plus haut, qui filtre les solutions de chaque appariement antérieur entre les positions domaine et codomaine dans le motif. Cependant, étant relativement couteux en temps de calcul par rapport au reste des opérations, il est plus judicieux de le calculer seulement si une extension de motif semble

fréquente. En effet, il est clair que beaucoup plus d'extensions infréquentes vont être testées que des extensions fréquentes.

Finalement, si la vérification s'avère fructueuse, le nouveau motif est ajouté à la pile de traitements pour être étendu à son tour. En outre, le curseur sur le domaine dans l'historique est mis à jour (ligne 34).

Ensuite, l'algorithme va répéter ce processus en décalant la position domaine de un cran vers le début du motif jusqu'à avoir testé toutes les combinaisons possibles. Si un motif est de taille k (k classes) alors l'opérateur va devoir tester $k-1$ combinaisons puisque le codomaine ne bouge pas.

Le reste de cette section va décrire la seconde procédure *apparie_prop* présentée dans l'algorithme 5 (p.109). Sachant que $cursor$ prend pour valeur $-(pos_{cod} - pos_{dom} - 1)$, la procédure *apparie_prop* va procéder de la façon suivante :

1. filtrer les positions restantes en tant que codomaine dans le dernier vecteur de bits du $ctx.hist$;
2. filtrer les positions de domaine possibles dans $ctx.hist(sid, cursor_{dom})$ avec les résultats des autres appariements situés entre le domaine et le codomaine ;
3. pour chacune de ses positions, intersecter dans $ctx.hist(sid, cursor_{dom})$, les positions domaines et les positions possibles indiquées dans la base verticale ;
4. filtrer les résultats des autres appariements dans le motif avec les nouveaux résultats pour la position domaine ;

D'emblée, il est aisé de remarquer que cet opérateur repose de façon intensive sur des opérations de filtrage des plusieurs vecteurs d'appariement du motif. Nous reviendrons sur ces filtrages après avoir expliqué le fonctionnement de la recherche de propriétés.

Pour valider l'appariement entre une séquence et un motif, cet opérateur va séquentiellement appliquer quatre conditions que nous désignerons respectivement par α, β, χ et

Algorithm 5: apparie_prop : calcule l'extension d'un motif avec une propriété

```
1  apparie_prop : sid : identifiant de la sequence à tester ;
2  p : propriété ;
3  cursor : curseur dans l'historique ;
4  ctx : contexte d'un motif fréquent de taille k − 1 ;
5  vdb : base verticale des classes ;
6  k : taille du motif à évaluer ;
7  stack : pile d'exécution globale ;
8  sids_par_prop : ensemble des séquences à tester pour une propriété ;
9
10 α ← F_{Ω.P|σ}[(p, sid)] ⊗ ctx.hist(sid, −1)
11 positions ← itemize(α)            ▷ Transforme les bits restants en liste de positions
12 vecteur_domaine ← ∅
13                                    ▷ Agrégat des positions restantes pour la classe domaine
14 β ← ctx.hist(sid, cursor) ⊗ ctx.hist(sid, cursor + 1)^◁
15 success ← |positions|
16                                    ▷ Pour chaque position de codomaine restante...
17 forall pos ∈ positions do
18 |                                  ▷ ... cherche si des positions domaines sont valides
19 |  χ ← β ⊗ vdb_p^{sid,pos}         ▷ Intersecte les positions restantes avec les domaines
20 |  if ∅ ≠ χ then
21 |  |  vecteur_dom ← vecteur_dom ⊙ χ   ▷ Agrège les résultats sans écraser les
   |  |   précédents
22 |  else
23 |  |  success ← success − 1
24 if 0 ≠ success then
25 |  ctx.hist(sid, cursor) ← vecteur_dom
26 |  sids_par_prop.add(sid, prop)
```

ξ. Ces pour valider un appariement lors de l'ajout d'une propriété $p(pos_{dom}, pos_{cod})$ à un motif sont les suivantes :

1. $\alpha \leftarrow \mathscr{F}_{\Omega.P|\sigma}[(p, sid)] \otimes ctx.hist(sid, -1) \neq \varnothing$: Intersecter les positions restantes pour la dernière classe du motif avec celles ou un triplet p possède un codomaine en pos_{cod} ;
2. $\beta \leftarrow ctx.hist(sid, cursor) \otimes ctx.hist(sid, cursor + 1)^{\triangleleft} \neq \varnothing$: Appliquer la restriction due à l'élément suivant du domaine ;

3. $\chi \leftarrow ctx.hist(sid, cursor) \otimes vdb_p^{sid,dom} \neq \varnothing$: Intersecter les positions restantes de domaine avec celles ou une propriété démarre et qui finit en pos_{cod} ;
4. $\xi \leftarrow \bigotimes_{i \in]pos_{dom}, pos_{cod}]} ctx.hist(sid, i) \otimes ctx.hist(sid, i-1)^{\triangleright} \neq \emptyset$: Re-filtrer tout les blocs en appliquant le restriction du nouveau domaine ;
5. $\alpha \otimes \beta \otimes \chi \otimes \xi \neq \varnothing$;

L'opérateur $\_^{\triangleleft}$ est défini par dualité avec $\_^{\triangleright}$, il est la permutation du bit le plus à droite ainsi que la mise à 1 de tous les bits plus à gauche. Si $\_^{\triangleright}$ se traduit par la création d'un masque de conservation des bits situés après une position pivot alors $\_^{\triangleleft}$ représente la création d'un masque de conservation des bits situés avant la position pivot.

La première condition α (ligne 10) nous impose de vérifier qu'il existe au moins un triplet dont le codomaine correspond à au moins une des solutions identifiées pour la classe qui va jouer le rôle de codomaine. Tout d'abord, les solutions identifiées pour la classe qui va jouer le rôle de codomaine correspondent simplement au vecteur d'appariement présent en dernière position dans l'historique, autrement dit $ctx.hist(sid, -1)$. Pour être en mesure de satisfaire cette condition, il est nécessaire de combiner les positions valides de $ctx.hist(sid, -1)$ avec les positions des triplets existants dans les données. Cette informations nous est accessible dans la base verticale. Néanmoins, notre base verticale est construite par paire (p, iid) autrement dit, par propriété et position de codomaine.

Ceci implique qu'utiliser la base verticale revient à intersecter autant de vecteur de bits qu'il y a de positions de codomaine possibles dans la sequence. Selon nous, ce choix n'est pas le plus judicieux. En effet, à ce stade, l'algorithme a déjà calculé toutes les positions valides en tant que codomaine dans la sequence au sein du contexte du *1-motif* fréquent contenant seulement la propriété à tester. Cette information est accessible à travers la structure $\mathscr{F}_{\Omega.P|\sigma}$ (proto-motifs de propriétés).

Nous avons opté pour la seconde option car elle permet de filtrer les positions codo-

maines en une seule opération même si elle implique de conserver les contextes des 1-propriétés fréquentes. En termes de complexité, la première option est d'ordre linéaire puisqu'elle nécessite de se référer à la base verticale pour chaque position restante dans le vecteur. Même en négligeant les couts d'interrogation de la base verticale la seconde option est préférable. Quoi qu'il en soit, les deux options sont viables et opérationnelles (implémentation). Les figures 5.11, 5.18 et 5.13 à la page 111 illustrent le fonctionnement de la première option. Tandis que la figure 5.14 à la page 112 illustre le fonctionnement de la seconde option.

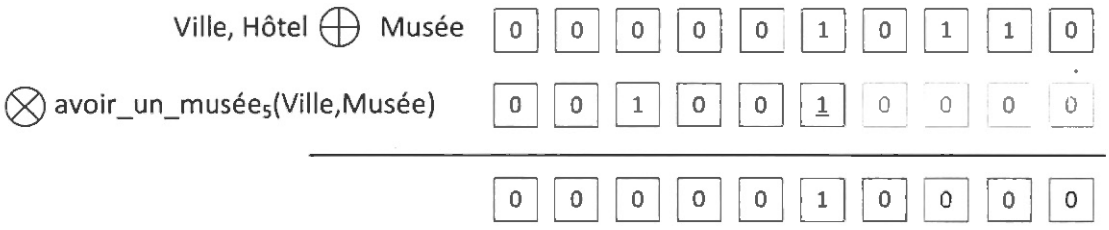

Figure 5.11: Vecteur α pour l'ajout de propriété *avoir_un_musee*$_5$

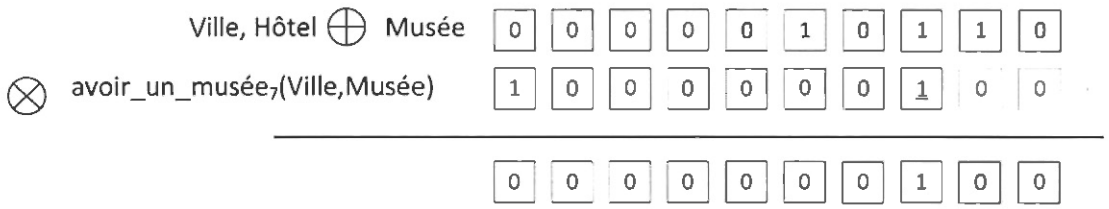

Figure 5.12: Vecteur α pour l'ajout de propriété *avoir_un_musee*$_7$

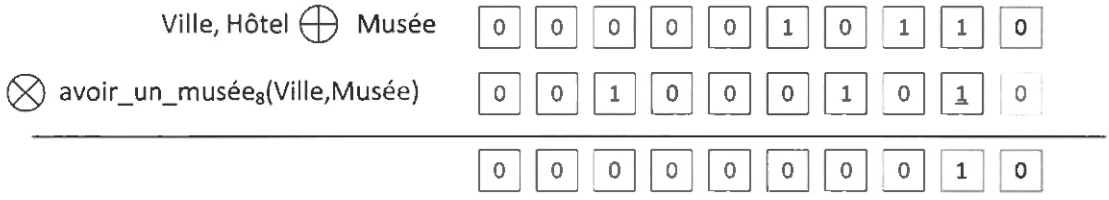

Figure 5.13: Vecteur α pour l'ajout de propriété *avoir_un_musee*$_8$

La figure en question représente les intersections entre les vecteurs de bits après l'application du dernier opérateur et le vecteur de bits dans la base verticale pour la propriété

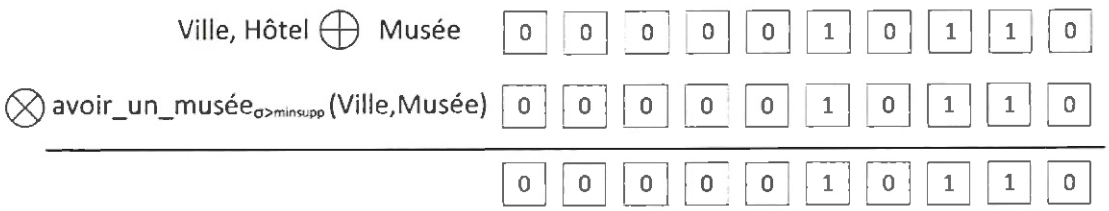

Figure 5.14: Vecteur α pour l'ajout de propriété "one-shot" *avoir_un_musee*

à ajouter sur le *sid* courant à partir de la position soulignée dans le vecteur. De là, imaginer le fonctionnement de la seconde option est trivial, il faut se représenter un seul vecteur composé des positions soulignées dans les vecteurs de propriétés et effectuer une seule fois le même calcul que dans les exemples.

La seconde condition β (ligne 14) est un héritage direct de SPAM. En effet, chaque vecteur d'appariement contient toutes les solutions pour le problème d'appariement à résoudre lors de sa construction. Avant de pouvoir conclure qu'un triplet utilisant p est bien présent sur la sequence *sid* aux positions étudiées, il est nécessaire d'intersecter le vecteur résultat avec une transformation du vecteur d'appariement suivant dans l'historique. Ainsi, des solutions pour un problème d'appariement précédent peuvent se trouver "à droite" d'une solution pour le problème d'appariement (dans le vecteur) suivant. En effet, le vecteur d'appariement suivant détermine une position limite pour l'appariement actuel en ceci que l'appariement actuel peut trouver des solutions jusqu'en fin de séquence. Toutefois, l'appariement suivant via sa solution la plus à droite nous donne une limite "vers la droite" à ne pas dépasser lors de notre appariement actuel. Or, ceci ne constitue pas un appariement valide. Alors, filtrer les solutions présentes dans le vecteur d'appariement pour la classe domaine en supprimant toutes les positions présentes plus à droite que et sur la dernière solution pour la classe suivante dans le motif devient nécessaire. En théorie, il faudrait effectuer ce filtrage sur toutes les positions du motif entre le position domaine et la position codomaine. Mais même si cette opération pourrait aussi être faite à chaque ajout ou spécialisation de classe,

ce nettoyage n'est nécessaire que pour la recherche de propriétés. C'est la raison pour laquelle nous avons opté pour son calcul uniquement dans la p-step. Le coût de calcul de ce filtrage est négligeable d'autant plus qu'il n'est pas toujours nécessaire. En effet, il n'est utile que s'il existe une position plus à droite dans le vecteur à traiter que la position la plus à droite dans le vecteur suivant. Aussi, nous avons fait le choix de ne filtrer que le vecteur d'appariement en position domaine avec son successeur car un filtrage global est aussi nécessaire en fin d'appariement. Il s'agit de la condition ξ. Il n'est donc pas nécessaire d'effectuer cette vérification sur tous les éléments appareillés du motif entre la position du domaine et celle du codomaine. En effet, deux cas de figure existent :

— soit il s'agit de la première propriété ajoutée dans le motif, auquel cas un ajout de propriété précédent n'a pas pu "fausser" les résultats des appariements ;

— soit ne s'agit pas de la première propriété ajoutée dans le motif, auquel cas le filtrage final ξ évoqué plus haut appliqué en sortie de l'appariement filtre correctement les positions restantes ;

Ainsi, grâce au filtrage ξ appliqué en sortie, chaque les positions des appariements sont "à jour" et respectent toutes les contraintes lors de l'entrée dans le processus d'appariement. Satisfaire cette seconde condition nous permet de conclure qu'il existe un triplet satisfaisant la nouvelle condition mais ne nous permet pas de valider que la solution à cette nouvelle contrainte ne viole pas les autres contraintes déjà présentes dans le motif. Un exemple de cette condition β est visible à la figure 5.15.

La troisième condition χ (ligne 19) demande de croiser les positions restantes pour le domaine avec les positions de domaines au sein des séquences utilisateur (triplets). Pour ce faire, nous allons devoir utiliser la base verticale pour vérifier chacune des positions restantes dans le vecteur d'appariement avec les positions candidates pour le domaine. Deux exemples sont visibles aux figures 5.16 et 5.17.

La quatrième condition permet de vérifier que la restriction sur les solutions due à

l'ajout de la propriété ne viole pas les autres contraintes (classes et/ou propriétés) du motif. Elle nous oblige de filtrer les vecteurs d'appariement dans $\{ctx.hist(sid, i)\}, i \in]dom, cod[$. Cette quatrième condition ξ est représentée par la procédure *calculeRafinage* dans l'algorithme 4(p.106). Un exemple de ce filtrage est visible à la figure 5.18 en page 115.

Figure 5.15: Vecteur β pour l'ajout de propriété

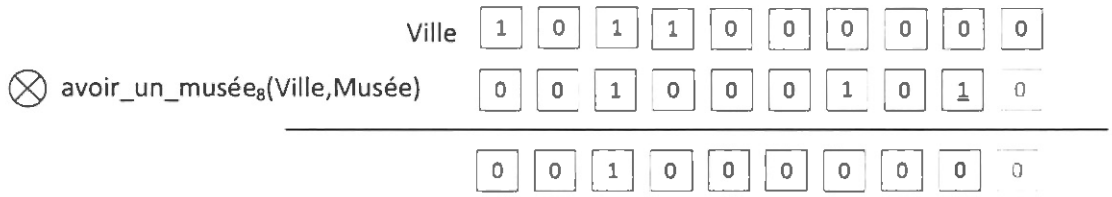

Figure 5.16: Vecteur χ pour l'ajout de propriété *avoir_un_musee*$_8$

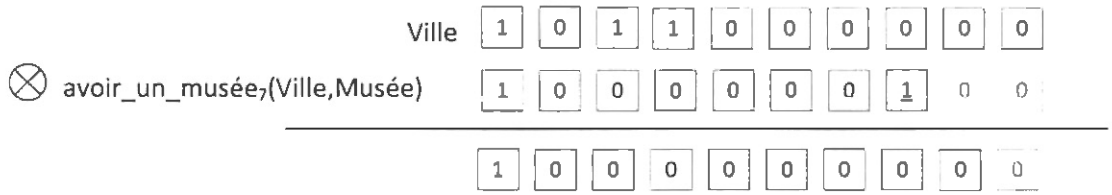

Figure 5.17: Vecteur χ pour l'ajout de propriété *avoir_un_musee*$_7$

Ce second filtrage est plus couteux mais n'est appliqué sur un vecteur d'appariement que si :

— le motif est apparemment fréquent (c'est-à-dire que tous les tests excepté le filtrage restant indiquent qu'un nombre d'appariements suffisants sont possibles) ce qui permet de le réduire sensiblement son coût total ;

— si le bit le plus à gauche dans le vecteur d'appariement pour le domaine a changé. En effet, le filtrage élimine des positions plus à gauche que le premier bit à 1 du vecteur. Si celui-ci ne change pas, les autres positions sur les autres

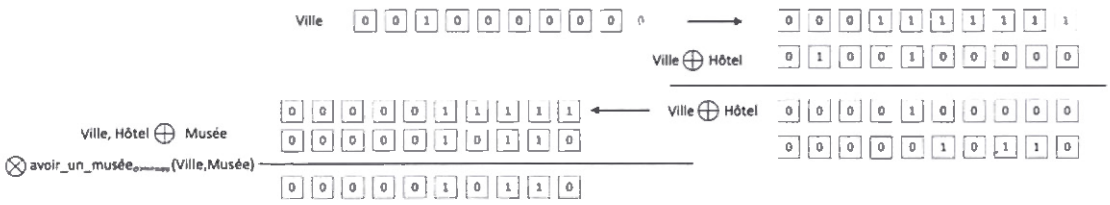

Figure 5.18: Vecteur ξ pour l'ajout de propriété *avoir_un_musee*

vecteurs restent valides. En outre, récupérer ce vecteur le plus à gauche est "gratuit" car pour vérifier si l'appariement est valide, il est déjà nécessaire de vérifier que les primitives (int32) composant le vecteur soient non nulles, autrement dit que la position du bit à 1 le plus à gauche existe (non-nulle) ;

À noter aussi que dernier filtrage correspond aux problèmes de retour en arrière évoques dans le chapitre précédent. Il s'agit de s'assurer que la restriction sur les solutions en tant que domaine n'invalident pas les solutions pour les classes suivantes dans le motif. En effet, dans le cas où (et uniquement si) la solution la plus à gauche dans le vecteur de bits change, il se peut qu'elle impacte sur les solutions pour les vecteurs suivants. Dans ce cas, si des propriétés existent déjà dans le motif et dont la position domaine est plus à droite que celle actuelle, il se peut que la nouvelle restriction invalide certains appariements pour cette propriété, voire tous. Il faut donc s'assurer de la validité des anciennes positions en appliquant un masque ET sur les anciens vecteurs d'appariement en vérifiant qu'il sont différents du vecteur vide. Dès que l'un de ces vecteurs devient vide, alors le masquage s'arrête et le support du motif est décrémenté. Si après toutes les vérifications le nouveau support est inférieur au support minimal alors le motif n'est plus fréquent.

À tout moment, sur chacune des quatre premières conditions si une des étapes résulte en un vecteur vide, il possible de stopper l'appariement puisqu'un vecteur vide est une condition nécessaire et suffisante pour un appariement impossible. En termes d'implémentation, cela signifie qu'il faut que chaque résultat temporaire dans chacun des blocs (ex : int32) du vecteur d'appariement soit différent de zero. De ceci résulte la cinquième

et dernière condition.

Nous avons donc une séquence de quelques opérations très rapides à calculer lorsqu'un appariement est possible mais dans la grande majorité des cas, l'appariement est en mesure d'être détecté comme impossible de façon encore plus rapide.

À la fin, s'il reste des positions dans le vecteur associé à la position domaine dans le motif alors toutes ces positions constituent des solutions au problème d'appariement (ligne 24). Pour chaque position en tant que codomaine, les vecteurs représentant le résultat de l'intersection sur les domaines sont agrégés avec une union des vecteurs de bits.

Comme nous l'avons proposé précédemment, nous disposons d'un opérateur capable de trouver toutes les solutions compatibles avec une nouvelle propriété ajoutée au motif en seulement quelques opérations simples. Dans le pire des cas, si une séquence est encodée dans un seul entier (max 32 positions) alors le nombre d'opérations nécessaires devient $O(1+1+m(O(1)))$ avec m étant le nombre de positions codomaines restantes après le premier filtrage. Si l'on tient compte du filtrage nécessaire après appariement (donc si le motif semble fréquent), la complexité devient $O(1+1+m(O(1))+(r-1)(O(1))$ sachant que la base verticale est requetable en $O(1)$ avec r l'écart entre les positions de domaine et de codomaine.

Si on considère les séquences de toutes les tailles dont le nombre d'entiers nécessaires à leur encodage est t alors la complexité devient $O(2t+m(O(1))+(r-1)(O(t))$. En résumé nous disposons d'un opérateur dont le complexité dépend uniquement du nombre de positions codomaines possibles et non pas du produit $pos_{dom} \times pos_{cod}$ comme précédemment. Ou encore, d'un opérateur dont pour une position codomaine fixe, la recherche de toutes les positions domaines compatibles est calculable en $O(m)$ ce qui constitue un gain non négligeable. Sachant qu'avec les données dont nous disposons, dans la grande majorité des cas, les séquences sont encodées dans une seule primitive

et que toutes les séquences sont encodées dans maximum deux primitives de programmation (int32).

Le dernier point à préciser est que ces deux filtrages sont associés à des *sid* spécifiques. Il serait donc envisageable d'appliquer un mécanisme de filtrage paresseux qui calcule d'abord les p-step pour les *sid* non impactés par un décalage (entrée et sortie) puis si le motif est déjà fréquent alors ne pas appliquer les transformations sur les *sid* restants et reporter ces transformations dans le contexte des motifs fils. Si le support temporaire n'est pas suffisamment élevé, alors calculer $support_{max} = support_{temp} + nbr\_sid\_avec\_transformation$ représentant le support maximal pouvant être atteint en appliquant les transformations et celles-ci débouchant sur une incrémentation du support. Si ce résultat est encore inférieur au support minimal alors quoi qu'il arrive le motif ne pourra pas être fréquent et il est donc inutile de calculer ces transformations. Il s'agit donc d'une autre technique d'élagage potentiellement intéressante en vue de limiter les calculs inutiles.

5.2.5 Spécialisation de propriété, $p_{\downarrow}$-step

Le dernier opérateur est une simplification du précédent. En effet, ici nul besoin de tester des combinaisons entre plusieurs positions au sein du motif à étendre. À l'instar de la spécialisation de classe, il suffit juste de filtrer les solutions du motif parent avec les vecteurs de bits des propriétés filles présents dans l'en-tête $ctx.\theta_{\downarrow}$ (ligne 8) grâce à l'opérateur précédent (p-step).

Néanmoins, tout comme avec l'ajout de propriété la notion de filtrage est essentielle. À ceci près que le filtrage en entrée n'est pas nécessaire puisque l'opérateur précédent est soit un ajout de propriété ou une spécialisation de propriété qui tous deux effectuent directement le raffinage lorsque nécessaire contrairement aux opérateurs sur les classes.

Pour chaque codomaine restant dans le vecteur de bits, la procédure recherche dans

la base verticale si la position existe pour cette propriété. Il est cependant, comme précédemment, plus rapide d'intersecter le vecteur de bits du motif correspondant au proto-motif composé par une seule propriété (1-propriété) fréquente spécialisée (calculée directement après construction de la base verticale).

De même que pour l'ajout de propriété, si la position existe, il faut intersecter les position de domaine avec le vecteur dans la liste des transformations grâce à la base verticale. À partir de là, il est nécessaire de filtrer les résultats raffiner. Si le motif est établi comme fréquent, il faut remplacer la dernière propriété ajoutée au motif par sa spécialisation présentement testée (ligne 18) puis remplacer les spécialisations possibles de propriétés avec les spécialisations de la propriété actuellement évaluée (ligne 21), puis placer le contexte du motif dans la pile d'exécution globale.

Nous n'allons pas illustrer cet opérateur avec un exemple puisque toutes les étapes de son fonctionnement ont déjà été détaillées dans la partie précédente.

5.3 Expérimentation de la nouvelle approche

Dans cette dernière partie nous allons proposer quelques expérimentations comparatives entre notre algorithme et le précédent afin de montrer nos gains sur les temps d'exécution. Pour ce faire nous disposons de trois jeux de données adaptés aux motifs que nous recherchons :

- Tourisme, un jeu de données synthétiques instanciant une ontologie du tourisme. Il s'agit du même jeu de données que celui utilisé dans le chapitre 4,
- Phylo_Manuel, un jeu de données réel instanciant une ontologie de workflows phylogéniques,
- Phylo_Auto, un second jeu de données réel plus massif instanciant une seconde ontologie de workflows phylogéniques,

Le tableau ci-après montre les caractéristiques de chacun de ces trois jeux de données

Algorithm 6: $p_{\downarrow}$-step : calcule toutes les spécialisations de propriétés d'un motif

```
1  p↓-step : ctx : contexte d'un motif fréquent de taille k − 1,
2  vdb : base verticale des classes ;
3  k : taille du motif à évaluer ;
4  stack : pile d'exécution globale ;
5  frequents : ensemble des motifs fréquents ;
6  sup_min : support minimum ;
7  sids_par_prop : ensemble des séquences à tester pour une propriété ;
8
9  Θ ← ctx.θ↓                          ▷ Récupère les spécialisations possibles
10 sids_par_prop ← {}
11 if Θ ≠ ∅ then
12     forall sid ∈ ctx do
13         forall prop ∈ Θ do
14             apparie_prop(ctx, vdb, k, stack, sid, sids_par_prop)
15     forall prop ∈ Θ do
16         if |sids_par_prop[prop]| < sup_min then
17             Θ ← Θ − {prop}
18     forall prop ∈ Θ do
19         m' ← context.motif ⊖ m.θ[|m.θ| − 1] ⊕ prop
20         sids ← sids_par_prop.get(prop)
21         ctx' ← createContext(m', k, sids, ctx.ζ, ctx.θ, ∅, Ω.getSpeProps(prop))
22         ctx'.lastOp ← SPEPROP
23         stack.addfirst(ctx')
24         frequents.push(m', k, |sids|)
```

ainsi que de l'ontologie qui leur est associée à chacun. Toutes les séquences utilisateurs sont ici des séquences d'événements (itemsets singletons). Les nombres indiqués entre parenthèses correspondent au nombre de classes/propriétés racines dans chaque jeu de données.

Jeu de données	Nbr. séq.	Moyenne trx/séq.	|O.I|	|O.C|	|O.P|	|O.T|
Tourisme	57	19,8	136	156(6)	35(17)	298
Phylo_Manuel	94	11,4	3698	115(7)	14(14)	6240
Phylo_Auto	485	17,2	592	94(7)	14(6)	473

Tourisme va être utilisé pour montrer l'impact de la variation de l'espace des motifs sur chacun des algorithmes. En effet, nous allons fixer un seul de support minimal et faire varier la taille maximale des motifs. Ainsi, nous montrerons la robustesse de chacune des approches face à l'explosion de l'espace des motifs.

Phylo_Manuel et Phylo_Auto vont être utilisés de la même façon que l'ont été les autres jeux de données dans nos expérimentations sur les approches séquentielles. Nous allons faire varier le seuil de support et observer les écarts de performance sur les temps d'exécutions.

La première expérimentation compare quatre approches : OP-2014S, OP-2016 et chacune des deux approches configurées pour ne tenir compte que des classes et leurs spécialisations. Le seuil de support minimal choisi ici est de 15% (15 séquences).

Cette expérimentation semble indiquer que les écarts de performance sont de plus en plus flagrants au fur et à mesure que le nombre de motifs à tester augmente. Aussi, elle indique de façon très claire que la difficulté de fouille de l'espace des motifs en incluant les propriétés devient de plus en plus grande dans chacune des deux approches lorsque la taille maximale des motifs augmente. Il est aisé de constater que le cout d'exploration de l'espace des motifs sans propriétés est beaucoup moins couteux qu'avec.

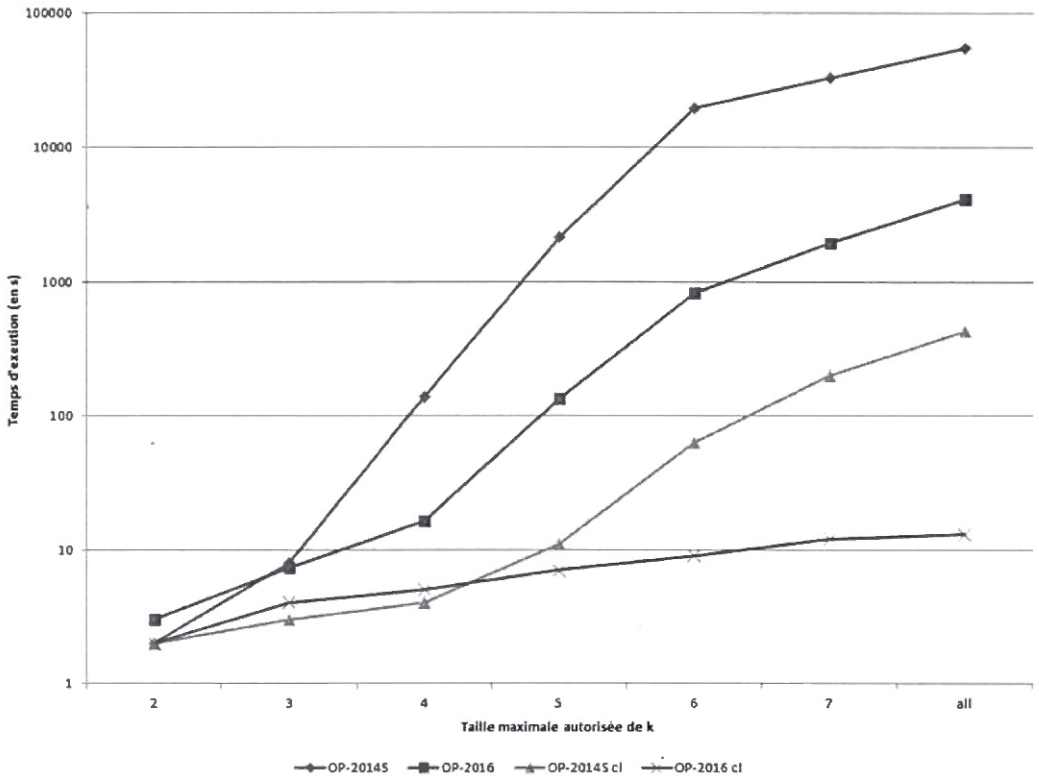

Figure 5.19: Comparaison des approches sur Tourisme avec $minsup$=15%

De plus, nous pouvons constater que la nouvelle approche grâce à ses techniques d'élagage de l'espace subit beaucoup moins l'explosion du nombre de combinaisons à traiter que l'ancienne méthode. Cependant, nous pouvons aussi constater que la nouvelle approche, puisqu'il s'agit d'une approche de type verticale demande quelques secondes pour la construction de la base verticale et est donc moins efficace lorsque k est petit. Nous pouvons observer aussi que le cout de construction de la base verticale est rentabilisé des k=5 pour uniquement les classes alors qu'il est rentabilisé des k=3 pour l'espace des motifs avec les propriétés.

Pour trouver tous les motifs fréquents sans propriétés de taille une à sept, il faut 427 secondes à OP-2014S alors qu'il faut seulement 13 secondes à OP-2016. Le ratio entre les deux est croissant pour atteindre 32 fois en cherchant tous les motifs.

Pour trouver tous les motifs de taille une à cinq, il faut 134 secondes à la nouvelle approche et 2160 à l'ancienne. Il s'agit donc d'un gain d'environ 16 fois. Pour trouver tous les motifs fréquents de taille une à sept, l'ancienne approche nécessite environ 55000 secondes, autrement dit un peu plus de 15 heures alors qu'il faut seulement un peu plus d'une heure à la nouvelle approche. Le gain est ici d'environ 13 fois. En moyenne sur l'ensemble des valeurs maximales de k, l'écart de performance est de 11 fois en faveur de la nouvelle approche. Néanmoins, son maximum est atteint pour un taille de k maximale de 6 avec 23 fois puis redescend à 16 fois pour k=7. Ainsi, intuitivement cet écart semble augmenter mais ce n'est pas le cas.

Toutefois, il s'agit d'un gain significatif puisque la nouvelle technique calcule toutes les solutions pour chaque motif alors que l'ancienne se contente de trouver une seule solution valide. Le fait que cet écart augmente puis diminue sur ce jeu de données et ce seuil de support est une surprise puisque le parcours global de l'espace des motifs est différent dans les deux approches. De plus, nous analysons ici les écarts de quotient entre les deux résultats, pas les écarts en secondes qui eux augmentent de façon croissante. Il est difficile de se prononcer avec certitude si ce comportement est du à une grande efficacité pour des valeurs de k entre 4 et 6 ou plutôt à une moins grande efficacité sur des valeurs de k supérieures à 6. Selon nous, ce comportement est dû à la densité du jeu de données synthétiques qui diminue l'utilité des listes locales de la nouvelle approche qui sont déjà très petites pour les propriétés dès le début (en moyenne pas plus de deux). Dans cette expérimentation, les gains offerts par les nouveaux opérateurs et l'élagage de l'espace de recherche semblent compenser le calcul de toutes les solutions.

Quoi qu'il en soit, nous verrons dans les autres expérimentations qu'en diminuant le seuil de support le ratio évolue en faveur de la nouvelle approche.

Notre seconde expérimentation compare les temps d'exécution sur le premier jeu de données phylogéniques. Il est important de faire remarquer que les différences de per-

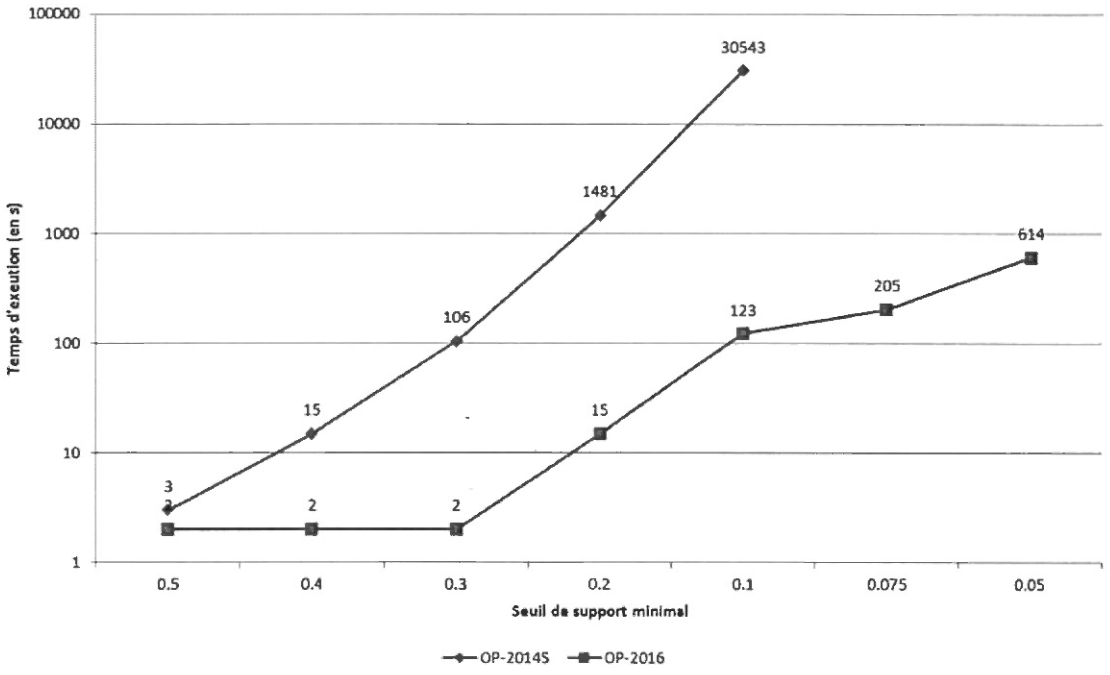

Figure 5.20: Comparaison des approches sur Phylo-Manuel

formances sont tellement importantes que nous avons utilisé une échelle logarithmique pour représenter les temps d'exécution. Nos mesures considèrent les seuils de support allant de 50% à 5%.

Nous remarquons très facilement le coût plus qu'exponentiel de l'abaissement du seuil de support pour l'ancienne approche. La seconde approche quant à elle semble suivre une tendance légèrement sous-exponentielle à partir d'un seuil de 30%. Avec de seuils supérieurs, le temps de fouille est instantané, les temps mesures sont les temps de construction de la base verticale. Le gain de performance de la nouvelle version par rapport à l'ancienne est ici en augmentation constante. En effet, il est d'environ 7,5 fois pour un support minimal de 40% et atteint 248 fois pour un support de 10%. Nous n'avons pas été en mesure de mesurer les temps d'exécution de l'ancienne implémentation pour des support inférieurs à 10%. Nous faisons toutefois figurer les mesures avec la nouvelle approche à titre informatif. Ici nous constatons facilement les gains de

performance apportes à la fois par les nouveaux opérateurs et le nouveau parcours de l'espace des motifs. Ces observations s'alignent avec nos remarques sur l'importance du parcours de l'espace des motifs.

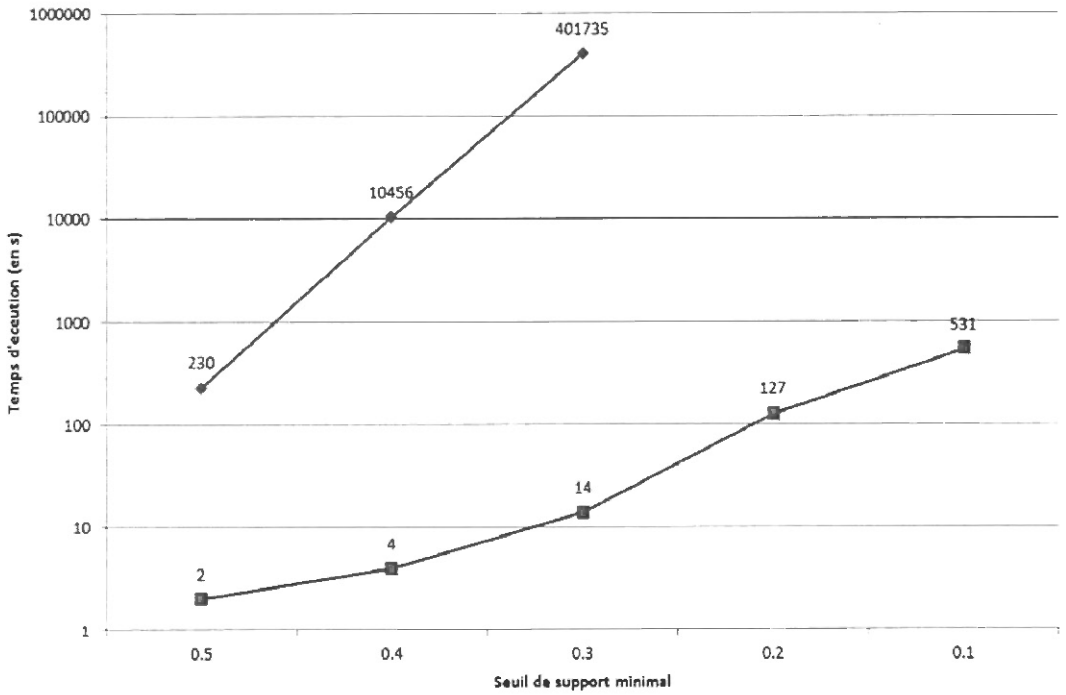

Figure 5.21: Comparaison des approches sur Phylo-Auto

Notre troisième et dernière expérimentation exploite le second jeu de données phylogéniques. Tout comme avec l'expérimentation précédente une échelle logarithmique est aussi nécessaire. Tout d'abord nous tenons à faire remarquer la grandeur des temps d'exécution sur ce jeu de données. En effet, avec un seuil de support de 30%, l'ancienne approche nécessite plus de 400000 secondes, c'est à dire 111 heures ou presque 5 jours. La nouvelle approche ne demande que 14 secondes. Ces gains proviennent en majeure partie de techniques d'élagage de l'espace des motifs. Les gains comparatifs entre les deux approches vont ici de 115 fois pour $minsup$=50% jusqu'au nombre astronomique de 28000 fois pour un seuil de 30%.Ces écarts de performance entre les versions sont principalement à imputer au parcours de l'espace de recherche et à la méthode d'appa-

riement motif-sequence pour l'ajout de propriétés ainsi que bien sur à la représentation verticale de la base de séquences. Les nouveaux opérateurs d'ajout de propriétés sont capables de trouver toutes les solutions pour une position fixe et une propriété fixe en une seule opération alors que l'ancienne approche effectue potentiellement un grand nombre d'allers-retours dans la structure d'appariement puisque les blocs de recherche de sujet et d'objets sont séparés. De plus, la recherche d'une position de triplet compatible est linéaire alors qu'ici grâce à la représentation verticale toutes les positions compatibles pour l'objet sont calculables en une seule opération.

Ces trois expérimentations montrent bien que concevoir une méthode de parcours global de l'espace de recherche des motifs relativement efficace et développer une méthode performante de calcul de support d'un motif est loin d'être suffisant. En effet, comme nous l'avons montré à travers notre travail dans ce mémoire, l'élagage à priori de l'espace de recherche des motifs constitue le point critique lors de la conception d'un nouvel algorithme de fouille de motifs fréquents. Évidemment toutes ces trois conditions (parcours, élagage et calcul de support) doivent être réunies pour développer un algorithme réellement performant.

Malgré tout cela, si nous comparons nos nouveaux résultats avec les résultats observes dans nos expérimentations sur les algorithmes de fouille de motifs séquentiels, nous pouvons constater que nos approches sont loin d'offrir des temps d'exécution compétitifs par rapport aux approches dites plus classiques. Certes, comme nous l'avons montré dans le début de ce travail, les espaces à parcourir sont considérablement plus massifs et plus complexes ici, néanmoins selon nous beaucoup d'améliorations sont encore possibles.

CONCLUSION

5.4 Synthèse

À travers ce travail de mémoire, nous avons successivement proposé deux approches différentes (une originale et une verticale) pour rendre la fouille de motifs séquentiels de données liées plus beaucoup abordable en termes de ressources nécessaires. Nous avons aussi proposé un état de l'art des approches existantes pour la fouille de motifs séquentiels ainsi que pour l'intégration de connaissances du domaine au sein du processus de fouille. En outre, nous avons effectué une comparaison des approches les plus efficaces de motifs séquentiels dans différents cas typiques et réutilisé une partie des observations tirées de ce travail au sein de notre approche verticale. Ces observations nous permettent d'organiser les travaux futurs à apporter sur notre nouvelle approche.

Notre objectif était d'améliorer grandement les performances affichées par xPMiner et ainsi rendre la fouille de motifs séquentiels de données liées réaliste en pratique. Nous pouvons considérer que cet objectif est pleinement atteint même si comme nous l'avons indiqué précédemment beaucoup d'améliorations sont encore possibles.

En termes de gain global, par rapport à la version initiale de xPMiner de 2013, nous avons atteint un gain de plusieurs milliers de fois, ce qui constitue une amélioration plus que conséquente. Ceci nous permet de conclure sur la faisabilité de la fouille de motifs séquentiels de données liées en pratique. A ceci il faut ajouter que nos approches ne sont pas exemptes de défauts et que des travaux futurs viendront les rendre encore plus efficaces. Nous n'avons par conséquent pas le moindre doute sur la capacité à exploiter

Néanmoins, nous n'avons pas montre l'utilité et la pertinence de ces motifs complexes

à extraire. Pour pouvoir justifier les ressources nécessaires à la fouille de ces motifs, nous souhaiterions disposer d'un jeu de données réelles à partir duquel nous puissions trouver quelques motifs à la fois intéressants, surprenants et simples à interpréter. Mais comme nous l'avons fait remarquer précédemment, le point négatif de notre travail est que nous manquons, à l'heure actuelle, de suffisamment de données réelles.

Nous avons donc montré que la taille à priori prohibitive de l'espace des motifs due à l'intégration d'une ontologie dans les séquences utilisateurs reste en réalité tout à fait de l'ordre du gérable compte tenu d'une représentation appropriée et d'une exploration élaguée dans la mesure du possible. En effet, notre approche est aujourd'hui en mesure de fournir des résultats dans des temps relativement acceptables. Mais même si nous en avons diminué l'impact, l'espace des motifs reste colossal. De meilleures approches seront sans aucun doute a développer dans des travaux futurs.

5.5 Travaux futurs

Avec cette seconde approche nous avons conçu et développé des opérateurs efficaces, parallélisables et plus intéressant encore GPU-friendly. Explorer les gains atteignables grâce à la parallélisation de nos opérateurs que ce soit simplement multi-thread, sur un cluster de machines ou sur une voire plusieurs GPU sera l'objet de travaux futurs centrés sur la mise en pratique concrète de nos approches. En outre, ceci pourrait être l'occasion de travailler avec d'autres jeux de données réels. Le manque de données réelles étant la faiblesse principale de notre travail.

À plus court terme, nous sommes en mesure de poursuivre notre travail sur l'élagage de l'espace des motifs mais pas seulement. Dans ce travail, mous avons utilisé des techniques d'élagage des divers espaces de recherche (motifs, séquences et itemsets) mais un grand nombre d'améliorations sont encore possibles surtout sur l'espace des motifs. En effet, dans nos expérimentations sur les motifs séquentiels, nous avons montré

qu'une approche handicapée par une technique reconnue comme inefficace et consommatrice de ressource mais couplée à un parcours de l'espace des motifs judicieuse était en mesure de tenir tête à la meilleure approche connue en 2016. Toutefois nous n'avons pas inclus nos travaux sur cette approche au sein de l'algorithme proposé dans la partie précédente puisque il était d'abord nécessaire de montrer qu'une approche verticale était la plus judicieuse.

Ainsi, nous comptons poursuivre nos travaux sur ces motifs séquentiels de données liées en exploitant pleinement nos toutes nos avancées sur la question du parcours de l'espace des motifs. Selon nous, il est vital d'intégrer les avancées proposées par les approches de type élagage précoce telles que CM-SP, HVSM et notre amélioration de Spami-FTS. L'impact de ces techniques est d'autant plus important qu'ici l'espace des motifs est conséquent. La suppression d'une combinaison le plus tôt possible élimine d'office un nombre beaucoup plus élevé de super-combinaisons que dans une fouille de motifs séquentiels. Nous pensons donc que dans un premier temps, il est nécessaire de trouver tous les 2-motifs de classes fréquents puis toutes les 2-séquences de propriétés fréquentes via les positions de leurs codomaines respectifs. Dans un troisième temps, utiliser ces informations pour trouver tous les triplets fréquents (qui sont des 2-motifs avec une propriété). À partir de cet instant, il est simple de considérer tous les triplets comme des extensions possibles pour les motifs d'ores et déjà fréquents. Les techniques d'élagage des motifs seraient ici focalisée sur les combinaisons fréquentes de classes et de propriété initialement validées compatibles avec la dernière classe ou propriété ajoutée dans un motif. Cependant, nos expérimentations sur les séquences nous ont montre qu'utiliser uniquement les 2-séquences fréquentes commençant par le dernier itemset ajoute à un motif n'est pas suffisant. Il est donc nécessaire d'utiliser cette technique en conjonction avec les listes locales.

À plus long terme, il est crucial de définir des motifs séquentiels de données liées fermés ou du moins maximaux. En effet, même si l'utilisation de hiérarchies nous per-

met d'utiliser des seuils de support minimaux plus élevés que dans une fouille sans hiérarchies (et ainsi en théorie obtenir moins de motifs), l'ajout des propriétés génère potentiellement plusieurs versions du même motif fréquent partageant la même chaine de classes mais avec une propriété différente si ce motif est fréquent. De plus, si plusieurs de ces propriétés sont localement fréquentes, il est possible que la combinaison des deux soit aussi fréquente et donc rajoute une nouvelle fois un autre motif partageant la même chaine de classes. Tout ceci crée des redondances au sein des motifs. À ceci il faut ajouter qu'au delà des gains computationels lors de la fouille, il est bien connu que les motifs fermes fréquents sont beaucoup moins nombreux et n'occasionnent aucune perte d'information par rapport aux motifs fréquents. Bien entendu cette observation est relative à la densité du jeu de donnes. En effet, plus le jeu de données est creux plus la différence se fait sentir en faveur des motifs fermes.

Il serait intéressant d'observer dans quelle mesure ces nouveaux motifs pourraient apporter une efficacité supplémentaire à un classifieur basé sur des séquences fréquentes (Lesh *et al.*, 1999). De même, intégrer des itemsets non singletons dans notre approche pourrait s'avérer utile. Nous pourrions aussi observer leur comportement dans un classifieur à base de règles. Cependant, l'espace des motifs n'en serait que plus conséquent. En outre, calculer et évaluer des règles à partir de nos nouveaux motifs pourrait constituer une façon d'évaluer une ontologie à travers son utilisation voire même à identifier quels seraient les liens triplets manquants entre certains objets.

En outre, définir puis exploiter des générateurs des motifs séquentiels fermes de données liées permettrait de construire des règles d'association beaucoup plus efficaces que simplement avec les motifs fréquents ou fermes (Li *et al.*, 2006).

La dernière branche à explorer selon nous serait la branche des approches de fouille incrémentales (streaming) telles que (El-Sayed *et al.*, 2004; Parthasarathy *et al.*, 1999). En effet, il n'est pas certain que maintenir l'ensemble des motifs en temps réel soit

chose facile en considérant la taille de l'espace de nos motifs. Cependant, notre hypothèse selon laquelle des techniques de fouille originellement conçues pour traiter des itemsets seraient efficaces pour nos motifs s'est avérée correcte. Par conséquent nous pensons qu'une telle approche est sérieusement envisageable.

Pour terminer, maintenant que nous disposons d'une approche proposant des temps d'exécution acceptables, il serait désirable d'inclure les instances de l'ontologie dans le processus de fouille. En effet, notre approche étant de type top-down rajouter les instances dans le cas ou une classe feuille serait localement fréquente ne semble pas être un obstacle majeur et pourrait permettre de disposer de motifs encore plus informatifs puisqu'ils pourraient associer simultanément des classes et des instances.

Pour compenser ces couts supplémentaires il serait avantageux de concevoir des approches pour explorer l'ontologie avec plus de parcimonie. En effet, l'espace des motifs de l'approche actuelle subit pleinement l'épaisseur des deux hiérarchies de l'ontologie. La première passe pour k=1 pourrait servir à déterminer quel serait le niveau initial optimal plutôt que de toujours employer les classes racines. Une approche naïve serait par exemple de ne pas tester directement les motifs composes de classes dont toutes les sous-classes sont déjà fréquentes. Néanmoins, il a notre connaissance il n'existe d'approche miracle pour ce problème, autrement dit, dans certains cas, l'exploration devra remonter dans la hiérarchie. Notre approche ne serait plus de type top-down mais hybride.

Pour terminer, nous mentionnerons que nous avons considéré concevoir notre nouvelle version en explorant des approches basées sur les avancées provenant des triplestores ainsi que celles combinant des alphabets succins avec la recherche de sous-graphes labellisés (Barbay *et al.*, 2012; Ferragina et Manzini, 2001; Brisaboa *et al.*, 2009). À notre connaissance, aucune de ces deux branches n'a été explorée au sein d'une approche de fouille de motifs. Ceci fera sans doute l'objet de travaux futurs.

RÉFÉRENCES

Adda, M. (2008). *Intégration des connaissances ontologiques dans la fouille de motifs séquentiels avec application à la personnalisation web.* (Thèse de doctorat). Thèse de doctorat dirigée par Valtchev, Petko Djeraba, Chaabane et Missaoui, Rokia Informatique Lille 1 2008.

Adda, M., Valtchev, P., Missaoui, R. et Djeraba, C. (2007). Toward sendation Based on Ontology-Powered Web-Usage Mining. *Internet Computing, IEEE*, *11*(4), 45–52.

Adda, M., Valtchev, P., Missaoui, R. et Djeraba, C. (2010). A framework for mining meaningful usage patterns within a semantically enhanced web portal. Dans *Proceedings of the Third C* Conference on Computer Science and Software Engineering*, 138–147. ACM.

Adda, M., Valtchev, P., Missaoui, R., Djeraba, C. *et al.* (2006). Semantically enhanced sequential patterns for content adaptation on the web. *Proceedings of MCeTech*, *6*, 177–191.

Aggarwal, C. C. et Han, J. (2014). *Frequent Pattern Mining* (2014 éd.). Springer.

Agrawal, R. et Srikant, R. (1995). Mining sequential patterns. Dans *Data Engineering, 1995. Proceedings of the Eleventh International Conference on*, 3–14. IEEE.

Agrawal, R., Srikant, R. *et al.* (1994). Fast algorithms for mining association rules. Dans *Proc. 20th int. conf. very large data bases, VLDB*, volume 1215, 487–499.

Anand, S. S., Bell, D. A. et Hughes, J. G. (1995). The role of domain knowledge in data mining. Dans *Proceedings of the fourth international conference on Information and knowledge management*, 37–43. ACM.

Antunes, C. et Oliveira, A. (2004). Sequential pattern mining algorithms : trade-offs between speed and memory.

Ayres, J., Flannick, J., Gehrke, J. et Yiu, T. (2002). Sequential pattern mining using a bitmap representation. Dans *Proceedings of the eighth ACM SIGKDD international conference on Knowledge discovery and data mining*, 429–435. ACM.

Barbay, J., Aleardi, L. C., He, M. et Munro, J. I. (2012). Succinct representation of labeled graphs. *Algorithmica*, *62*(1-2), 224–257.

Berendt, B. (2006). Using and learning semantics in frequent subgraph mining. In *Advances in Web Mining and Web Usage Analysis* 18–38. Springer.

Brisaboa, N. R., Ladra, S. et Navarro, G. (2009). k2-trees for compact web graph representation. Dans *International Symposium on String Processing and Information Retrieval*, 18–30. Springer.

Chambi, S., Lemire, D., Kaser, O. et Godin, R. (2015). Better bitmap performance with roaring bitmaps. *Software : practice and experience*.

Chen, X., Zhou, X., Scherl, R. et Geller, J. (2003). Using an interest ontology for improved support in rule mining. In *Data Warehousing and Knowledge Discovery* 320–329. Springer.

Chiu, D.-Y., Wu, Y.-H. et Chen, A. L. (2004). An efficient algorithm for mining frequent sequences by a new strategy without support counting. Dans *Data Engineering, 2004. Proceedings. 20th International Conference on*, 375–386. IEEE.

El-Sayed, M., Ruiz, C. et Rundensteiner, E. A. (2004). Fs-miner : efficient and incremental mining of frequent sequence patterns in web logs. Dans *Proceedings of the 6th annual ACM international workshop on Web information and data management*, 128–135. ACM.

Febrer-Hernández, J. K., Hernández-Palancar, J., Hernández-León, R. et Feregrino-Uribe, C. (2014). Spami-fts : An efficient algorithm for mining frequent sequential patterns. In *Progress in Pattern Recognition, Image Analysis, Computer Vision, and Applications* 470–477. Springer.

Ferragina, P. et Manzini, G. (2001). An experimental study of an opportunistic index. Dans *Proceedings of the twelfth annual ACM-SIAM symposium on Discrete algorithms*, 269–278. Society for Industrial and Applied Mathematics.

Fortin, S. et Liu, L. (1996). An object-oriented approach to multi-level association rule mining. Dans *Proceedings of the fifth international conference on Information and knowledge management*, 65–72. ACM.

Fournier-Viger, P., Gomariz, A., Campos, M. et Thomas, R. (2014a). Fast vertical mining of sequential patterns using co-occurrence information. In *Advances in Knowledge Discovery and Data Mining* 40–52. Springer.

Fournier-Viger, P., Gomariz, Gueniche, T. A., Soltani, A., Wu., C. et Tseng, V. S. (2014b). Spmf : a java open-source pattern mining library. *Journal of Machine Learning Research (JMLR), 15 : 3389-3393.*

Gouda, K., Hassaan, M. et Zaki, M. J. (2007). Prism : A primal-encoding approach

for frequent sequence mining. Dans *Data Mining, 2007. ICDM 2007. Seventh IEEE International Conference on*, 487–492. IEEE.

Halioui, A. (2017). *EXTRACTION DE FLUX DE TRAVAUX ABSTRAITS À PARTIR DES TEXTES : APPLICATION À LA BIOINFORMATIQUE.* (Thèse de doctorat). Université du Québec à Montréal.

Halioui, A., Martin, T., Valtchev, P. et Diallo, A. B. (2017). Ontology-based workflow pattern mining : application to bioinformatics expertise acquisition. Dans *Proceedings of the Symposium on Applied Computing*, 824–827. ACM.

Hall, M., Frank, E., Holmes, G., Pfahringer, B., Reutemann, P. et Witten, I. H. (2009). The weka data mining software : An update. *SIGKDD Explor. Newsl.*, *11*(1), 10–18. http://dx.doi.org/10.1145/1656274.1656278

Han, J. et Fu, A. (1999). Mining multiple-level association rules in large databases. *Knowledge and Data Engineering, IEEE Transactions on*, *11*(5), 798–805.

Han, J. et Fu, Y. (1995). Discovery of multiple-level association rules from large databases. Dans *VLDB*, volume 95, 420–431.

Han, J., Pei, J., Mortazavi-Asl, B., Chen, Q., Dayal, U. et Hsu, M.-C. (2000a). Freespan : frequent pattern-projected sequential pattern mining. Dans *Proceedings of the sixth ACM SIGKDD international conference on Knowledge discovery and data mining*, 355–359. ACM.

Han, J., Pei, J. et Yin, Y. (2000b). Mining frequent patterns without candidate generation. Dans *ACM SIGMOD Record*, volume 29, 1–12. ACM.

Hsieh, C.-Y., Yang, D.-L. et Wu, J. (2008). An efficient sequential pattern mining algorithm based on the 2-sequence matrix. Dans *2008 IEEE International Conference on Data Mining Workshops*, 583–591. IEEE.

Jin, R., Yang, G. et Agrawal, G. (2005). Shared memory parallelization of data mining algorithms : Techniques, programming interface, and performance. *IEEE Transactions on Knowledge and Data Engineering*, *17*(1), 71–89.

Kraemer, D., Di Jorio, L., Jouve, D., Serra, A., Raissi, C., Laurent, A., Teisseire, M. et Poncelet, P. (2006). Vpsp : extraction de motifs séquentiels dans weka. *Demonstration dans les 22emes journees de Bases de Donnees avancees.*

Lesh, N., Zaki, M. J. et Ogihara, M. (1999). Mining features for sequence classification. Dans *Proceedings of the Fifth ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, KDD '99, 342–346., New York, NY, USA. ACM.

Li, J., Li, H., Wong, L., Pei, J. et Dong, G. (2006). Minimum description length principle : Generators are preferable to closed patterns. Dans *Proceedings of the National Conference on Artificial Intelligence*, volume 21, p. 409. Menlo Park, CA ; Cambridge, MA ; London ; AAAI Press ; MIT Press ; 1999.

Lu, Y. et Ezeife, C. I. (2003). Position coded pre-order linked wap-tree for web log sequential pattern mining. In *Advances in Knowledge Discovery and Data Mining* 337–349. Springer.

Mabroukeh, N. (2011). Semaware : An ontology-based web recommendation system.

Mabroukeh, N. R. et Ezeife, C. I. (2010). A taxonomy of sequential pattern mining algorithms. *ACM Computing Surveys*, *43*(1), 1–41.

Masseglia, F., Cathala, F. et Poncelet, P. (1998). The psp approach for mining sequential patterns. In *Principles of Data Mining and Knowledge Discovery* 176–184. Springer.

Matsubara, Y., Miyazaki, J., Fujisawa, M., Amano, T. et Kato, H. (2011). Cc-paid : A cache-conscious parallel sequential pattern mining algorithm. *IPSJ Transactions on Databases, 4(2) :88-100, Jul 2011.*

Matsubara, Y., Miyazaki, J., Yamamoto, G., Uranishi, Y., Ikeda, S. et Kato, H. (2012). Ccdr-paid : more efficient cache-conscious paid algorithm by data reconstruction. Dans *Proceedings of the 27th Annual ACM Symposium on Applied Computing*, 193–198. ACM.

Mooney, C. H. et Roddick, J. F. (2013). Sequential pattern mining–approaches and algorithms. *ACM Computing Surveys (CSUR)*, *45*(2), 19.

Parthasarathy, S., Zaki, M. J., Ogihara, M. et Dwarkadas, S. (1999). Incremental and interactive sequence mining. Dans *Proceedings of the eighth international conference on Information and knowledge management*, 251–258. ACM.

Pei, J., Han, J., Mortazavi-Asl, B., Pinto, H., Chen, Q., Dayal, U. et Hsu, M.-C. (2001). Prefixspan : Mining sequential patterns efficiently by prefix-projected pattern growth. Dans *icccn*, p. 0215. IEEE.

Pei, J., Han, J., Mortazavi-Asl, B. et Zhu, H. (2000). Mining access patterns efficiently from web logs. In *Knowledge Discovery and Data Mining. Current Issues and New Applications* 396–407. Springer.

Peterson, E. A. et Tang, P. (2008). Mining frequent sequential patterns with first-occurrence forests. Dans *Proceedings of the 46th Annual Southeast Regional Conference on XX*, 34–39. ACM.

Pinto, H., Han, J., Pei, J., Wang, K., Chen, Q. et Dayal, U. (2001). Multi-dimensional

sequential pattern mining. Dans *Proceedings of the tenth international conference on Information and knowledge management*, 81–88. ACM.

Plantevit, M., Choong, Y., Laurent, A., Laurent, D. et Teisseire, M. (2005). M2sp : Mining sequential patterns among several dimensions. In A. Jorge, L. Torgo, P. Brazdil, R. Camacho, et J. Gama (dir.), *Knowledge Discovery in Databases : PKDD 2005*, volume 3721 de *Lecture Notes in Computer Science* 205–216. Springer Berlin Heidelberg

Plantevit, M., Laurent, A. et Teisseire, M. (2006). HYPE : mining hierarchical sequential patterns. Dans *DOLAP '06 : Proceedings of the 9th ACM international workshop on Data warehousing and OLAP*. ACM Request Permissions.

Rabatel, J., Bringay, S. et Poncelet, P. (2010). Contextual Sequential Pattern Mining. *Audio, Transactions of the IRE Professional Group on*, 981–988.

Rabatel, J., Croitoru, M., Ienco, D. et Poncelet, P. (2014). Contextual itemset mining in dbpedia. Dans *LD4KD'2014 : 1st Workshop on Linked Data for Knowledge Discovery with ECML PKDD'2014 : The European Conference on Machine Learning and Principles and Practice of Knowledge Discovery in Databases*, volume 1232, http–ceur. CEUR.

Rabatel, J., Fabregue, M., Bringay, S., Poncelet, P. et Teisseire, M. (2011). Prédiction du grade d'un cancer du sein par la découverte de motifs séquentiels contextuels dans des puces à adn. Dans *ECS'11 : Atelier Extraction de Connaissances et Santé*, 37–48.

Ramezani, R., Saraee, M. et Nematbakhsh, M. A. (2014). Mrar : Mining multi-relation association rules. *Journal of Computing and Security*, *1*(2).

Salvemini, E., Fumarola, F., Malerba, D. et Han, J. (2011). Fast sequence mining based on sparse id-lists. Dans *International Symposium on Methodologies for Intelligent Systems*, 316–325. Springer.

Savary, L. et Zeitouni, K. (2005). Indexed bit map (ibm) for mining frequent sequences. In *Knowledge Discovery in Databases : PKDD 2005* 659–666. Springer.

Song, S., Hu, H. et Jin, S. (2005). Hvsm : a new sequential pattern mining algorithm using bitmap representation. In *Advanced data mining and applications* 455–463. Springer.

Srikant, R. et Agrawal, R. (1996). Mining sequential patterns : Generalizations and performance improvements. Dans *Proceedings of the 5th International Conference on Extending Database Technology : Advances in Database Technology*, EDBT '96, 3–17., London, UK, UK. Springer-Verlag.

Tang, P., Turkia, M. P. et Gallivan, K. A. (2007). Mining web access patterns with first-occurrence linked wap-trees. Dans *SEDE*, 247–252. Citeseer.

Traore, Y., Malo, S., Diop, C. T., Lo, M. et Stanislas, O. (2015). Approche de découverte de nouvelles catégories dans un wiki sémantique basée sur les motifs fréquents. Dans *IC2015*.

Yang, Z. et Kitsuregawa, M. (2005). Lapin-spam : An improved algorithm for mining sequential pattern. Dans *Data Engineering Workshops, 2005. 21st International Conference on*, 1222–1222. IEEE.

Yang, Z., Kitsuregawa, M. et Wang, Y. (2006). Paid : Mining sequential patterns by passed item deduction in large databases. Dans *2006 10th International Database Engineering and Applications Symposium (IDEAS'06)*, 113–120. IEEE.

Yang, Z., Wang, Y. et Kitsuregawa, M. (2007). Lapin : effective sequential pattern mining algorithms by last position induction for dense databases. In *Advances in Databases : Concepts, Systems and Applications* 1020–1023. Springer.

Zaki, M. J. (2001). Spade : An efficient algorithm for mining frequent sequences. *Machine learning*, *42*(1-2), 31–60.

Zaki, M. J., Parthasarathy, S., Ogihara, M., Li, W. *et al.* (1997). New algorithms for fast discovery of association rules. Dans *KDD*, volume 97, 283–286.

Zhou, X. et Geller, J. (2007). Raising, to enhance rule mining in web marketing with the use of an ontology. *Data Mining with Ontologies : Implementations, Findings and Frameworks*, 18–36.