Extraction efficace de motifs séquentiels généralisés : le prétraitement des données

F. Masseglia[1,2] P. Poncelet[3] M. Teisseire[1,4]

[1] LIRMM - [2] PRISM - [3] IUT Aix-en-Provence/LIM - [4] ISIM

[1]LIRMM UMR CNRS 5506
161, Rue Ada
34392 Montpellier Cedex 5, France

[2]Laboratoire PRiSM, Université de Versailles
45 Avenue des Etats-Unis
78035 Versailles Cedex, France

E-mail : {massegli,poncelet,teisseire}@lirmm.fr

Résumé

Proche du problème de la recherche de règles d'association dans de grandes bases de données, l'extraction de motifs séquentiels nécessite de prendre en compte des contraintes temporelles. Initialement introduite par R. Agrawal et R. Srikant, cette problématique est traitée par l'approche GSP (Generalized Sequential Patterns). Se situant dans le même contexte, notre proposition, GTC (Graph for Time Constraints), est basée sur une gestion fine des contraintes temporelles et permet une recherche plus efficace des motifs séquentiels généralisés grâce à un traitement préalable des données.

1 Introduction

Motivé par des problèmes d'aide à la décision, le *data mining*, aussi appelé Extraction de Connaissances dans les Bases de Données (ECBD), a été largement étudié ces dernières années [5]. Dans ce domaine l'extraction de règles d'association a fait l'objet de nombreuses contributions [1, 2, 4, 5, 6, 12, 13, 16] contrairement aux motifs séquentiels [3, 15, 7, 17] bien que ces derniers soient d'un grand intérêt pratique (e.g. gestion des alarmes dans les réseau de télécommunication, analyse des comportements de clients, analyse des accès dans un serveur Web, ...). La problématique de la recherche de motifs séquentiels consiste à détecter des comportements typiques dans le temps. Dans un tel contexte, un motif séquentiel peut être par exemple "5% des clients qui achètent les produits A et B, le même jour, achètent le produit C moins de 15 jours après.".

Le problème de l'extraction de motifs séquentiels dans une base de données a été introduit dans [3] où un algorithme efficace, appelé AprioriAll, est proposé. Considérons un ensemble de séquences appelées séquences de données, chaque séquence est une liste de transactions composées d'une estampille et d'un ensemble d'items. Il s'agit alors de rechercher toutes les séquences dont le nombre d'occurrences dans la base est supérieur à un support spécifié par l'utilisateur. Les séquences vérifiant cette contrainte sont appelées des *séquences fréquentes* ou *motifs séquentiels*. Dans [15], la problématique est étendue pour prendre en compte des contraintes temporelles et des taxonomies (hiérarchie *is-a*). Les séquences fréquentes vérifiant, en plus du support, ces

contraintes sont appelées *motifs séquentiels généralisés*. De plus, l'algorithme proposé, GSP (Generalized Sequential Patterns), est 20 fois plus rapide qu'AprioriAll. Dans [10], nous avons proposé un nouvel algorithme, PSP (Prefix-tree for Sequential Patterns), qui reprend les principes fondamentaux de GSP mais utilise une structure de données différente pour optimiser la recherche des motifs. Au cours des différentes expérimentations, nous avons montré que PSP est plus performant que GSP.

Notre proposition se situant dans cette problématique, le paragraphe suivant présente les différents concepts fondamentaux associés aux motifs séquentiels et plus spécifiquement aux motifs séquentiels généralisés. Traitant ces derniers, les algorithmes GSP et PSP sont ensuite décrits paragraphe 3. Le paragraphe 4 détaille notre approche, l'algorithme GTC (Graph for Time Constraints) basé sur une gestion fine des contraintes temporelles, qui permet une recherche plus efficace des motifs séquentiels généralisés grâce à un traitement préalable des données. Les évaluations des différentes approches sont ensuite proposées au paragraphe 5. Le paragraphe 6 conclue cet article et souligne les points forts de notre approche.

2 Définitions préliminaires

Initialement introduite dans [3], la notion de séquence est définie de la manière suivante :

Définition 1 Une *transaction* constitue, pour un client C, l'ensemble des items achetés par C à une même date, elle est décrite sous forme d'un triplet : <id-client, id-date, itemset>. Un *itemset* est un ensemble non vide d'items noté $(i_1 i_2 \dots i_k)$. Une *séquence* ou motif séquentiel est une liste ordonnée, non vide, d'itemsets notée $< s_1 s_2 \dots s_n >$ où s_j, $1 < j < n$, est un itemset (une séquence est donc une suite de transactions avec une relation d'ordre entre les transactions), représentant l'ensemble des achats d'un client. Une séquence regroupant k items est dîte une *k-squence*. Soit T_1, $T_2, \dots,$ T_m les transactions d'un client, ordonnées par date d'achat croissante et soit $itemset(T_i)$, $1 < i < m$, l'ensemble des items correspondants à T_i, alors la séquence de données de ce client est $< itemset(T_1)\ itemset(T_2) \dots\ itemset(T_m) >$.

Exemple 1 *Soit C un client et S=<* (30) (40 50) (80) *>, la séquence de données représentant les achats de ce client. S peut être interprétée par "C a acheté l'item 30, puis, de façon simultanée, les items 40 et 50 et enfin l'item 80"* □

Définition 2 Soit $s_1 =< a_1 a_2 \dots a_n >$ et $s_2 =< b_1 b_2 \dots b_m >$ deux séquences de données. s_1 est *incluse* dans s_2 ($s_1 \prec s_2$) si et seulement si il existe $i_1 < i_2 < \dots < i_n$ des entiers tels que $a_1 \subseteq b_{i_1},\ a_2 \subseteq b_{i_2}, \dots\ a_n \subseteq b_{i_n}$.

Exemple 2 *La séquence s1 =<* (3) (4 5) (8) *> est incluse dans la séquence s2 =<* (7) (3 8) (9) (4 5 6)(8) *> (i.e. s1* $\prec$ *s2) car* (3) $\subseteq$ (3 8), (4 5) $\subseteq$ (4 5 6) *et* (8) $\subseteq$ (8). *En revanche <* (3) (5) *>*$\not\prec$*<* (3 5) *> (et vice versa)* □

De manière générale, l'approche retenue par la plupart des algorithmes pour rechercher des itemsets fréquents ou des motifs séquentiels consiste à chercher l'ensemble L_1 d'items vérifiant le support de la base, puis à générer, à partir de celui-ci, un ensemble C_2 de 2-items appelé candidats. Un nouveau parcours de la base permet de ne conserver que les candidats dont le support est supérieur au support minimum formant ainsi l'ensemble L_2 des fréquents de taille 2. A partir de L_2, de nouveaux candidats sont générés constituant l'ensemble C_3 des candidats de taille 3. Le support associé à ces candidats est calculé par un autre examen de la base. Le principe énoncé s'applique tant qu'il est possible de trouver des fréquents parmi les candidats générés.

Telle qu'elle a été introduite, la notion de séquence présente une rigidité rédhibitoire pour certains types d'application. En effet, si l'intervalle de temps entre deux transactions est suffisamment court, il peut être alors envisageable de traiter celles-ci comme une unique transaction. De façon similaire, un intervalle trop long peut rendre inintéressant l'étude de la séquence des deux transactions. C'est pourquoi la notion de *séquences généralisées* a été introduite dans [15] pour pallier ces restrictions en proposant l'introduction de contraintes temporelles comme l'indique la définition suivante :

Définition 3 Soient *minGap* une durée minimum, *maxGap* une durée maximum et *windowSize* une durée de rectification, spécifiées par l'utilisateur. Une séquence de données $d =< d_1 ... d_m >$ supporte une séquence $s =< s_1 ... s_n >$ s'il existe des entiers $l_1 \leq u_1 < l_2 \leq u_2 < ... < l_n \leq u_n$ tels que :

1. $s_i \subset \cup_{k=l_i}^{u_i} d_k$, $1 \leq i \leq n$;
2. date(d_{u_i}) - date(d_{l_i}) $\leq windowSize$, $1 \leq i \leq n$;
3. date(d_{l_i}) - date($d_{u_{i-1}}$) $> minGap$, $2 \leq i \leq n$;
4. date(d_{u_i}) - date($d_{l_{i-1}}$) $\leq maxGap$, $2 \leq i \leq n$.

Le *support* de s, noté $supp(s)$, est le pourcentage de toutes les séquences dans D qui supportent s. Si $supp(s) \geq minsupp$, avec une valeur de support minimum $minsupp$, la séquence s est dite *fréquente*.

Ce concept de motifs séquentiels généralisés permet une manipulation plus souple des transactions des clients dans la mesure où il est désormais possible de :

- regrouper des itemsets lorsque leurs dates sont assez proches via la contrainte de windowSize,
- considérer des itemsets comme trop rapprochés pour apparaître dans la même séquence fréquente avec la contrainte de minGap,
- considérer des itemsets comme trop éloignés pour apparaître dans la même séquence fréquente avec la contrainte de maxGap.

Exemple 3 *Considérons l'exemple suivant qui illustre la prise en compte des contraintes de temps sur une base de données D réduite à 4 clients (C.f. Figure 1). Avec un support minimum strictement supérieur à 50%, i.e. pour qu'une séquence soit supportée elle doit être vérifiée par au moins deux séquences de la base de données, les motifs séquentiels sont, d'après la définition 3 :* $< (20)\ (90) >$, $< (30)\ (90) >$ *et* $< (30)\ (40, 70) >$, *les deux premières car elles peuvent être retrouvées pour les deux clients* C_1 *et* C_4, *et la dernière car elle intervient pour les clients* C_2 *et* C_4.
Si nous considérons une fenêtre de temps de 2 jours (windowSize = 2 jours), nous pouvons regrouper ensemble tous les items qui ont été achetés dans un intervalle de 2 jours même s'ils sont dans des transactions différentes. Dans ce cas, une nouvelle séquence fréquente $< (20\ 30)\ (90) >$ *est extraite car elle se retrouve dans la première transaction de* C_4 *tout en étant détectée pour* C_1 *(avec un couple de transactions respectant le windowSize).*
Considérons maintenant, du point de vue de l'utilisateur, que deux itemsets extraits successivement ne sont pas intéressant lorsqu'ils sont séparés par un intervalle de temps de 15 jours ou plus. Cette contrainte de maxGap a pour effet d'éliminer $< (30)\ (40, 70) >$ *de l'ensemble des séquences fréquentes parce que, pour le client* C_2, *il y a 17 jours entre les deux transactions. La*

séquence de données de C_2 ne satisfait pas la contrainte de maxGap et la contrainte de support minimal n'est donc plus vérifiée.
Considérons maintenant la séquence extraite via la contrainte de windowSize, i.e. $<(20\ 30)\ (90)>$, *son premier itemset est composé de deux items différents intervenus dans un intervalle de temps de 2 jours. Si nous considérons l'item* (30), *nous voyons qu'il existe un intervalle de temps de 14 jours avec* (90). *Cependant, le délai observé est de 16 jours entre les items* (20) *et* (90) *donc la contrainte de maxGap n'est pas vérifiée. La séquence est alors supprimée de l'ensemble des séquences fréquentes* □

Client	Date	Items	Client	Date	Items
C_1	01/01/1999	20,60	C_3	05/01/1999	30,50,70
C_1	02/02/1999	20	C_3	12/02/1999	10,20
C_1	04/02/1999	30			
C_1	18/02/1999	80,90			
C_2	11/01/1999	10	C_4	06/02/1999	20,30
C_2	12/01/1999	30	C_4	07/02/1999	40,70
C_2	29/01/1999	40,60,70	C_4	08/02/1999	90

FIG. 1 – *Base de données exemple*

3 Travaux antérieurs

Dans cette section, nous présentons les travaux associés aux motifs séquentiels généralisés et plus particulièrement les algorithmes GSP et PSP qui serviront de comparaison à notre proposition. Une discussion fera ensuite un bilan critique des approches GSP et PSP.

3.1 L'algorithme GSP - Generalized Sequential Patterns

Pour construire les séquences candidates et les séquences fréquentes, l'algorithme GSP [15] procède selon différentes étapes comme la plupart des algorithmes de recherche des itemsets fréquents (C.f. Paragraphe 2). Il s'agit, tout d'abord, de calculer le support de chaque item de la base. A la fin du parcours de celle-ci, les items fréquents, i.e. vérifiant la contrainte de support minimum, sont découverts. Ils sont considérés comme des 1-séquences fréquentes (séquence composée d'un itemset réduit à un singleton). Cet ensemble initial est le point de départ de la deuxième étape. L'ensemble des 2-séquences candidates est alors construit en combinant tous les couples d'items fréquents, à la fois dans la même transaction et dans une transaction séparée. A partir de ce point, toute étape k ($k > 2$) réalise, à partir d'un ensemble de (k-1)-séquences fréquentes, les deux sous-étapes suivantes :

- La première sous-étape (*join phase*) concerne la génération des candidats. L'idée principale est de retrouver parmi toutes les séquences dans L_{k-1}, des couples de séquences (s, s') tels qu'en éliminant le premier item de s et le dernier item de s', nous retrouvions les deux mêmes sous-séquences. Quand un couple de séquence (s, s') vérifie la condition, un nouveau candidat est construit en ajoutant le dernier item de s' à s. Dans la séquence candidate, ajoutée à C_k, le découpage des transactions est respecté.
- La seconde sous-étape (*prune phase*) a pour but d'obtenir l'ensemble L_k des k-séquences fréquentes en éliminant de C_k les séquences ne vérifiant pas le support minimum. Il s'agit donc de compter le nombre de séquences candidates intervenant dans les séquences de données.

Les séquences candidates sont stockées dans une structure de *hash-tree* afin d'assurer un accès efficace. Plus précisément, les séquences candidates sont stockées dans les feuilles de l'arbre alors que les nœuds intermédiaires contiennent des tables de hachage pour optimiser la recherche. Chaque séquence de données d de la base de données est hachée pour trouver les candidats contenus dans d. En parcourant une séquence de données, les contraintes de temps doivent être également prises en compte. Ceci est réalisé en parcourant l'arbre de "haut en bas" et de "bas en haut" afin d'appliquer une première fois les contraintes de temps. Le parcours de l'arbre permet d'obtenir des ensembles de feuilles contenant des candidats potentiels. A partir de ces feuilles, GSP recherche tous les candidats effectivement présents dans la séquence de données et qui vérifient les contraintes de temps. Pour cela, chaque candidat est stocké dans une nouvelle structure et est comparé avec la séquence de données. La comparaison a lieu en parcourant en avant et en arrière la séquence candidate et la séquence de données. La phase "avant" est réalisée en examinant les items progressivement. Pendant cette phase, la condition de minGap est appliquée pour éliminer les itemsets qui sont trop proches de leur précédent. Tout en parcourant les itemsets, windowSize est appliqué pour redéfinir les limites des transactions. La phase "retour arrière" est nécessaire lorsque la contrainte de maxGap n'est plus vérifiée. Dans ce cas, il est nécessaire d'éliminer tous les itemsets pour lesquels maxGap n'est pas vérifié et de recommencer un parcours avant à partir du dernier item qui vérifiait la condition de maxGap. Toutes les séquences fréquentes sont alors stockées dans une nouvelle structure de hash-tree multi-niveau [14] pour permettre d'optimiser la génération des candidats.

Exemple 4 *La figure 2 illustre la façon dont GSP gère la structure de hash-tree. Les deux arbres servent à conserver les mêmes candidats mais les feuilles de l'arbre du haut peuvent contenir plus de deux séquences candidates alors que dans l'arbre de droite seulement deux séquences candidates peuvent être stockées dans la même feuille. Nous remarquons que la feuille qui porte l'étiquette 20 dans l'arbre de gauche (le deuxième fils de la racine) est utilisée pour stocker les séquences candidates* $<(20\ 30\ 40)>$ *et* $<(20)\ (30\ 40)>$*. Quand GSP atteint cette feuille, il n'a aucun moyen de savoir si c'est la séquence* $<(20\ 30\ 40)>$ *ou bien la séquence* $<(20)\ (30\ 40)>$ *qui l'a conduit jusqu'à cette feuille. C'est pour cette raison que GSP doit tester les séquences candidates contenues dans les feuilles atteintes afin de savoir quel(s) support(s) incrémenter* □

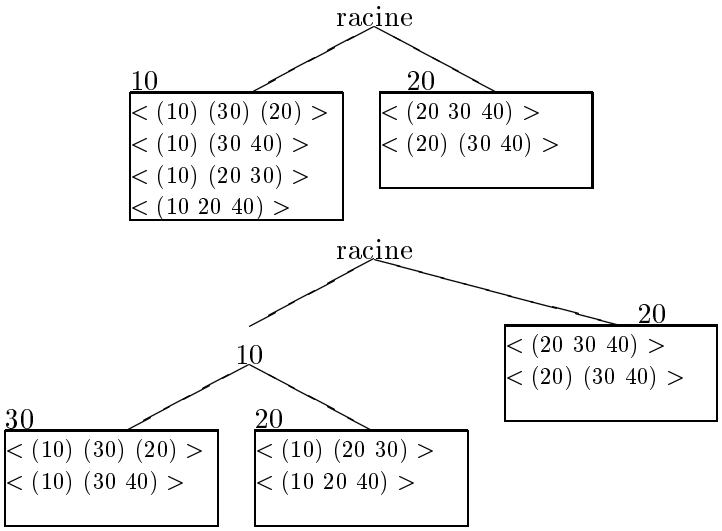


FIG. 2 – *La structure hash-tree de GSP*

3.2 L'algorithme PSP - Prefix-tree for Sequential Patterns

Dans [10], nous avons proposé l'approche PSP (*Prefix-tree for Sequential Patterns*) dont l'originalité est d'utiliser une structure différente de celle adoptée par GSP. C'est ainsi qu'une nouvelle organisation des séquences candidates permet de trouver plus efficacement l'ensemble des candidats inclus dans une séquence de données. En effet, la structure de hash-tree utilisée dans GSP consiste à mettre en commun des préfixes sans faire de distinction entre deux items du même itemset et deux items avec changement d'itemsets. De ce fait, il se voit contraint, lorsqu'il atteint une séquence, de tester si les dates des items qu'elle représente sont adéquats par rapport à la séquence de données testée.
Par contre, la structure de données utilisée par PSP est une structure de *prefix-tree*, proche de celle utilisée dans [11]. Elle factorise les séquences candidates selon leur préfixe et tient compte des éventuels changements d'itemsets. Nous avons montré dans [9] que la structure prefix-tree était plus efficace en mémoire que celle de hash-tree utilisée par GSP. En outre, la structure utilisée offre deux avantages principaux :

1. les séquences fréquentes sont stockées dans le même arbre que les séquences candidates afin d'optimiser la génération des candidats ;
2. contrairement à GSP qui nécessite une phase supplémentaire pour gérer les contraintes de temps, celles-ci peuvent être prises en compte directement lors du parcours de l'arbre.

De manière générale, à la $k^{ième}$ étape, l'arbre est d'une profondeur k. Il stocke toutes les k-séquences candidates de la manière suivante : chaque branche, de la racine à une feuille correspond à une séquence candidate et en considérant une simple branche, chaque nœud à la profondeur l ($k \geq l$) correspond au $l^{ième}$ item de la séquence. En outre, le support de la séquence de la racine à une feuille est associé à chaque nœud terminal. Pour distinguer les itemsets à l'intérieur d'une séquence (par exemple, (10 20) et (10)(20)) les fils d'un nœud sont séparés en deux catégories : "*same-transaction*" ou "*other-transaction*" (branche en traits pleins dans la figure 3).

Exemple 5 *Considérons l'ensemble de 2-séquences fréquentes suivant :* $L_2 = \{< (10)\ (30) >, < (10)\ (40) >, < (30)\ (20) >, < (30\ 40) >, < (40\ 10) >\}$. *Il est organisé dans la structure prefix tree comme illustré figure 3. Chaque nœud terminal contient un item et un compteur. Si nous considérons le nœud correspondant à l'item* **40**, *la valeur 2 associée indique que deux occurrences de la séquence* $\{< (10)\ (40) >\}$ *ont déjà été détectées. L'item 20, fils de la racine, est un fréquent de longueur 1* □

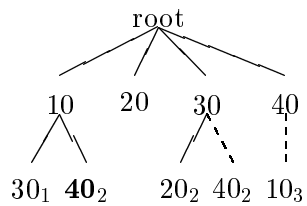


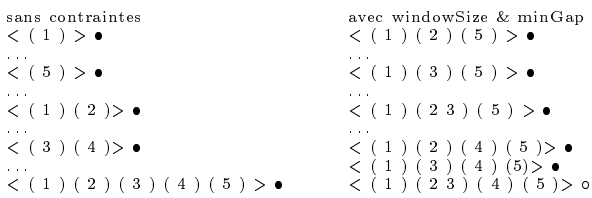
FIG. 3 – *La structure de prefix-tree de PSP*

3.3 Discussion et Problématique

Si la structure utilisée dans PSP semble bien adaptée au problème de la recherche de séquences généralisées, il semble, en revanche, que l'algorithme mis en œuvre n'exploite pas toutes ses possibilités. En effet, l'algorithme PSP peut être amélioré dans certains cas. Comme illustration,

considérons la séquence de données suivante : < (1) (2) (3) (4) (5) > extraite de la base représentée par la figure 5.

Avec windowSize=1 et minGap=1, l'algorithme va tester les combinaisons suivantes afin de vérifier si elles sont susceptibles d'incrémenter le support d'une séquence candidate :



Nous remarquons alors que les séquences marquées par un • sont incluses dans la séquence marquée par un ∘. En effet si une séquence candidate fait partie du support pour < (1) (2 3) (4) (5) > alors elle fait également partie du support pour < (1) (2) (4) (5) > et < (1) (3) (4) (5) >, ce qui rend le test de ces deux séquences inutile car elles sont incluses dans une autre plus générale.

Client	Date	Item
C1	01/04/99	1
C1	03/04/99	2
C1	04/04/99	3
C1	07/04/99	4
C1	17/04/99	5

FIG. 4 – *Exemple de base de données*

En fait, le problème soulevé nous amène à nous poser la question suivante : comment réduire le temps d'analyse d'une séquence de données par rapport à l'arbre des séquences candidates. Pour cela, il est nécessaire de résoudre les deux points suivants :

- le parcours de l'arbre des séquences candidates doit être indépendant des contraintes de temps définies par l'utilisateur. Cela implique un parcours sans retour arrière permettant tout de même de vérifier toutes les combinaisons de windowSize possibles qui sont, pour l'instant, calculées "à la volée" ;
- ce parcours doit vérifier le moins de combinaisons possibles, surtout si un certain nombre de ces combinaisons suffit à tester l'ensemble des candidats nécessaires, c'est à dire un ensemble de combinaisons qui regroupe toutes celles possibles (comme {< (1) (2 3) (4) (5) >} dans l'exemple précédent).

Ces deux points essentiels sont atteints par l'algorithme GTC, décrit de façon détaillée dans le paragraphe suivant.

4 *GTC* : le Graphe solution aux traitements des Contraintes Temporelles

Dans cette section, nous détaillons l'algorithme GTC. Nous présentons tout d'abord, paragraphe 4.1, l'algorithme GTC_1 chargé de gérer les contraintes de minGap et maxGap. Le paragraphe

4.2 décrit la façon dont l'algorithme évolue vers GTC pour intégrer le problème de windowSize tout en gardant les avantages précédents.

4.1 Prise en compte de minGap et maxGap

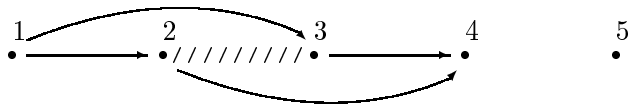


FIG. 5 – *Visualisation d'une séquence de données*

Reprenons le problème de la contrainte de minGap et maxGap sans tenir compte, pour l'instant, de windowSize. L'idée directrice qui nous intéresse ici consiste à dire que le parcours de l'arbre des séquences candidates doit être simplifié. Pour cela nous proposons de calculer à l'avance les combinaisons de séquences issues de la prise en compte des contraintes de temps sur une séquence de données.
Considérons la représentation graphique d'une séquence de données (C.f. Figure 5) correspondant à la base de la figure 4. Le séparateur //////// entre deux items a et b indique que a et b sont trop proches au sens de la contrainte de minGap. En supposant une contrainte de maxGap égale à 9, sa prise en compte a pour effet de séparer la séquence de données en deux sous-séquences de données sur lesquelles vont s'effectuer les calculs: (1, 2, 3, 4) et (5).
Si nous observons uniquement le sous-graphe limité aux sommets (1, 2, 3, 4), nous pouvons extraire deux séquences qu'il est utile de chercher dans l'arbre des séquences candidates: < (1) (2) (4) > et < (1) (3) (4) >, qui correspondent aux deux chemins de source "1" du graphe.

Problème des chemins de taille maximale respectant minGap

Définition 4 Une *sous-séquence de données* d est une séquence de données respectant la contrainte maxGap. d peut être décrite par un graphe orienté acyclique, le *graphe de séquence* $G_d(S, A)$, où S est l'ensemble des sommets et A l'ensemble des arcs du graphe. G_d est tel que: les sommets sont les items de d, tout chemin ayant un sommet source pour origine et un sommet puits pour arrivée n'est inclus dans aucun autre et toute séquence candidate c supportée par d est incluse dans un tel chemin au moins.

Définition 5 Un *chemin de séquence* est un chemin du graphe de séquence G_d ayant pour sommet de départ un sommet source de G_d et pour sommet d'arrivée un sommet puits. L'ensemble SP_d des chemins de séquence représente les sequences à tester dans l'arbre des séquences candidates As pour une séquence de données d. Quelle que soit c une séquence candidate sous-séquence de d, il existe p appartenant à SP_d, tel que p supporte c.

Exemple 6 *Considérons la base de données représentée par la figure 6. Pour un minGap de 2, la figure 7 représente le graphe de séquence correspondant, déduit de la séquence de données du client* $C1$ □

Nous sommes donc conduits à résoudre le "Problème des chemins de taille maximale respectant minGap" qui consiste à énumérer, pour une sous-séquence de données d, l'ensemble SP_d des

Client	Date	Items
C1	01/04/99	1
C1	05/04/99	2
C1	06/04/99	3, 4
C1	07/04/99	5
C1	09/04/99	6

FIG. 6 – *Exemple de base de données*

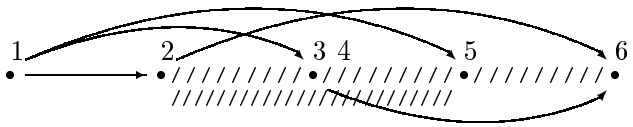


FIG. 7 – *Un minGap plus contraignant*

chemins de séquence à tester dans l'arbre des séquences candidates As et vérifiant les deux propriétés suivantes :

1. $\forall s1, s2 \in SP_d / s1 \not\subset s2$.
2. $\forall c \in As/d$ *supporte* c, $\exists p \in SP_d / p$ *supporte* c.
3. $\forall p \in SP_d, \forall c \in As/p$ *supporte* c, *alors* d *supporte* c.

Ce problème est résolu par l'algorithme GTC_1 qui fournit une solution optimale (C.f. Théorème 1) en $O(n^2)$ (C.f. Théorème 2).

Construction du graphe de séquence

Pour trouver l'ensemble complet des chemins minimaux, GTC_1 utilise la structure de données suivante: un graphe orienté dont chaque sommet représente un itemset et possède un vecteur *isPrec* de taille $|S|$ indiquant pour ce sommet quel autre sommet le précède par un chemin quelconque (il s'agit donc d'un vecteur de booléens qui indique au sommet x, pour chaque sommet y si y est prédécesseur de x ou non). L'algorithme de construction du graphe explore chaque sommet x pour lui appliquer une méthode qui peut se décomposer en deux étapes :

1. **Phase de propagation** : l'algorithme cherche le premier sommet y qui vérifie $(y.date() - x.date() > minGap)$ [1] et construit l'arc (x, y). Ensuite pour chaque sommet z tel que $(z.date() - y.date() > minGap)$ l'algorithme met à jour $z.isPrec[x]$ pour conserver le fait que x atteindra z quoiqu'il arrive en passant par y. Soit Z l'ensemble des sommets z ainsi traités.
2. **Phase de la main tendue** : le but dans cette phase est d'utiliser l'information "x atteindra $z(z \in Z)$" pour construire les arcs (x, t) sachant que $(t.date() - x.date() > minGap)$ et $t.isPrec[x] \neq 1$ (donc $t \notin Z$ et x n'atteindra pas t si l'on ne construit pas l'arc (x, t)).

Une fois l'algorithme GTC_1 appliqué à une sous-séquence de données d, l'ensemble SP_d des séquences à tester dans l'arbre des séquences candidates se construit en cherchant dans le graphe tous les chemins de séquence.

1. *date*() est une fonction permettant d'accéder à la date de transaction de l'item ou de l'itemset.

function GTC_1
Input : une sous-séquence de données d
Output : G_d(S,A) le graphe de séquence de d
foreach *itemset* $i \in d$ **do**
$S = S \cup \{i\}$;
endforeach
foreach $x \in S$ **do**
//Phase de propagation
$y = x$;
while $y.date() - x.date() < minGap$ **do** $y++$;
$A = A \cup \{x, y\}$;
$i_p = \{i \in S / i.date() - y.date() > minGap\}$;
foreach $z \in i_p$ **do**
$z.isPrec[x] = 1$;
endforeach
// Phase de la main tendue
$j_p = \{j \in S / j.date() > y.date()$ *and* $j.isPrec[x] = 0\}$;
foreach $t \in j_p$ **do**
$A = A \cup \{x, t\}$;
endforeach
endforeach
end function GTC_1

Algorithme 1: GTC_1, *solution pour minGap*

Exemple 7 *La figure 8 représente quatre étapes de l'algorithme de construction du graphe.*

- $x = 1$*: Première phase. l'algorithme cherche le premier sommet y tel que* $y.date() - x.date() > minGap$*, il s'agit du sommet* 2*. Ensuite les sommets* 4 *et* 5 *étant accessibles depuis le sommet* 2 *(i.e. les arcs* $(2, 4)$ *et* $(2, 5)$ *respectent la contrainte de minGap), leur composant isPrec est mis à jour (*$4.isPrec[1] = 1$ *et* $5.isPrec[1] = 1$*) pour indiquer que ces sommets sont accessibles à partir de* 1 *mais par l'intermédiaire d'un chemin plus long que l'arc* $(1, 4)$ *ou* $(1, 5)$*.*
Deuxième phase : pour tous les sommets t successeurs de 2 *qui vérifient* $minGap(x, t)$ *et tels que* $t.isPrec[1] = 0$*, l'algorithme construit l'arc* (x, t) *(ici seul le sommet* 3 *repond aux deux critères).*
- $x = 2$*: Première phase.* 4 *est le premier sommet accessible depuis* 2 *donc l'arc* $(2, 4)$ *est construit.* 5 *est le seul successeur de* 4 *donc* $5.isPrec[2] = 1$*. Deuxième phase : il n'y a aucun sommet t tel que* $minGap(2, t)$ *et* $t.isPrec[2] = 0$ *soient vérifiées.*
- $x = 3$*: Première phase.* 4 *est le premier sommet accessible depuis* 3 *donc l'arc* $(3, 4)$ *est construit.* 5 *est le seul successeur de* 4 *donc* $5.isPrec[3] = 1$*. Deuxième phase : il n'y a aucun sommet t tel que* $minGap(3, t)$ *et* $t.isPrec[3] = 0$ *soient vérifiées.*
- $x = 4$*: Première phase. Nous construisons l'arc* $(4, 5)$*. La deuxième phase est annulée car 5 n'a pas de successeur (au sens de la datation)* □

Théorème 1 *L'algorithme* GTC_1 *construit exactement toutes les solutions de la plus grande taille possible pour le problème des chemins respectant minGap.*

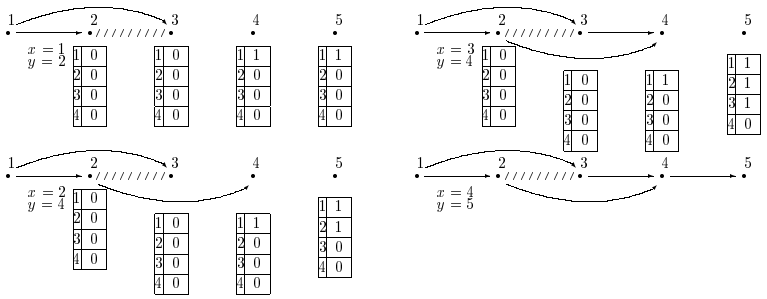


FIG. 8 – *Quatre étapes de GTC_1*

Démonstration : Nous prouvons, dans un premier temps, que l'inclusion de chemins de séquence est impossible après l'exécution de GTC_1. Ensuite, nous montrons que pour toute séquence candidate c supportée par une séquence de données d, il existe un chemin de séquence dans le graphe de séquence calculé à partir de d, qui représente une séquence support pour c.
Supposons qu'il existe $s1$ et $s2$, deux chemins de séquence dans SP_d, l'ensemble des séquences à tester, tels que $s1 \subset s2$. Cela voudrait dire que le graphe de séquence contient le sous-graphe représenté par la figure 9, c'est à dire qu'il existerait un chemin $(a, \ldots, c)$ de longueur ≥ 2 et un arc (a, c). Or, si le chemin (a, c) existe, nous avons $c.isPrec[a] = 1$. Il y a en effet deux possibilités d'avoir un chemin de longueur ≥ 1 de a vers b : un arc (a, b) ou un chemin $(a, \ldots, b)$. Dans le premier cas, $c.isPrec$ est immédiatement mis à jour par l'instruction $c.isPrec[a] \leftarrow 1$ et dans le second cas il existe a' un sommet du chemin $(a, \ldots, b)$ tel que l'arc (a, a') fait partie de ce chemin, auquel cas l'instruction $c.isPrec[a] \leftarrow 1$ a eu lieu lors de la construction de l'arc (a, a'). Donc, après la construction du chemin $(a, \ldots, b, \ldots, c)$ nous avons bien $c.isPrec[a] = 1$ et la construction de l'arc (a, c) est rendue impossible. Le sous-graphe représenté par la figure 9 est donc impossible après l'exécution de GTC_1.
Pour montrer que pour toute séquence candidate c supportée par une séquence de données d, il existe un chemin dans le graphe de séquence calculé à partir de d qui représente une séquence support pour c, nous allons exploiter le fait que GTC_1 explore tous les sommets du graphe et ne construit pas de séquences incluses. L'algorithme parcourt tous les sommets du graphe, ce qui implique que si un chemin respectant minGap contient le sommet x, alors ce chemin fait partie du graphe après l'exécution de GTC_1 (même si il est restreint au seul sommet x). Tous les sommets font partie d'un chemin, l'inclusion de chemin est impossible et si deux chemins $(x, \ldots, y)(y', \ldots, z)$ peuvent être fusionnés (i.e. $y'.date() - y.date() > minGap$), ils le seront car l'algorithme en explorant le sommet y va construire l'arc (y, y') (ce raisonnement peut s'étendre à "si deux chemins $(x, \ldots, y)(y', \ldots, z)$ peuvent être fusionnés (i.e. $y'.date() - y.date() > minGap$) ils le seront car l'algorithme après avoir exploré le sommet y va construire le chemin $(y, \ldots, y')$").
L'algorithme GTC_1 construit donc exactement toutes les solutions de la plus grande taille possible pour le problème des chemins respectant minGap □

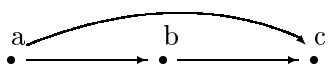


FIG. 9 – *Schéma d'inclusion minimum*

Théorème 2 *L'algorithme GTC_1 a une complexité en $O(n^2)$.*

Démonstration : Le pire des cas se présente lorsque minGap=0. Il est inutile d'utiliser GTC_1 dans un tel cas de figure, dans la mesure où PSP est capable de résoudre le problème aussi bien sinon mieux que GTC_1 (C.f. Paragraphes 5 et 6). Pourtant nous analysons la complexité de GTC_1 malgré cet aspect car il sera indispensable lors de son évolution vers la prise en compte de windowSize.
Si minGap=0, GTC_1 parcours le graphe sommet par sommet et pour chaque sommet x, évalue la distance entre x et chacun de ses successeurs y. Si minGap=0 alors le premier successeur y de x rencontré est pris en compte. A partir de y, GTC_1 considère tous les successeurs de y à deux reprises (une fois pour y, i.e. la phase de propagation et une fois pour x, i.e. la phase de la main tendue).
Le nombre d'opérations nécessaires peut alors s'exprimer sous la forme d'une suite arithmétique : $\sum_{k=1}^{n-1} 2k - 1$.

4.2 Prise en compte de minGap, maxGap et windowSize

Le nombre de séquences incluses qui pose problème, comme nous l'avons vu dans le paragraphe 4, est encore plus important si nous considérons la contrainte de windowSize. En effet avec la base de données représentée par la figure 10, windowSize=5 et mingap=1, l'algorithme PSP de la première approche teste les séquences suivantes (nous ne considérons ici que les séquences calculées avec windowSize, le problème de minGap et maxGap ayant été vu dans le paragraphe 4.1) :
< (1) (2) (3) (5) (6) > •
< (1) (2) (3) (4) (6) > •
< (1) (2) (3) (4 5) (6) > ∘
< (1) (2) (3 4) (6) > ·
< (1) (2) (3 4 5) (6) > ⊙

Les séquences marquées par • (respectivement ·) sont incluses dans celle marquée par ∘ (respectivement ⊙). Cela nous indique que seulement deux de ces séquences sont nécessaires car elles couvrent l'ensemble à tester :
< (1) (2) (3) (4 5) (6) >
< (1) (2) (3 4 5) (6) >

Pour prendre en compte la contrainte windowSize nous étendons l'algorithme GTC en calculant les combinaisons cohérentes de windowSize en début d'algorithme puis, une fois le graphe respectant minGap calculé, l'algorithme détecte alors les inclusions. Le résultat de cette manipulation est illustré par la figure 11 qui représente le graphe de séquence de la base représentée par la figure 10.
Construction du graphe de séquence
Pour énumerer les combinaisons de windowSize possibles, l'algorithme parcours chaque sommet x, et détermine pour chacun d'entre eux quels autres sommets peuvent être "fusionnés" avec x (x et y peuvent fusionnés si l'on peut grouper les itemsets qu'ils représentent, i.e. si $y.date() - x.date() < windowSize$). Nous adaptons la structure de données utilisée pour prendre en considération cette fusion. Désormais chaque sommet représente un itemset et possède un vecteur *isPrev* de taille $|S|$ indiquant pour ce sommet quel autre sommet le précède par un chemin quelconque et un itemset possède une date de début (date de transaction du premier item) et une date de fin (date de transaction du dernier item) qui seront accédés par $i.begin()$

Client	Date	Item
C1	01/04/99	1
C1	07/04/99	2
C1	13/04/99	3
C1	17/04/99	4
C1	18/04/99	5
C1	24/04/99	6

FIG. 10 – *Exemple de base de données*

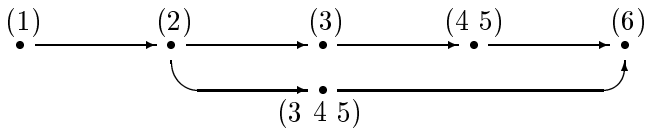


FIG. 11 – *Un graphe de séquence calculé avec windowSize*

et $i.end()$.

Définition 6 *Un itemset i est inclus dans un autre itemset j si et seulement si les deux conditions suivantes sont respectées :*

- $i.begin() \geq j.begin()$,
- $i.end() \leq j.end()$.

Une fois le graphe respectant minGap calculé (C.f. Algorithme GTC_1, Paragraphe 4.1), l'algorithme détecte les inclusions de la manière suivante: pour chaque sommet x, déterminer l'ensemble de ses successeurs : $x.next$. Pour chaque sommet y dans $x.next$, si $y \subset z, z \in x.next$ et $y.next \subseteq z.next$ alors retirer le sommet y du graphe.

FIG. 12 – *Graphe de séquence après la première et la seconde étape*

Exemple 8 *Considérons la base de données représentée par la figure 10, le graphe résultant de la première étape est représenté par la figure 12. En effet, windowSize étant fixé à 4, les items* 3, 4 *et* 5 *peuvent être fusionnés. Cependant, il ne faut pas négliger la possibilité que < (3) (4 5) > ou encore < (3 4) (5) > peuvent faire partie d'une séquence candidate et donc doivent être testées. C'est pourquoi l'algorithme construit les sommets qui représentent les itemsets (3 4), (4 5) et (3 4 5).*
le graphe résultant de la seconde étape est illustré par la figure 12. Il s'agit du graphe après l'exécution de GTC_1. L'adaptation pour prendre en compte les possibilités ajoutées par windowSize se fait donc de manière immédiate.

function GTC
Input : une sous-séquence de données d
Output : $G_d(S, A)$ le sequence-graph de d
foreach *itemset* $i \in d$ **do**
    $S = S \cup \{i\}$;
endforeach
$addWindowSize(S)$; //Ajoute à l'ensemble des sommets les combinaisons possibles avec windowSize
foreach $x \in S$ **do**
    //Phase de propagation
    $y = x$;
    while $y.begin() - x.begin() < minGap$ **do** $y++$;
    $A = A \cup \{x, y\}$;
    $i_p = \{i \in S / i.begin() - y.begin() > minGap\}$;
    foreach $z \in i_p$ **do**
        $z.isPrec[x] = 1$;
    endforeach
    // Phase de la main tendue
    $j_p = \{j \in S / j.begin() > y.begin()$ *and* $j.isPrec[x] = 0\}$;
    foreach $t \in j_p$ **do**
        $A = A \cup \{x, t\}$;
    endforeach
endforeach
$pruneIncluded(S, A)$; // Supprime du graphe les sommets faisant partie de séquences incluses.
end function GTC

Algorithme 2: *GTC version complète, solution pour minGap & windowSize*

La méthode utilisée pour détecter l'inclusion est illustrée figure 13. Nous constatons que les séquences $<$ *(3) (4) (6)* $>$ *et* $<$ *(3) (5) (6)* $>$ *sont incluses dans la séquence* $<$ *(3) (4 5) (6)* $>$. *En effet* $3.next = \{(4), (5), (4\ 5)\}$, $4.next = 5.next = (4\ 5).next = 6$ *et comme* $4 \subset (4\ 5)$ *et* $5 \subset (4\ 5)$ *les sommets 4 et 5 peuvent être supprimés. En revanche,* $2.next = \{(3), (3\ 4), (3\ 4\ 5)\}$ *mais* $3.next \not\subset (3\ 4\ 5).next$ *donc le sommet 3 n'est pas supprimé.*
Le graphe utilisé lors de la vérification des candidats est illustré par la figure 11 □

Le théorème 3 garantit que l'algorithme GTC, adapté à la contrainte de windowSize, résoud le problème des chemins respectant minGap et windowSize.

Théorème 3 *L'algorithme GTC (version finale) construit exactement toutes les solutions de la plus grande taille possible pour le problème des chemins respectant minGap et windowSize.*

Démonstration : Nous avons déjà vu (Théorème 1) que l'inclusion de chemins de séquence (au sens de la définition 2) ne peut pas exister dans un graphe de séquence issu de la première version de GTC. Nous allons donc nous intéresser au seul cas de windowSize. Supposons que le graphe de séquence calculé par GTC contienne deux chemins de séquence $s1$ et $s2$ tels que $s1 \subset s2$. Cela voudrait dire que le graphe de séquence contient le sous-graphe représenté par la figure 14. Or, la dernière phase de l'algorithme (C.f. Fonction $pruneIncluded(S, A)$) consiste pour chaque sommet x à déterminer l'ensemble de ses successeurs "$x.next$". pour chaque sommet y dans $x.next$, si $y \subset z, z \in x.next$ et $y.next \subseteq z.next$ le sommet y peut alors être retiré du graphe.

procedure *addWindowSize*
Input : l'ensemble des sommets S à traiter (trié par ordre croissant (1^{ere} clé : date de début, 2^{nde} clé : date de fin)).
$a = S.first()$;
while $a \neq S.last()$ **do**
  $b = S.succ(a)$;
  while $b.end() - a.begin() < windowSize$ **do**
    $i = group(a, b)$;
    //insère le sommet i avant le sommet b dans S
    $S.insert(i, b)$;
    $a = S.succ(a)$;
    $b = S.succ(a)$;
  endwhile
  $a = S.succ(a)$;
endwhile
end procedure *addWindowSize*

Algorithme 3: *Calcul des combinaisons possibles pour windowSize*

procedure *pruneIncluded*
Input : le graphe $G_d(S, A)$ dans lequel il faut traiter les cas d'inclusions dues à windowSize
foreach $x \in S$ **do**
  foreach $y \in x.next$ **do**
    foreach $z \in x.next$ **do**
      if $y \subset z$ *and* $y.next \subseteq z.next$ **then** $prune(y)$;
    endforeach
  endforeach
endforeach
end procedure *pruneIncluded*

Algorithme 4: *Détection et suppression des inclusions dues à windowSize*

Par construction, ce schéma ne peut pas apparaître dans le graphe issu de GTC.

5 Expérimentation

Pour analyser le comportement de l'algorithme en fonction de l'augmentation du nombre de transactions, nous avons réalisé des expériences sur une station de travail Enterprise 2 (processeur Ultra Sparc) avec une fréquence d'horloge CPU de 200MHz, 256MB de mémoire centrale et utilisant Unix. Les données sont stockées sur un disque de 9GB Ultra Wide SCSI 3.5".

Pour comparer les performances des deux algorithmes GTC et PSP mis en jeu, nous avons utilisé le générateur de données *synthetic data-gen*[2] proposé par [3]. Ce générateur est à l'heure actuelle considéré comme un benchmark typique pour l'évaluation des algorithmes de recherche de règles d'association ou de motifs séquentiels ([3, 6, 13, 16, 4, 15, 8]). La génération des données synthétiques est expliquée dans [3]. Les paramètres utilisés pour la génération de bases de

2. *http://www.almaden.ibm.com/cs/quest/syndata.html*

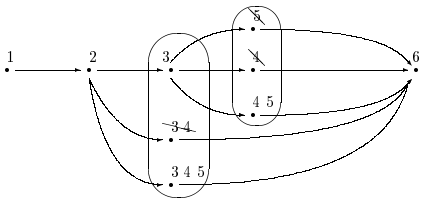


FIG. 13 – *Méthode de détection d'inclusions*

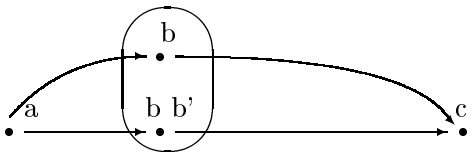


FIG. 14 – *Schéma d'inclusion minimum dû à windowSize*

données sont décrits dans la figure 15. En outre, pour compléter les tests, nous avons appliqué les algorithmes sur un fichier access log[3] de 85MB, 2000 clients et 1500 items (opérations effectuées sur le serveur).

$\|D\|$	Nombre de séquence de données
$\|C\|$	Nombre moyen de transactions par séquence de données
$\|S\|$	Longueur moyenne des séquences fréquentes potentielles
N	Nombre d'items

FIG. 15 – *Paramètres du générateur de données*

Les figures 17 et 18 illustrent les résultats obtenus sur les trois bases de données : l'axe des x correspond à une variation de windowSize et l'axe des y représente respectivement les temps de réponse et le nombre d'appels récursifs. Pour prendre en compte les contraintes temporelles, nous étudions les variations de windowSize avec un minGap=1 et maxGap=∞ (pas de contrainte de MaxGap).

Nous pouvons constater, après les series de tests effectués, que PSP teste trop de possibilités à partir d'une séquence de données et avec des contraintes de minGap et de windowSize non nulles. En effet, plus windowSize augmente, plus PSP effectue d'appels récursifs. Malgré un comportement linéaire par rapport à l'augmentation de windowSize et l'utilisation des contraintes temporelles, PSP affiche des temps de réponse moins performants que GTC sur les benchmarks testés. Un aspect intéressant des tests pratiqués réside dans la quantité d'appels récursifs relevée pour chaque algorithme. Nous pouvons en effet constater que, pour chaque courbe, une corrélation est observable entre les temps de réponse et le nombre d'appels récursifs effectués (une courbe représentant les temps de réponse a exactement la même allure que celle dénombrant les appels récursifs).

Il est également important de constater que, pour ces benchmarks, les fréquents sont volontairement de grande taille. Nous avons donc généré des instances de grand potentiel, et les complexités des algorithmes étant dépendantes de la taille de la réponse, nous avons pu constater que GTC, en intégrant des contraintes de temps, est une solution particulièrement efficace pour de telles instances.

3. Un fichier access log contient toutes les connexions à un serveur web (démon http).

Pour vérifier le comportement de GTC lors de l'augmentation du nombre de transactions, nous avons fait varier le nombre de séquences de données $|D|$ et nous avons remarqué que GTC affiche un comportement linéaire [4] avec des temps de réponse proportionnels à l'augmentation du nombre de séquences, quel que soit le support spécifié.

Nom	$\|D\|$	$\|C\|$	$\|S\|$	$\|N\|$
C20-D100-S10-N10	100K	20	10	10000
C20-D100-S8-N10	100K	20	4	10000

FIG. 16 – *Paramètres des fichiers de test*

Enfin, pour analyser complètement le comportement de GTC par rapport à PSP, nous avons effectué des tests en enlevant la contrainte de temps de windowSize et en faisant varier le support (C.f. Figure 19). Nous constatons, comme attendu, que le coût de l'algorithme dépend fortement de la valeur du support : quand le support diminue, le nombre total de séquences fréquentes augmente et nécessite donc des calculs supplémentaires. Les comparatifs sont donc très significatifs pour des supports faibles du fait de l'importance des temps des calculs impliqués.
En outre, nous constatons que GTC a un comportement moins efficace que PSP. Cela s'explique par le fait que GTC est obligé de créer le graphe avant de pouvoir rechercher directement les candidats d'une séquence de données.

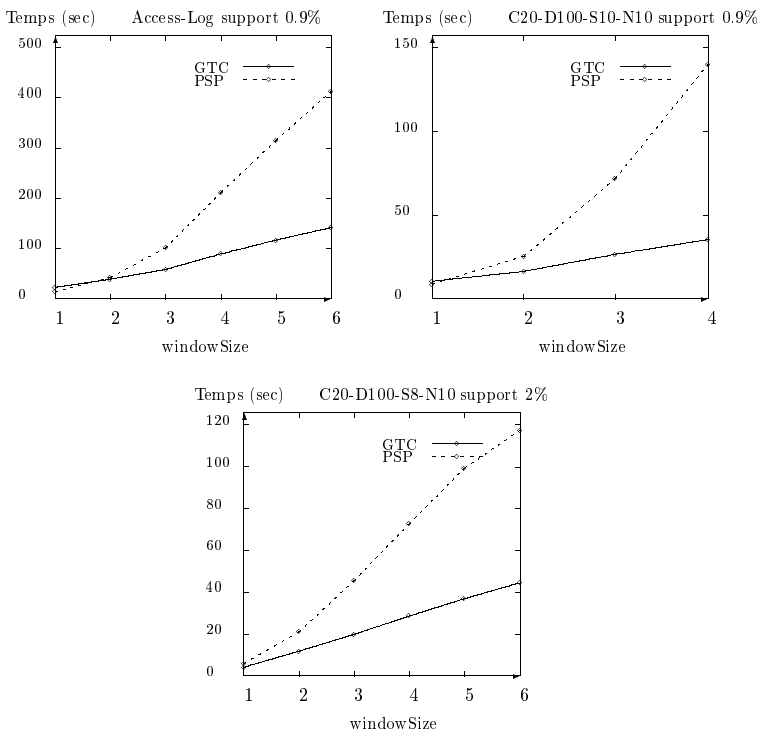


FIG. 17 – *Temps d'exécution comparés de PSP et GTC avec minGap*

4. Par manque de place, nous n'avons pas décrit les résultats mais le lecteur intéressé peut se reporter à [9].

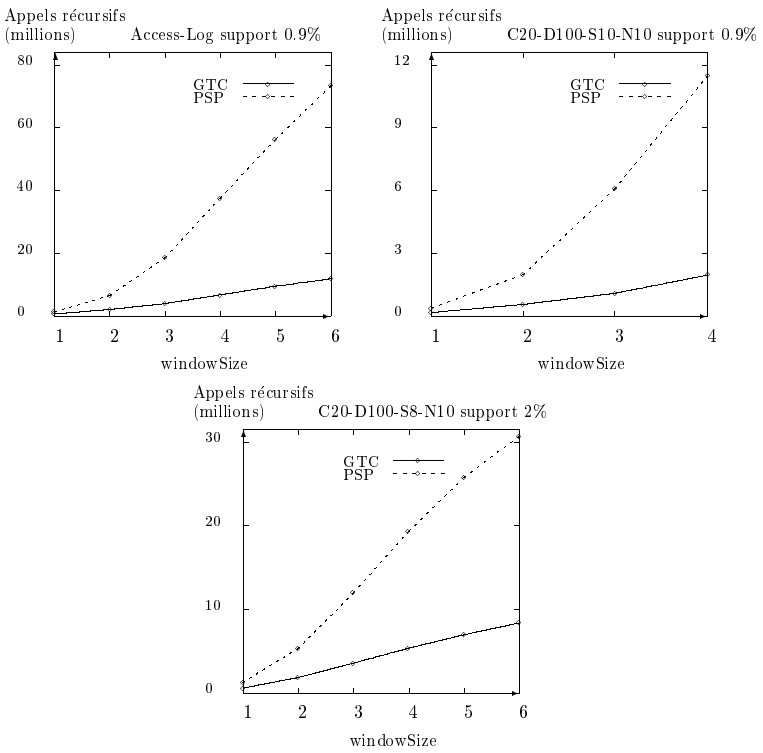


FIG. 18 – *Nombre d'appels récursifs comparés de PSP et GTC avec minGap*

L'algorithme de constrution du graphe semble être un investissement rentable quand la longueur des candidats à vérifier est grande. Il est, en effet, immédiat de constater que le temps de construction du graphe doit être inférieur au temps de parcours de PSP diminué du temps de parcours de GTC. Or, le temps nécessaire pour parcourir l'arbre des séquences candidates avec une séquence de données dans PSP croît de façon bien plus rapide que le temps nécessaire pour parcourir l'arbre des séquences candidates avec un graphe de séquence, alors que le temps de construction du graphe de séquence, lui, reste constant.

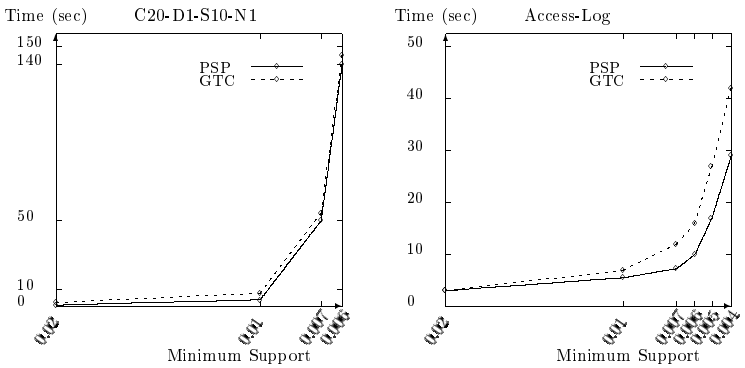


FIG. 19 – *Temps d'exécution comparés de PSP et GTC, windowSize=0*

Ces différences de temps requis sont, de plus, proportionnelles à l'importance des contraintes de temps spécifiées par l'utilisateur. Nous avons pourtant constaté que pour des contraintes de temps assez faible (windowSize=1 ou bien minGap=0) ou pour des fréquents relativement

courts, la seconde passe sur la base de données (destinée à tester les candidats de taille 2) restait encore plus efficace avec l'algorithme PSP. Malgré cela, GTC affiche des temps de réponse plus performants que PSP dans de tels cas de figure (grace à une plus grande rapidité d'exécution sur les passes suivantes). Pour exploiter ces comportements, nous avons créé un troisième algorithme (PG-*Hybrid*), hybride, utilisant la technique de PSP pour la seconde passe et ensuite GTC sur les passes suivantes. Comme le montrent les résultats illustrés par la figure 20, cette technique s'avère efficace pour des instances de faible potentiel (des fréquents courts), alors que pour des fréquents longs la différence de temps devient minime en proportion du temps total requis. En effet, nous constatons, figure 20, que pour des fréquents de taille 1, GTC peut se dérouler en 1,3 fois le temps requis par l'hybride. Ce facteur multiplicatif diminue lorsque la taille des fréquents augmente. Cela s'explique par la différence des deux algorithmes quant à la gestion de la première passe, cette dernière devenant insignifiante si le nombre de passes augmente (i.e. la taille des fréquents augmente).

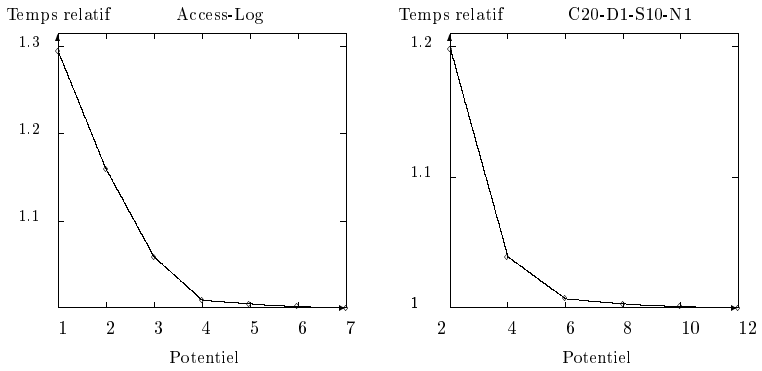


FIG. 20 – *Temps d'exécution relatif de GTC par rapport à hybrid, windowSize=1*

6 Conclusion

Dans cet article, nous avons présenté une approche efficace pour extraire des motifs séquentiels généralisés. L'originalité de cette approche est d'effectuer un prétraitement des séquences de données de la base en intégrant les contraintes temporelles pour ne rechercher que les séquences candidates maximales.
Nous avons également montré que dans le cadre de la recherche de motifs séquentiels, i.e. sans prise en compte des contraintes de temps, l'approche PSP [10] est dans l'ensemble plus rapide que GTC. Cette différence s'explique par le coût de la construction du graphe, qui est mis en œuvre de manière inutile quand aucune contrainte de temps n'est demandée (C.f. Figure 19).

Références

[1] R. Agrawal, T. Imielinski, and A. Swami. Mining Association Rules between Sets of Items in Large Databases. In *Proceedings of the 1993 ACM SIGMOD Conference*, pages 207–216, Washington DC, USA, May 1993.

[2] R. Agrawal and R. Srikant. Fast Algorithms for Mining Generalized Association Rules. In *Proceedings of the 20th International Conference on Very Large Databases (VLDB'94)*, Santiago, Chile, September 1994.

[3] R. Agrawal and R. Srikant. Mining Sequential Patterns. In *Proceedings of the 11th International Conference on Data Engineering (ICDE'95)*, Tapei, Taiwan, March 1995.

[4] S. Brin, R. Motwani, J.D. Ullman, and S. Tsur. Dynamic Itemset Counting and Implication Rules for Market Basket Data. In *Proceedings of the International Conference on Management of Data (SIGMOD'97)*, pages 255–264, Tucson, Arizona, May 1997.

[5] U.M. Fayad, G. Piatetsky-Shapiro, P. Smyth, and R. Uthurusamy, editors. *Advances in Knowledge Discovery and Data Mining*. AAAI Press, Menlo Park, CA, 1996.

[6] G. Gardarin, P. Pucheral, and F. Wu. Bitmap Based Algorithms For Mining Association Rules. In *Actes des journées Bases de Données Avancées (BDA'98)*, Hammamet, Tunisie, October 1998.

[7] H. Mannila, H. Toivonen, and A.I. Verkamo. Discovery of Frequent Episodes in Event Sequences. *Data Mining and Knowledge Discovery*, 1(3), September 1997.

[8] H. Mannila, H. Toivonen, and A. I. Verkano. Discovering Frequent Episodes in Sequences. In *Proceedings of the 1st International Conference on Knowledge Discovery in Databases and Data Mining (KDD'95)*, pages 210–215, Montreal, Canada, August 1995.

[9] F. Masseglia. Le pré-calcul appliqué à l'extraction de sequential patterns en data mining. Technical report, LIRMM, France, June 1998.

[10] F. Masseglia, F. Cathala, and P. Poncelet. The PSP Approach for Mining Sequential Patterns. In *Proceedings of the 2nd European Symposium on Principles of Data Mining and Knowledge Discovery (PKDD'98), LNAI, Vol. 1510*, pages 176–184, Nantes, France, September 1998.

[11] A. Mueller. Fast Sequential and Parallel Algorithms for Association Rules Mining: A Comparison. Technical Report CS-TR-3515, Department of Computer Science, University of Maryland-College Park, August 1995.

[12] N. Pasquier, Y. Bastide, R. Taouil, and L. Lakhal. Efficient Mining of Association Rules Using Closed Itemset Lattices. *Information Systems*, 19(4):33–54, 1998.

[13] A. Savasere, E. Omiecinski, and S. Navathe. An Efficient Algorithm for Mining Association Rules in Large Databases. In *Proceedings of the 21 st International Conference on Very Large Databases (VLDB'95)*, pages 432–444, Zurich, Switzerland, September 1995.

[14] R. Srikant. Fast Algorithms for Mining Association Rules and Sequential Patterns. Technical Report PHD Dissertation, University of Wisconsin - Madison, 1996.

[15] R. Srikant and R. Agrawal. Mining Sequential Patterns: Generalizations and Performance Improvements. In *Proceedings of the 5th International Conference on Extending Database Technology (EDBT'96)*, pages 3–17, Avignon, France, September 1996.

[16] H. Toivonen. Sampling Large Databases for Association Rules. In *Proceedings of the 22nd International Conference on Very Large Databases (VLDB'96)*, September 1996.

[17] M. Zaki. Scalable Data Mining for Rules. Technical Report PHD Dissertation, University of Rochester - New York, 1998.