ALGORITHMES POUR LA DÉCOUVERTE DE MOTIFS PROFITABLES

THÈSE PRÉSENTÉE À LA FACULTÉ DES ÉTUDES SUPÉRIEURES ET DE LA RECHERCHE EN VUE DE L'OBTENTION DE LA MAÎTRISE ÈS SCIENCES EN INFORMATIQUE

SOULEYMANE ZIDA

DÉPARTEMENT D'INFORMATIQUE
FACULTÉ DES SCIENCES
UNIVERSITÉ DE MONCTON

AOÛT 2015

COMPOSITION DU JURY

Président du jury :

Julien Chiasson

Professeur d'informatique, Université de Moncton

Examinateur externe :

Nicolas Béchet

Maître de conférences
Université de Bretagne-Sud, France

Examinateur interne :

Éric Hervet

Professeur d'informatique, Université de Moncton

Directeur de thèse :

Philippe Fournier-Viger

Professeur d'informatique, Université de Moncton

REMERCIEMENTS

Je tiens à remercier grandement mon directeur de thèse, Philippe Fournier-Viger pour toute son aide. Je suis ravi d'avoir travaillé avec Philippe Fournier-Viger, il a toujours été là pour me soutenir tout au long de la réalisation de cette thèse, le temps qu'il a bien voulu me consacrer et sans qui cette thèse n'aurait jamais vu le jour.

Mes remerciements s'adressent également aux membres du jury, pour avoir accepté d'évaluer mon travail.

Je remercie également les autres professeurs du Département d'informatique pour leur encouragement et leur enseignement.

Enfin, j'exprime ma profonde gratitude à mes parents en signe d'amour, de reconnaissance pour leur soutien et tous les sacrifices qu'ils ont faits pour ma réussite. Merci aussi à mes amis, et à tous mes proches qui ont participé à la réalisation de ce travail.

TABLE DES MATIÈRES

TABLE OF CONTENTS

LISTE DES TABLEAUX

LISTE DES FIGURES

RÉSUMÉ

La fouille de données est un domaine important de l'informatique portant sur la découverte d'informations nouvelles, inattendues et utiles dans des bases de données. Un problème important est la découverte de motifs fréquents, qui consiste à découvrir des itemsets (ensemble d'articles) achetés fréquemment dans des transactions de consommateurs. Les algorithmes classiques pour ce problème souffrent toutefois d'une limite importante : ils ne considèrent pas la quantité des articles (items) dans les transactions et leur profit unitaire. Par conséquent, ces algorithmes peuvent découvrir de nombreux motifs fréquents qui rapportent un faible profit et ignorer les motifs rares générant un profit élevé. Pour pallier cette limite, la découverte de motifs profitables (générant un haut profit) a émergé comme un problème important. Néanmoins, les algorithmes actuels souffrent de limites importantes. Tout d'abord, plusieurs de ces algorithmes ne considèrent pas l'ordonnancement des achats et la confiance, c'est-à-dire la probabilité qu'un client achète un produit sachant ses achats antérieurs, ce qui est désirable par exemple, pour la recommandation de produits. De plus, ces algorithmes sont généralement très demandant en termes de mémoire et temps d'exécution. Dans cette thèse, nous proposons deux algorithmes efficients pour la découverte de motifs profitables, afin de pallier ces limites. Le premier, HUSRM, est conçu pour être appliqué à des bases de données de séquences de transactions. Il vise la découverte de règles séquentielles de la forme $X \rightarrow Y$ indiquant que l'achat de certains articles X implique l'achat d'autres articles Y avec une certaine confiance, et rapporte un profit élevé. Des résultats expérimentaux sur plusieurs jeux de données montrent que HUSRM est 25 fois plus rapide et consomme 50% moins de mémoire qu'une version sans optimisations. Le second, EFIM, est appliqué à des bases de données de transactions sans ordre temporel. Il vise à réduire la consommation en mémoire et temps d'exécution pour la découverte des itemsets profitables. EFIM introduit plusieurs nouvelles stratégies pour élaguer l'espace de recherche et réduire la complexité du problème. Une étude expérimentale approfondie montre que EFIM est jusqu'à 7000 fois plus rapide et consomme moins de mémoire que plusieurs algorithmes de la littérature.

Mots clés : découverte de motifs profitables, règles séquentielles, itemsets.

Abstract

Data mining is an important research field, aiming at discovering novel, unexpected and useful patterns in databases. A fundamental task in data mining is the discovery of frequent patterns. It consists of discovering itemsets (groups of items) frequently purchased together by customers in a transaction database. Algorithms for this problem however suffer from a major limitation: they do not consider the unit profits of items and their quantities in transactions. Therefore, these algorithms may discover many frequent patterns that yield a low profit and ignore rare patterns that yield a high profit. To address this issue, the task of discovering high utility patterns (patterns generating a high profit) has emerged as a major research problem. However, current algorithms suffer from important limitations. First, many of these algorithms do not consider the purchase order of items and the confidence (the probability that a customer will buy an item given its previous purchases). These features are however desirable for applications such as product recommendation. Moreover, these algorithms are generally very demanding in terms of memory and execution time. In this thesis, we propose two efficient algorithms for the discovery of high-utility patterns to overcome these limitations.

The first algorithm, HUSRM, is designed to be applied to sequences of transactions (a sequence database). It discovers sequential rules of the form $X \rightarrow Y$ indicating that the purchase of some items X is followed by the purchase of some other items Y with a certain confidence, and yields a high profit. Experimental results on several datasets show that HUSRM is 25 times faster and consumes up to 50% less memory than a version without optimizations.

The second algorithm, EFIM, is applied to a transaction database that does not contain information about the temporal ordering of transactions. EFIM aims at reducing the memory consumption and execution time for discovering high-utility itemsets. EFIM introduces several new strategies to prune the search space and improve the performance of this task. A thorough experimental study shows that EFIM is up to 7000 times faster and consumes less memory than the state-of-the-art algorithms for this task.

Keywords: high-utility mining, pattern mining, sequential rules, itemsets.

INTRODUCTION GÉNÉRALE

La quantité de données numériques stockées dans des bases de données croît à une vitesse vertigineuse pour de nombreux domaines d'application. Un besoin important est de pouvoir analyser ces données pour en tirer des informations utiles, par exemple, pour la prise de décisions. Par exemple, une compagnie peut accumuler une grande quantité d'information sur les transactions réalisées par ses clients sans avoir les outils appropriés pour analyser ces données et en extraire des informations utiles pour le marketing.

Pour répondre à cette problématique, la fouille de données (en anglais, *data mining*) a émergé comme une discipline importante en informatique. La fouille de données est un domaine qui se trouve à l'intersection de plusieurs disciplines telles que la statistique, l'intelligence artificielle et l'apprentissage machine. Elle vise le développement d'algorithmes automatiques ou semi-automatiques pour l'extraction de connaissances nouvelles, inattendues et utiles à partir de bases de données [23–26]. Ces connaissances peuvent prendre diverses formes et peuvent être ensuite exploitées pour la prise de décisions ou bien simplement pour mieux comprendre les données. Le domaine de la fouille de données contient un certain nombre d'axes de recherche fondamentaux comme la classification, la catégorisation, la découverte d'anomalies et la découverte de motifs et d'associations [23–26].

Dans cette thèse, nous sommes intéressés par la découverte de motifs, et plus particulièrement par le problème de l'analyse du panier des consommateurs. Ce problème porte sur l'analyse des transactions effectuées par des clients dans un commerce. Dans ce contexte, une base de données transactionnelle est un ensemble de transactions où une transaction est l'achat d'un ensemble d'articles (*items*) par un client. Un problème classique est celui de découvrir les *itemsets fréquents* (DIF) dans une base de données transactionnelle, c'est-à-dire les ensembles d'articles fréquemment achetés par des clients [1]. Un ensemble d'articles est dit fréquent s'il est présent dans au moins $minsup$ transactions, où $minsup$ est un seuil choisi par l'utilisateur.

Le problème de la découverte d'itemsets fréquents est un problème difficile, car un algorithme

efficient doit éviter de calculer la fréquence de tous les ensembles d'articles. Par exemple, si 1,000 articles sont vendus dans un commerce, il y a en théorie $2^{1000} - 1$ ensembles d'articles à considérer. Pour résoudre le problème de la découverte des itemsets fréquents de façon efficiente, plusieurs algorithmes ont été proposés tels qu'Apriori [1], FPGrowth [6] et LCM [31].

Bien que ces algorithmes utilisent différentes procédures de recherche et différentes structures de données, ils emploient tous la propriété d'anti-monotonicité de la fréquence pour élaguer l'espace de recherche. Cette propriété indique que la fréquence d'un ensemble d'articles ne peut être qu'inférieure ou égale à celle de ses sous-ensembles. Cette propriété est très efficace pour élaguer l'espace de recherche ; si un ensemble n'est pas fréquent, il est inutile d'explorer tous ses surensembles.

Bien que la découverte d'itemsets fréquents a un très grand nombre d'applications dans une multitude de domaines, ce problème présente des limites importantes. Tout d'abord, la quantité de chaque article dans une transaction n'est pas prise en compte. Par exemple, l'achat d'un pain par un client est considéré de la même façon que l'achat de dix pains. Deuxièmement, il est supposé que tous les articles ont la même importance (poids/profit unitaire). Or, dans un commerce, tous les articles n'ont pas la même importance. Par exemple, la vente d'une bouteille de champagne et celle d'un pain ne devrait pas être considérée comme ayant la même importance du point de vue de la profitabilité. Troisièmement, la DIF vise à découvrir les ensembles d'articles fréquemment achetés. Toutefois, ce qui est fréquent n'est pas nécessairement intéressant. Par exemple, la DIF peut énumérer les ensembles d'articles fréquemment achetés ensembles tels que $\{pain, lait\}$ et ignorer des ensembles d'articles achetés moins souvent, mais qui rapportent tout de même une grande quantité d'argent comme $\{caviar, vin\}$.

Pour pallier ces limites, le problème de la découverte d'itemsets fréquent a été redéfini comme le problème de la découverte de motifs profitables (DMP) [2, 5, 11, 15, 18, 19, 27, 30]. Ce problème consiste à découvrir les ensembles d'articles qui sont profitables dans une base de données transactionnelle, c'est-à-dire les ensembles d'articles rapportant un profit supérieur à $minutil$ dollars, où $minutil$ est un paramètre spécifié par l'utilisateur. Dans le problème de DMP, les quantités des articles dans les transactions sont considérées ainsi que le profit unitaire de chaque article. Le

problème de la DMP est un sujet de recherche populaire actuellement, qui a plusieurs applications telles que l'analyse des achats de consommateurs et l'analyse de flux de clic sur un site Web [21,32].

Le problème de la découverte de motifs profitables est un problème reconnu comme étant plus difficile que le problème de la DIF. La raison est que la mesure du profit (appelée aussi *utility*) utilisée dans la DMP n'est pas anti-monotone, contrairement à la mesure de fréquence utilisée dans la DIF [2,5,11,18]. Par conséquent, les techniques développées pour élaguer l'espace de recherche pour la DIF ne peuvent pas être appliquées directement au problème de la DMP. Plusieurs algorithmes ont été proposés pour la découverte d'itemsets profitables [2,5,11,12,15,18,19,27,28,30]. Afin de pouvoir élaguer l'espace de recherche, ces algorithmes calculent des bornes supérieures sur le profit d'ensembles d'articles et leurs surensembles. Plusieurs algorithmes tels que Two-Phase [15], IHUP [2], UP-Growth [18], UP-Growth+ [19] et BAHUI [30] opèrent en deux phases. Dans une première phase, ils découvrent un ensemble de motifs candidats en surestimant leur profit. Puis dans la deuxième phase, ces algorithmes calculent le profit exact des candidats pour ne conserver que les motifs profitables. Cette approche est néanmoins très couteuse en terme de temps d'exécution et mémoire, car un grand nombre de candidats peut être généré. Pour pallier cette limite, des algorithmes dits à "une phase" ont été proposés tels que d^{2}HUP [28], HUP-Miner [12], HUI-Miner [11] et FHM [5]. Ces algorithmes sont nettement plus performants que les algorithmes à deux phases. Néanmoins, ces algorithmes restent très exigeants en temps et mémoire, et ont une complexité élevée. Un problème de recherche important est donc de développer des algorithmes plus efficients.

Par ailleurs, un autre problème important de la DMP est que la plupart des algorithmes développés pour la DMP ne tiennent pas compte de l'ordre entre les achats des consommateurs. Pour pallier ce problème, la découverte de motifs séquentiels profitables (DMSP) a été proposée récemment [21, 22]. Un algorithme de DMSP prend en entrée une base de données de séquences de transactions et un seuil $minutil$ fourni par l'utilisateur [21]. Chaque séquence contient les transactions effectuées par un consommateur, ordonnées selon le temps. Comme pour la DMP, les quantités des articles dans les transactions et les profits unitaires sont pris en compte. Un algorithme de DMSP retourne en sortie les sous-séquences d'articles rapportant plus que $minutil$ dollars. Bien que le problème de DMSP ait plusieurs applications [21, 22], une limite importante est que

les motifs découverts n'indiquent pas la probabilité qu'ils soient suivis. Par exemple, soit le motif $\langle\{lait\}, \{pain\}, \{champagne\}\rangle$. Ce motif sera découvert s'il génère un profit élevé. Toutefois, ce motif n'est pas utile pour prédire que les clients qui achètent du lait et du pain achèteront du champagne. La raison est que la probabilité que l'achat de lait et de pain soit suivi de l'achat de champagne est faible, car l'achat de lait et pain est fréquent, mais l'achat de champagne est rare. L'absence d'information sur la probabilité qu'un motif soit suivi entrave donc considérablement l'utilisation des motifs découverts dans des applications réelles telle que la recommandation de produits.

Dans le domaine de la découverte de motifs fréquents, une solution pour trouver des motifs indiquant une probabilité d'être suivi est la découverte de règles séquentielles [3]. Plusieurs algorithmes ont été développés à cette fin [3, 4, 16, 17]. Cependant, ces algorithmes ne tiennent pas compte de la quantité des articles dans les séquences et leur profit unitaire.

Objectif. L'objectif général de la thèse est de développer de nouveaux algorithmes efficients pour la découverte de motifs profitables. Plus précisément, deux sous-objectifs sont visés :

— Proposer un nouvel algorithme pour la DMP dans les bases de données transactionnelles, plus efficient en temps et mémoire que les algorithmes actuels.
— Adapter le problème de la DMSP pour offrir une information sur la probabilité qu'un motif soit suivi, en combinant la définition de motif séquentiel profitable et de règle séquentielle. Présenter un nouvel algorithme efficient pour découvrir le nouveau type de motifs proposé.

Hypothèse. Les principales hypothèses de ce travail sont :

— Un algorithme plus efficient pour la découverte de motifs profitables pourra être obtenu en concevant un algorithme à une phase introduisant de nouvelles bornes supérieures et de nouvelles optimisations, et en implémentant la plupart des opérations de l'algorithme en utilisant des opérations ayant une complexité linéaire en temps.
— S'inspirer des travaux sur les règles séquentielles et des travaux sur la découverte de motifs profitables en une seule phase, permettra de développer un algorithme efficient pour découvrir des règles séquentielles profitables.

Contribution. Les contributions principales de la thèse sont la proposition de deux nouveaux

algorithmes pour la découverte de motifs profitables.

— HUSRM [32], un algorithme de découverte de règle séquentielle qui associe une nouvelle structure de données nommée *compact utility-table* à chaque règle. Cette structure de données permet de calculer le profit, la fréquence, la confiance d'une règle et aussi d'élaguer l'espace de recherche. De plus, HUSRM introduit cinq optimisations permettant d'améliorer ses performances en temps d'exécution et consommation de mémoire.

— EFIM [33], un algorithme de découverte d'itemsets profitables qui inclut de nouvelles bornes supérieures pour élaguer efficacement l'espace de recherche et aussi une nouvelle structure de données pour calculer ces bornes supérieures de façon efficiente. De plus, pour réduire le coût de parcourir la base de données, EFIM effectue des projection de la base de données et fusionne les transactions identiques.

Organisation de la thèse. Cette thèse suit le format d'une thèse par articles. Les chapitres 1 et 2 contiennent deux publications acceptées pour publication dans les actes de conférences internationales sur la fouille de données. Chaque article présente un algorithme pour la découverte de motifs profitables.

— Zida, S., Fournier-Viger, P., Wu, C.-W., Lin, J. C. W., Tseng, V.S. (2015). Efficient Mining of High-Utility Sequential Rules Dans : *Procceding 11th International Conference on Machine Learning and Data Mining (MLDM 2015). Springer, LNAI, 15 pages.*

— Zida, S., Fournier-Viger, P., Wu, C.-W., Lin, J. C. W., Tseng, V.S. (2015). EFIM : Efficient High-Utility Itemset Mining. Dans : *Procceding 14th Mexican International Conference on Artificial Intelligence (MICAI 2015). Springer, LNAI, 20 pages.*

Il est à noter que ces chapitres ajoutent certains détails qui avaient été omis dans les versions publiées par manque d'espace. Finalement, la section 4 présente la conclusion générale de la thèse, incluant une discussion des limites du travail et des travaux futurs envisagés.

CHAPITRE I

EFFICIENT MINING OF HIGH-UTILITY SEQUENTIAL RULES

DÉCOUVERTE EFFICIENTE DE RÈGLES SÉQUENTIELLES PROFITABLES

Abstract

High-utility pattern mining is an important data mining task having wide applications. It consists of discovering patterns generating a high profit in databases. Recently, the task of high-utility sequential pattern mining has emerged to discover patterns generating a high profit in sequences of customer transactions. However, a well-known limitation of sequential patterns is that they do not provide a measure of the confidence or probability that they will be followed. This greatly hampers their usefulness for several real applications such as product recommendation. In this paper, we address this issue by extending the problem of sequential rule mining for utility mining. We propose a novel algorithm named HUSRM (High-Utility Sequential Rule Miner), which includes several optimizations to mine high-utility sequential rules efficiently. An extensive experimental study with four datasets shows that HUSRM is highly efficient and that its optimizations improve its execution time by up to 25 times and its memory usage by up to 50%.

Keywords: pattern mining, high-utility mining, sequential rules.

1 Introduction

Frequent Pattern Mining (FPM) is a fundamental task in data mining, which has many applications in a wide range of domains [1]. It consists of discovering groups of items appearing together frequently in a transaction database. However, an important limitation of FPM is that it assumes that items cannot appear more than once in each transaction and that all items have the same importance (e.g. weight, unit profit). These assumptions do not hold in many real-life applications. For example, consider a database of customer transactions containing information on quantity of purchased items and their unit profit. If FPM algorithms are applied on this database, they may discover many frequent patterns generating a low profit and fail to discover less frequent patterns that generate a high profit. To address this issue, the problem of FPM has been redefined as High-utility Pattern Mining (HUPM) [2, 5, 10, 15, 18]. However, these work do not consider the sequential ordering of items in transactions. High-Utility Sequential Pattern Mining (HUSP) was proposed to address this issue [21, 22]. It consists of discovering sequential patterns in sequences of customer transactions containing quantity and unit profit information. Although, this definition was shown to be useful,

an important drawback is that it does not provide a measure of confidence that patterns will be followed. For example, consider a pattern $\langle\{milk\}, \{bread\}, \{champagne\}\rangle$ meaning that customers bought milk, then bread, and then champagne. This pattern may generate a high profit but may not be useful to predict what customers having bought milk and bread will buy next because milk and bread are very frequent items and champagne is very rare. Thus, the probability or confidence that milk and bread is followed by champagne is very low. Not considering the confidence of patterns greatly hampers their usefulness for several real applications such as product recommendation.

In FPM, a popular alternative to sequential patterns that consider confidence is to mine sequential rules [3]. A sequential rule indicates that if some item(s) occur in a sequence, some other item(s) are likely to occur afterward with a given confidence or probability. Two main types of sequential rules have been proposed. The first type is rules where the antecedent and consequent are sequential patterns [16, 17]. The second type is rules between two unordered sets of items [3, 4]. In this paper we consider the second type because it is more general and it was shown to provide considerably higher prediction accuracy for sequence prediction in a previous study [4]. Moreover, another reason is that the second type has more applications. For example, it has been applied in e-learning, manufacturing simulation, quality control, embedded systems analysis, web page prefetching, anti-pattern detection, alarm sequence analysis and restaurant recommendation (see [4] for a survey). Several algorithms have been proposed for sequential rule mining. However, these algorithms do not consider the quantity of items in sequences and their unit profit. But this information is essential for applications such as product recommendation and market basket analysis. Proposing algorithms that mine sequential rules while considering profit and quantities is thus an important research problem.

However, addressing this problem raises major challenges. First, algorithms for utility mining cannot be easily adapted to sequential rule mining. The reason is that algorithms for HUPM and HUSP mining such as USpan [21], HUI-Miner [11] and FHM [5] use search procedures that are very different from the ones used in sequential rule mining [3, 4]. A distinctive characteristics of sequential rule mining is that items may be added at any time to the left or right side of rules to obtain larger rules and that confidence needs to be calculated. Second, the proposed algorithm should be efficient in both time and memory, and have an excellent scalability.

Table I-1 A Sequence Database

SID	Sequences
s_1	$\langle\{(a,1)(b,2)\}(c,2)(f,3)(g,2)(e,1)\rangle$
s_2	$\langle\{(a,1)(d,3)\}(c,4),(b,2),\{(e,1)(g,2)\}\rangle$
s_3	$\langle(a,1)(b,2)(f,3)(e,1)\rangle$
s_4	$\langle\{(a,3)(b,2)(c,1)\}\{(f,1)(g,1)\}\rangle$

Table I-2 External Utility Values

Item	a	b	c	d	e	f	g
Profit	1	2	5	4	1	3	1

In this paper, we address these challenges. Our contributions are fourfold. First, we formalize the problem of high-utility sequential rule mining and its properties. Second, we present an efficient algorithm named HUSRM (High-Utility Sequential Rule Miner) to solve this problem. Third, we propose several optimizations to improve the performance of HUSRM. Fourth, we conduct an extensive experimental study with four datasets. Results show that HUSRM is very efficient and that its optimizations improve its execution time by up to 25 times and its memory usage by up to 50%.

The rest of this paper is organized as follows. Section 2, 3, 4 and 5 respectively presents the problem definition and related work, the HUSRM algorithm, the experimental evaluation and the conclusion.

2 Problem Definition and Related Work

We consider the definition of a sequence database containing information about quantity and unit profit, as defined by Yin et al. [21].

Definition 1 (Sequence database). Let $I = \{i_1, i_2, ..., i_l\}$ be a set of items (symbols). An *itemset* $I_x = \{i_1, i_2, ..., i_m\} \subseteq I$ is an unordered set of distinct items. The *lexicographical order* $\succ_{lex}$ is defined as any total order on I. Without loss of generality, it is assumed in the following that all itemsets are ordered according to $\succ_{lex}$. A *sequence* is an ordered list of itemsets $s = \langle I_1, I_2, ..., I_n \rangle$ such that $I_k \subseteq I$ $(1 \leq k \leq n)$. A *sequence database* SDB is a list of sequences $SDB = \langle s_1, s_2, ..., s_p \rangle$ having sequence identifiers (SIDs) $1, 2...p$. Note that it is assumed that sequences cannot contain the same item more than once. Each item $i \in I$ is associated with a positive number

$p(i)$, called its *external utility* (e.g. unit profit). Every item i in a sequence s_c has a positive number $q(i, s_c)$, called its *internal utility* (e.g. purchase quantity).

Example 1. Consider the sequence database shown in Table. I-1, which will be the running example. It contains four sequences having the SIDs 1, 2, 3 and 4. Each single letter represents an item and is associated with an integer value representing its internal utility. Items between curly brackets represent an itemset. When an itemset contains a single item, curly brackets are omitted for brevity. For example, the first sequence s_1 contains five itemsets. It indicates that items a and b occurred at the same time, were followed by c, then f, then g and lastly e. The internal utility (quantity) of a, b, c, e, f and g in that sequence are respectively 1, 2, 2, 3, 2 and 1. The external utility (unit profit) of each item is shown in Table I-2.

The problem of sequential rule mining is defined as follows [3, 4].

Definition 2 (Sequential rule). A sequential rule $X \rightarrow Y$ is a relationship between two unordered itemsets $X, Y \subseteq I$ such that $X \cap Y = \emptyset$ and $X, Y \neq \emptyset$. The interpretation of a rule $X \rightarrow Y$ is that if items of X occur in a sequence, items of Y will occur afterward in the same sequence.

Definition 3 (Itemset/rule occurrence). Let $s = \langle I_1, I_2 ... I_n \rangle$ be a sequence. An itemset I occurs or is contained in s (written as $I \sqsubseteq s$) iff $I \subseteq \bigcup_{i=1}^{n} I_i$. A rule $r = X \rightarrow Y$ occurs or is contained in s (written as $r \sqsubseteq s$) iff there exists an integer k such that $1 \leq k < n$, $X \subseteq \bigcup_{i=1}^{k} I_i$ and $Y \subseteq \bigcup_{i=k+1}^{n} I_i$. Furthermore, let $seq(r)$ and $ant(r)$ respectively denotes the set of sequences containing r and the set of sequences containing its antecedent, i.e. $seq(r) = \{s | s \in SDB \wedge r \sqsubseteq s\}$ and $ant(r) = \{s | s \in SDB \wedge X \sqsubseteq s\}$.

Example 2. Consider the database of Table. I-1. According to the above definition of rule occurrence, the only requirement for a rule to appear in a sequence is that items from the rule antecedent appear before those of the rule consequent. But there is no restriction on the order between items from the rule antecedent or consequent, and items from the antecedent or consequent may not appear in the same itemset in the sequence. Thus, the rule $\{a, b, c\} \rightarrow \{e\}$ appears in sequence $s_1 = \langle \{(a, 1)(b, 2)\}(c, 2)(f, 3)(g, 2)(e, 1) \rangle$, and sequence $s_2 = \langle \{(a, 1)(d, 3)\}(c, 4), (b, 2), \{(e, 1)(g, 2)\} \rangle$.

Definition 4 (Support). The *support* of a rule $r = X \rightarrow Y$ in a sequence database SDB is defined as $sup_{SDB}(r) = |seq(r)|/|SDB|$.

Example 3. The rule $\{a, b, c\} \rightarrow \{e\}$, $\{a, b, c\} \rightarrow \{g\}$ and $\{a, d\} \rightarrow \{b, c, e, g\}$ respectively have a support of 0.50, 0.75 and 1.0.

Definition 5 (Confidence). The *confidence* of a rule $r = X \rightarrow Y$ in a sequence database SDB is defined as $conf_{SDB}(r) = |seq(r)|/|ant(r)|$.

Example 4. The rule $\{a, b, c\} \rightarrow \{e\}$, $\{a, b, c\} \rightarrow \{g\}$ and $\{a, d\} \rightarrow \{b, c, e, g\}$ respectively have a confidence of 0.66, 1.0 and 1.0.

Definition 6 (Sequential rule size). A rule $X \rightarrow Y$ is said to be of size $k * m$ if $|X| = k$ and $|Y| = m$. Note that the notation $k * m$ is not a product. It simply means that the sizes of the left and right parts of a rule are respectively k and m. Furthermore, a rule of size $f * g$ is said to be larger than another rule of size $h * i$ if $f > h$ and $g \geq i$, or alternatively if $f \geq h$ and $g > i$. Otherwise, the two rules are said to be incomparable.

Example 5. The rules $r = \{a, b, c\} \rightarrow \{e, f, g\}$ and $s = \{a\} \rightarrow \{e, f\}$ are respectively of size $3 * 3$ and $1 * 2$. Thus, r is larger than s.

Definition 7 (Problem of sequential rule mining). Let $minsup$, $minconf \in [0, 1]$ be thresholds set by the user and SDB be a sequence database. A sequential rule r is *frequent* iff $sup_{SDB}(r) \geq minsup$. A sequential rule r is *valid* iff it is frequent and $conf_{SDB}(r) \geq minconf$. The *problem of mining sequential rules* from a sequence database is to discover all valid sequential rules [3].

We adapt the problem of sequential rule mining to consider the utility (e.g. generated profit) of rules as follows.

Definition 8 (Utility of an item). The utility of an item i in a sequence s_c is denoted as $u(i, s_c)$ and defined as $u(i, s_c) = p(i) \times q(i, s_c)$.

Definition 9 (Utility of a sequential rule). Let be a sequential rule $r : X \rightarrow Y$. The utility of r in a sequence s_c is defined as $u(r, s_c) = \sum_{i \in X \cup Y} u(i, s_c)$ iff $r \sqsubseteq s_c$. Otherwise, it is 0. The utility of r in a sequence database SDB is defined as $u_{SDB}(r) = \sum_{s \in SDB} u(r, s)$ and is abbreviated as $u(r)$ when the context is clear.

Example 6. The itemset $\{a, b, f\}$ is contained in sequence s_1. The rule $\{a, c\} \rightarrow \{e, f\}$ occurs in s_1, whereas the rule $\{a, f\} \rightarrow \{c\}$ does not, because item c does not occur after f. The profit of item c in sequence s_1 is $u(c, s_1) = p(c) \times q(c, s_1) = 5 \times 2 = 10$. Consider a rule $r = \{c, f\} \rightarrow \{g\}$. The profit of r in s_1 is $u(r, s_1) = u(c, s_1) + u(f, s_1) + u(g, s_1) = (5 \times 2) + (3 \times 3) + (1 \times 2) = 18$. The profit of r in the database is $u(r) = u(r, s_1) + u(r, s_2) + u(r, s_3) + u(r, s_4) = 18 + 0 + 0 + 9 = 27$.

Definition 10 (Problem of high-utility sequential rule mining). Let $minsup, minconf \in [0, 1]$ and $minutil \in \mathbf{R}^+$ be thresholds set by the user and SDB be a sequence database. A rule r is a *high-utility sequential rule* iff $u_{SDB}(r) \geq minutil$ and r is a valid rule. Otherwise, it is said to be a low utility sequential rule. The *problem of mining high-utility sequential rules* from a sequence database is to discover all high-utility sequential rules.

Example 7. Table I-3 shows the sequential rules found for $minutil = 40$ and $minconf = 0.65$. In this example, we can see that rules having a high-utility and confidence but a low support can be found (e.g. r_7). These rules may not be found with regular sequential rule mining algorithms because they are rare, although they are important because they generate a high profit.

Algorithms for mining sequential rules explore the search space of rules by first finding frequent rules of size $1 * 1$. Then, they recursively append items to either the left or right sides of rules to find larger rules [3]. The *left expansion* of a rule $X \rightarrow Y$ with an item $i \in I$ is defined as $X \cup \{i\} \rightarrow Y$, where $i \succ_{lex} j, \forall j \in X$ and $i \notin Y$. The *right expansion* of a rule $X \rightarrow Y$ with an item $i \in I$ is defined as $X \rightarrow Y \cup \{i\}$, where $i \succ_{lex} j, \forall j \in Y$ and $i \notin X$. Sequential rule mining algorithms prune the search space using the support because it is anti-monotonic. However, this is not the case for the utility measure, as we show thereafter.

Property 1 (antimonotonicity of utility). Let be a rule r and a rule s, which is a left expansion of r. It follows that $u(r) < u(s)$ or $u(r) \geq u(s)$. Similarly, for a rule t, which is a right expansion of r,

Table I-3 Sequential rules found for $minutil = 40$ and $minconf = 0.65$

ID	Sequential rule	Support	Confidence	Utility
r_1	$\{a, b, c\} \rightarrow \{e\}$	0.50	0.66	42
r_2	$\{a, b, c\} \rightarrow \{e, g\}$	0.50	0.66	46
r_3	$\{a, b, c\} \rightarrow \{f, g\}$	0.50	0.66	42
r_4	$\{a, b, c\} \rightarrow \{g\}$	0.75	1.0	57
r_5	$\{a, b, c, d\} \rightarrow \{e, g\}$	0.25	1.0	40
r_6	$\{a, c\} \rightarrow \{g\}$	0.75	1.0	45
r_7	$\{a, c, d\} \rightarrow \{b, e, g\}$	0.25	1.0	40
r_8	$\{a, d\} \rightarrow \{b, c, e, g\}$	1.0	1.0	40
r_9	$\{b, c\} \rightarrow \{e\}$	0.50	0.66	40
r_{10}	$\{b, c\} \rightarrow \{e, g\}$	0.50	0.66	44
r_{11}	$\{b, c\} \rightarrow \{g\}$	0.75	1.0	52
r_{12}	$\{c\} \rightarrow \{g\}$	0.75	1.0	40

$u(r) < u(t)$ or $u(r) \geq u(t)$.

Example 8. The rule $\{a\} \rightarrow \{b\}$, $\{a\} \rightarrow \{b, g\}$ and $\{a\} \rightarrow \{b, e\}$ respectively have a utility of 10, 7 and 12.

In high-utility pattern mining, to circumvent the problem that utility is not anti-monotonic, the solution has been to use anti-monotonic upper-bounds on the utility of patterns to be able to prune the search space. Algorithms such as Two-Phase [15], IHUP [2] and UP-Growth [18] discover patterns in two phases. During the first phase, an upper-bound on the utility of patterns is calculated to prune the search space. Then, during the second phase, the exact utility of remaining patterns is calculated by scanning the database and only high-utility patterns are output. However, an important drawback of this method is that too many candidates may be generated and may need to be maintained in memory during the first phase, which degrades the performance of the algorithms. To address this issue, one-phase algorithms have been recently proposed such as FHM [5], HUI-Miner [11] and USpan [21] to mine high-utility patterns without maintaining candidates.

These algorithms introduces the concept of *remaining utility*. For a given pattern, the remaining utility is the sum of the utility of items that can be appended to the pattern. The main upper-bound used by these algorithms to prune the search space is the sum of the utility of a pattern and its remaining utility. Since one-phase algorithms were shown to largely outperform two-phase algorithms, our goal is to propose a one-phase algorithm to mine high-utility sequential rules.

3 The HUSRM Algorithm

In the next subsections, we first present important definitions and data structures used in our proposal, the HUSRM algorithm. Then, we present the algorithm. Finally, we describe additional optimizations.

3.1 Definitions and Data Structures

To prune the search space of sequential rules, the HUSRM algorithm adapts the concept of sequence estimated utility introduced in high-utility sequential pattern mining [21] as follows.

Definition 11 (Sequence utility). The *sequence utility* (SU) of a sequence s_c is the sum of the utility of items from s_c in s_c. i.e. $SU(s_c) = \sum_{\{x\} \sqsubseteq s_c} u(x, s_c)$.

Example 9. The sequence utility of sequences s_1, s_2, s_3 and s_4 are respectively 27, 40, 15 and 16.

Definition 12 (Sequence estimated utility of an item). The *sequence estimated utility* (SEU) of an item x is defined as the sum of the sequence utility of sequences containing x, i.e. $SEU(x) = \sum_{s_c \in SDB \wedge \{x\} \sqsubseteq s_c} SU(s_c)$.

Definition 13 (Sequence estimated utility of a rule). The *sequence estimated utility* (SEU) of a sequential rule r is defined as the sum of the sequence utility of sequences containing r, i.e. $SEU(r) = \sum_{s_c \in seq(r)} SU(s_c)$.

Example 10. The SEU of rule $\{a\} \rightarrow \{b\}$ is $SU(s_1) + SU(s_2) + SU(s_3) = 27 + 40 + 15 = 82$.

Definition 14 (Promising item). An item x is *promising* iff $SEU(x) \geq minutil$. Otherwise, it is *unpromising*.

Definition 15 (Promising rule). A rule r is *promising* iff $SEU(r) \geq minutil$. Otherwise, it is *unpromising*.

The SEU measure has three important properties that are used to prune the search space.

Property 2 (Overestimation). The sequence estimated utility (SEU) of an item/rule w is higher or equal to its utility, i.e. $SEU(w) \geq u(w)$.

Property 3 (Pruning unpromising items). Let x be an item. If x is unpromising, then x cannot be part of a high-utility sequential rule.

Property 4 (Pruning unpromising rules). Let r be a sequential rule. If r is unpromising, then any rule obtained by transitive expansion(s) of r is a low utility sequential rule.

We also introduce a new structure called *utility-table* that is used by HUSRM to quickly calculate the utility of rules and prune the search space. Utility-tables are defined as follows.

Definition 16 (Extendability). Let be a sequential rule r and a sequence s. An item i *can extend* r *by left expansion* in s iff $i \succ_{lex} j, \forall j \in X, i \notin Y$ and $X \cup \{i\} \rightarrow Y$ occurs in s. An item i *can extend* r *by right expansion* in s iff $i \succ_{lex} j, \forall j \in Y, i \notin X$ and $X \rightarrow Y \cup \{i\}$ occurs in s. Let $onlyLeft(r, s)$ denotes the set of items that can extend r by left expansion in s but not by right expansion. Let $onlyRight(r, s)$ denotes the set of items that can extend r by right expansion in s but not by left expansion. Let $leftRight(r, s)$ denotes the set of items that can extend r by left and right expansion in s.

Definition 17 (Utility-table). The *utility-table* of a rule r in a database SDB is denoted as $ut(r)$, and defined as a set of tuples such that there is a tuple $(sid, iutil, lutil, rutil, lrutil)$ for each sequence s_{sid} containing r (i.e. $\forall s_{sid} \in seq(r)$). The $iutil$ element of a tuple is the utility of r in

s_{sid}. i.e., $u(r, s_{sid})$. The $lutil$ element of a tuple is defined as $\sum u(i, s_{sid})$ for all item i such that i can extend r by left expansion in s_{sid} but not by right expansion, i.e. $\forall i \in onlyLeft(r, s_{sid})$. The $rutil$ element of a tuple is defined as $\sum u(i, s_{sid})$ for all item i such that i can extend r by right expansion in s_{sid} but not by left expansion, i.e. $\forall i \in onlyRight(r, s_{sid})$. The $lrutil$ element of a tuple is defined as $\sum u(i, s_{sid})$ for all item i such that i can extend r by left or right expansion in s_{sid}, i.e. $\forall i \in leftRight(r, s_{sid})$.

Example 11. The utility-table of the rule $\{a\} \rightarrow \{b\}$ is $\{(s_1, 5, 12, 3, 20), (s_2, 5, 0, 10, 0)\}$. The utility-table of $\{a\} \rightarrow \{b, c\}$ is $\{(s_1, 25, 12, 3, 0)\}$. The utility-table of the rule $\{a, c\} \rightarrow \{b\}$ is $\{(s_1, 25, 12, 3, 0)\}$.

The proposed *utility-table* structure has the following nice properties to calculate the utility and support of rules, and for pruning the search space.

Property 5. Let be a sequential rule r. The utility $u(r)$ is equal to the sum of $iutil$ values in $ut(r)$.

Property 6. Let be a sequential rule r. The support of r in a database SDB is equal to the number of tuples in the utility-table of r, divided by the number of sequences in the database, i.e. $sup_{SDB}(r) = |ut(r)|/|SDB|$.

Property 7. Let be a sequential rule r. The sum of $iutil$, $lutil$, $rutil$ and $lrutil$ values in $ut(r)$ is an upper bound on $u(r)$. Moreover, it can be shown that this upper bound is tighter than $SEU(r)$.

Property 8. Let be a sequential rule r. The utility of any rule t obtained by transitive left or right expansion(s) of r can only have a utility lower or equal to the sum of $iutil$, $lutil$, $rutil$ and $lrutil$ values in $ut(r)$.

Property 9. Let be a sequential rule r. The utility of any rule t obtained by transitive left expansion(s) of r can only have a utility lower or equal to the sum of $iutil$, $lutil$ and $lrutil$ values in $ut(r)$.

Now, an important question is how to construct utility-tables. Two cases need to be considered. For sequential rules of size $1 * 1$, utility-tables can be built by scanning the database once. For sequential rules larger than size $1 * 1$, it would be however inefficient to scan the whole database for building a utility-table. To efficiently build a utility-table for a rule larger than size $1 * 1$, we propose the following scheme.

Consider the left or right expansion of a rule with an item i. The utility-table of the resulting rule r' is built as follows. Tuples in the utility-table of r are retrieved one by one. For a tuple $(sid, iutil, lutil, rutil, lrutil)$, if the rule r' appears in sequence s_{sid} (i.e. $r \sqsubseteq s_{sid}$), a tuple $(sid, iutil', lutil', rutil', lrutil')$ is created in the utility-table of r'. The value $iutil'$ is calculated as $iutil + u(\{i\}, s_{sid})$. $lutil'$ is calculated as $lutil - u(j, s_{sid}) \forall j \notin onlyLeft(r', s_{sid}) \wedge j \in onlyLeft(r, s_{sid}) - [u(i, s_{sid})$ if $i \in onlyLeft(r, s_{sid})]$. The value $rutil'$ is calculated as $rutil - u(j, s_{sid}) \forall j \notin onlyRight(r', s_{sid}) \wedge j \in onlyRight(r, s_{sid}) - [u(i, s_{sid})$ if $i \in onlyRight(r, s_{sid})]$. Finally, the value $lrutil'$ is calculated as follow: $lrutil - u(j, s_{sid}) \; \forall j \notin leftRight(r', s_{sid}) \wedge j \in leftRight(r, s_{sid}) - [u(i, s_{sid})$ if $i \in leftRight(r, s_{sid})]$. This procedure for building utility-tables is very efficient since it requires to scan each sequence containing the rule r at most once to build the utility-table of r' rather than scanning the whole database.

Example 12. The utility-table of the rule $r : \{a\} \rightarrow \{e\}$ is $\{(s_1, 2, 14, 0, 11), (s_2, 2, 36, 2, 0), (s_3, 2, 4, 0, 9)\}$. By adding the $iutil$ values of this table, we find that $u(r) = 6$ (Property 5). Moreover, by counting the number of tuples in the utility-table and dividing it by the number of sequences in the database, we find that $sup_{SDB}(r) = 0.75$ (Property 6). We can observe that the sum of $iutil$, $lutil$, $rutil$ and $lrutil$ values is equal to 82, which is an upper bound on $u(r)$ (Property 7). Furthermore, this value tells us that transitive left/right expansions of r may generate high-utility sequential rules (Property 8). And more particularly, because the sum of $iutil$, $lutil$ and $lrutil$ values is equal to 80, transitive left expansions of r may generate high-utility sequential rules (Property 9). Now, consider the rule $r' : \{a, b\} \rightarrow \{e\}$. The utility-table of r' can be obtained from the utility-table of r using the aforementioned procedure. The result is $\{(s_1, 6, 10, 0, 11), (s_2, 6, 32, 2, 0), (s_3, 6, 0, 0, 9)\}$. This table, can then be used to calculate utility-tables of other rules such as $\{a, b, c\} \rightarrow \{e\}$, which is $\{(s_1, 16, 0, 0, 11), (s_2, 26, 12, 2, 0)\}$.

Up until now, we have explained how the proposed utility-table structure is built, can be used to calculate the utility and support of rules and can be used to prune the search space. But a problem remains. How can we calculate the confidence of a rule $r : X \rightarrow Y$? To calculate the confidence, we need to know $|seq(r)|$ and $|ant(r)|$, that is the number of sequences containing r and the number of sequences containing its antecedent X. $|seq(r)|$ can be easily obtained by counting $|ut(r)|$. However, $|ant(r)|$ is more difficult to calculate. A naive solution would be to scan the database to calculate $|ant(r)|$. But this would be highly inefficient. In HUSRM, we calculate $|ant(r)|$ efficiently as follows. HUSRM first creates a bit vector for each single item appearing in the database. The *bit vector* $bv(i)$ of an item i contains $|SDB|$ bits, where the j-th bit is set to 1 if $\{i\} \sqsubseteq s_j$ and is otherwise set to 0. For example, $bv(a) = 1111$, $bv(b) = 1011$ and $bv(c) = 1101$. Now to calculate the confidence of a rule r, HUSRM intersects the bit vectors of all items in the rule antecedent, i.e. $\bigwedge_{i \in X} bv(i)$. The resulting bit vector is denoted as $bv(X)$. The number of bits set to 1 in $bv(X)$ is equal to $|ant(r)|$. By dividing the number of lines in the utility-table of the rule $|ut(r)|$ by this number, we obtain the confidence. This method is very efficient because intersecting bit vectors is a very fast operation and bit vectors does not consume much memory. Furthermore, an additional optimization is to reuse the bit vector $bv(X)$ of rule r to more quickly calculate $bv(X \cup \{i\})$ for any left expansions of r with an item i (because $bv(X \cup \{i\}) = bv(X) \wedge bv(\{i\})$).

3.2 The Proposed Algorithm

HUSRM explores the search space of sequential rules using a depth-first search. HUSRM first scans the database to build all sequential rules of size $1 * 1$. Then, it recursively performs left/right expansions starting from those sequential rules to generate larger sequential rules. To ensure that no rule is generated twice, the following ideas have been used.

First, an important observation is that a rule can be obtained by different combinations of left and right expansions. For example, consider the rule $r : \{a, b\} \rightarrow \{c, d\}$. By performing a left and then a right expansion of $\{a\} \rightarrow \{c\}$, one can obtain r. But this rule can also be obtained by performing a right and then a left expansion of $\{a\} \rightarrow \{c\}$. A simple solution to avoid this problem is to not allow performing a left expansion after a right expansion but to allow performing a right expansion after a left expansion. Note that an alternative solution is to not allow performing a left

expansion after a right expansion but to allow performing a right expansion after a left expansion.

Second, another key observation is that a same rule may be obtained by performing left/right expansions with different items. For example, consider the rule $r_9 : \{b, c\} \rightarrow \{e\}$. A left expansion of $\{b\} \rightarrow \{e\}$ with item c results in r_9. But r_9 can also be found by performing a left expansion of $\{c\} \rightarrow \{e\}$ with item b. To solve this problem, we chose to only add an item to a rule by left (right) expansion if the item is greater than each item in the antecedent (consequent) according to the total order $\succ_{lex}$ on items. By using this strategy and the previous one, no rules is considered twice. This solution is illustrated in fig. I-1

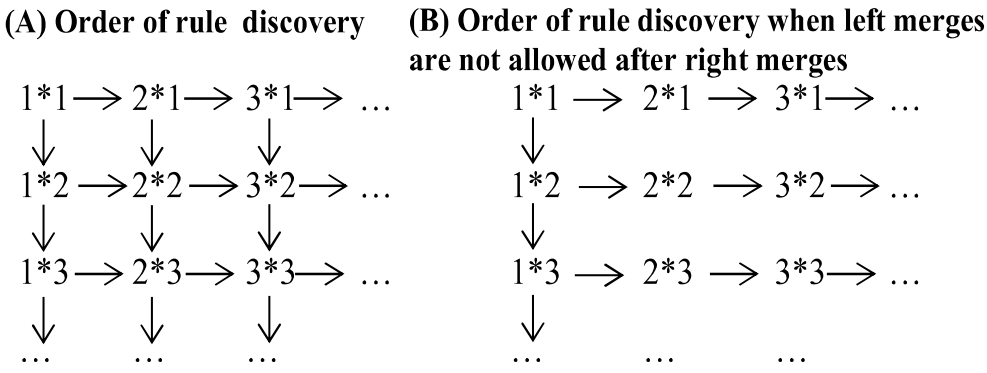

Figure I-1 The order of rule discovery by left/right merge operations

Fig. 1 shows the pseudocode of HUSRM. The HUSRM algorithm takes as parameters a sequence database SDB, and the $minutil$ and $minconf$ thresholds. It outputs the set of high-utility sequential rules. HUSRM first scans the database once to calculate the sequence estimated utility of each item and identify those that are promising. Then, HUSRM removes unpromising items from the database since they cannot be part of a high-utility sequential rule (Property 3). Thereafter, HUSRM only considers promising items. It scans the database to create the bit vectors of those items, and calculate $seq(r)$ and $SEU(r)$ for each rule r of size $1 * 1$ appearing in the database. Then, for each promising rule r, HUSRM scans the sequences containing r to build its utility-table $ut(r)$. If r is a high-utility sequential rule according to its utility-table and the bit-vector of its antecedent, the rule is output. Then, Property 8 and 9 are checked using the utility-table to determine if left and right expansions of r should be considered. Exploring left and right expansions is done by calling the *leftExpansion* and *rightExpansion* procedures.

The *leftExpansion* procedure (Algorithm 2) takes as input a sequential rule r and the other parameters of HUSRM. It first scans sequences containing the rule r to build the utility-table of each rule t that is a left-expansion of r. Note that the utility-table of r is used to create the utility-table of t as explained in Section 3.1. Then, for each such rule t, if t is a high-utility sequential rule according to its utility-table and the bit-vector of its antecedent, the rule is output. Finally, the procedure *leftExpansion* is called to explore left-expansions of t if Property 9 is verified. The *rightExpansion* procedure (Algorithm 3) is very similar to *leftExpansion* and is thus not described in details here. The main difference is that *rightExpansion* considers right expansions instead of left expansions and can call both *leftExpansion* and *rightExpansion* to search for larger rules.

Algorithm 1: The HUSRM algorithm

input : SDB: a sequence database, $minutil$ and $minconf$: the two user-specified thresholds

output: the set of high-utility sequential rules

1 Scan SDB to calculate the sequence estimated utility of each item $i \in I$;
2 $I^* \leftarrow \{i | i \in I \wedge SEU(i) \geq minutil\}$;
3 Remove from SDB each item $j \in I$ such that $j \notin I^*$;
4 Scan SDB to calculate the bit vector of each item $i \in I^*$;
5 Scan SDB to calculate R, the set of rules of the form $r : i \rightarrow j (i, j \in I^*)$ appearing in SDB and calculate $SEU(r)$ and $seq(r)$;
6 $R^* \leftarrow \{r | r \in R \wedge SEU(r) \geq minutil\}$;
7 **foreach** *rule* $r \in R^*$ **do**
8 Calculate $ut(r)$ by scanning $seq(r)$;
9 **if** $u(r) \geq minutil$ according to $ut(r)$ and $conf_{SDB}(r) \geq minconf$ **then** output r;
10 **if** r respects Property 8 according to $ut(r)$ **then** `rightExpansion` (r, SDB, $minutil$, $minconf$);
11 **if** r respects Property 9 according to $ut(r)$ **then** `leftExpansion` (r, SDB, $minutil$, $minconf$);
12 **end**

Algorithm 2: The leftExpansion procedure

input : r: a sequential rule $X \rightarrow Y$, SDB, $minutil$ and $minconf$

$rules \leftarrow \emptyset$;
foreach *sequence* $s \in seq(r)$ according to $ut(r)$ **do**
    foreach *rule* $t : X \cup \{i\} \rightarrow Y | i \in leftRight(t, s) \cup onlyLeft(t, s)$ **do**
    $rules \leftarrow rules \cup \{t\}$; Update $ut(t)$;
end
foreach *rule* $r \in rules$ **do**
    if $u(r) \geq minutil$ according to $ut(r)$ and $conf_{SDB}(r) \geq minconf$ **then** output r;
    if r respects Property 9 according to $ut(r)$ **then** `leftExpansion` (r, SDB, $minutil$, $minconf$);
end

3.3 Additional Optimizations

Two additional optimizations are added to HUSRM to further increase its efficiency. The first one reduces the size of utility-tables. It is based on the observations that in the *leftExpansion* procedure, (1) the $rutil$ values of utility-tables are never used and (2) that $lutil$ and $lrutil$ values are always summed. Thus, (1) the $rutil$ values can be dropped from utility-tables in *leftExpansion* and (2) the sum of $lutil$ and $lrutil$ values can replace both values. We refer to the resulting utility-tables as *Compact Utility-Tables* (CUT). For example, the CUT of $\{a, b\} \rightarrow \{e\}$ and $\{a, b, c\} \rightarrow \{e\}$ are respectively $\{(s_1, 6, 21), (s_2, 6, 32), (s_3, 6, 9)\}$ and $\{(s_1, 16, 11), (s_2, 26, 12)\}$. CUT are much smaller than utility-tables since each tuple contains only three elements instead of five. It is also much less expensive to update CUT.

The second optimization reduces the time for scanning sequences in the *leftExpansion* and *rightExpansion* procedures. It introduces two definitions. The *first occurrence* of an itemset X in a sequence $s = \langle I_1, I_2, ...I_n \rangle$ is the itemset $I_k \in s$ such that $X \subseteq \bigcup_{i=1}^{k} I_i$ and there exists no $g < k$ such that $X \subseteq \bigcup_{i=1}^{g} I_i$. The *last occurrence* of an itemset X in a sequence $s = \langle I_1, I_2, ...I_n \rangle$ is the itemset $I_k \in s$ such that $X \subseteq \bigcup_{i=k}^{n} I_i$ and there exists no $g > k$ such that $X \subseteq \bigcup_{i=g}^{n} I_i$. An

Algorithm 3: The rightExpansion procedure

input : r: a sequential rule $X \rightarrow Y$, SDB, $minutil$ and $minconf$

1 $rules \leftarrow \emptyset$;

2 **foreach** *sequence* $s \in seq(r)$ according to $ut(r)$ **do**

3 **foreach** *rule* t *of the form* $X \cup \{i\} \rightarrow Y$ *or* $X \rightarrow Y \cup \{i\}$ $|i \in leftRight(t,s) \cup onlyLeft(t,s) \cup onlyRight(t,s)$ **do** $rules \leftarrow rules \cup \{t\}$; Update $ut(t)$;

4 **end**

5 **foreach** *rule* $r \in rules$ **do**

6 **if** $u(r) \geq minutil$ according to $ut(r)$ and $conf_{SDB}(r) \geq minconf$ **then** output r;

7 **if** r respects Property 8 according to $ut(r)$ **then** `rightExpansion` (r, SDB, $minutil$, $minconf$);

8 **if** r respects Property 9 according to $ut(r)$ **then** `leftExpansion` (r, SDB, $minutil$, $minconf$);

9 **end**

important observation is that a rule $X \rightarrow Y$ can only be expanded with items appearing after the first occurrence of X for a right expansion, and occurring before the last occurrence of Y for a left expansion. The optimization consists of keeping track of the first and last occurrences of rules and to use this information to avoid scanning sequences completely when searching for items to expand a rule. This can be done very efficiently by first storing the first and last occurrences of rules of size $1 * 1$ and then only updating the first (last) occurrences when performing a left (right) expansion.

4 Experimental Evaluation

We performed experiments to evaluate the performance of the proposed algorithm. Experiments were performed on a computer with a fourth generation 64 bit core i7 processor running Windows 8.1 and 16 GB of RAM. All memory measurements were done using the Java API.

Experiments were carried on four real-life datasets commonly used in the pattern mining lit-

erature: *BIBLE*, *FIFA*, *KOSARAK* and *SIGN*. These datasets have varied characteristics and represents the main types of data typically encountered in real-life scenarios (dense, sparse, short and long sequences). The characteristics of datasets are shown in table 4), where the $|SDB|$, $|I|$ and $avgLength$ columns respectively indicate the number of sequences, the number of distinct items and the average sequence length. *BIBLE* is moderately dense and contains many medium length sequences. *FIFA* is moderately dense and contains many long sequences. *KOSARAK* is a sparse dataset that contains short sequences and a few very long sequences. *SIGN* is a dense dataset having very long sequences. For all datasets, external utilities of items are generated between 0 and 1,000 by using a log-normal distribution and quantities of items are generated randomly between 1 and 5, similarly to the settings of [2, 11, 18].

Dataset	$\|SDB\|$	$\|I\|$	$avgLength$	Type of data
BIBLE	36,369	13,905	21.64	book
FIFA	573,060	13,749	45.32	click-stream
KOSARAK	638,811	39,998	11.64	click-stream
SIGN	730	267	93.00	sign language

Table I-4 Dataset characteristics

Because HUSRM is the first algorithm for high-utility sequential rule mining, we compared its performance with five versions of HUSRM where optimizations had been deactivated (HUSRM_1, $\text{HUSRM}_{1,2}$, $\text{HUSRM}_{1,2,3}$, $\text{HUSRM}_{1,2,3,4}$ and $\text{HUSRM}_{1,2,3,4,5}$). The notation $\text{HUSRM}_{1,2,...n}$ refers to HUSRM without optimizations O_1, O_2 ... O_n. Optimization 1 (O_1) is to ignore unpromising items. Optimization 2 (O_2) is to ignore unpromising rules. Optimization 3 (O_3) is to use bit vectors to calculate confidence instead of lists of integers. Optimization 4 (O_4) is to use compact utility-tables instead of utility-tables. Optimization 5 (O_5) is to use Property 9 to prune the search space for left expansions instead of Property 8. The source code of all algorithms and datasets can be downloaded as part of the SPMF data mining library at `http://goo.gl/qS7MbH` [7].

We ran all the algorithms on each dataset while decreasing the $minutil$ threshold until algorithms became too long to execute, ran out of memory or a clear winner was observed. For these experiments, we fixed the $minconf$ threshold to 0.70. However, note that results are similar for

other values of the $minconf$ parameter since the confidence is not used to prune the search space by the compared algorithms. For each dataset and algorithm, we recorded execution times and maximum memory usage.

Execution times. Fig. I-2 shows the execution times of each algorithm. Note that results for HUSRM$_{1,2,3,4,5}$ are not shown because it does not terminate in less than $10,000s$ for all datasets. HUSRM is respectively up to 1.8, 1.9, 2, 3.8 and 25 times faster than HUSRM$_1$, HUSRM$_{1,2}$, HUSRM$_{1,2,3}$, HUSRM$_{1,2,3,4}$ and HUSRM$_{1,2,3,4,5}$. It can be concluded from these results that HUSRM is the fastest on all datasets, that its optimizations greatly enhance its performance, and that O_5 is the most effective optimization to reduce execution time. In this experiment, we have found up to 100 rules, which shows that mining high-utility sequential rules is very expensive. Note that if we lower $minutil$, it is possible to find more than 10,000 rules using HUSRM.

Memory usage. Table 4 shows the maximum memory usage of the algorithms for the BIBLE, FIFA, KOSARAK and SIGN datasets. Results for HUSRM$_{1,2,3,4,5}$ are not shown for the same reason as above. It can be observed that HUSRM always consumes less memory and that this usage is up to about 50% less than that of HUSRM$_{1,2,3,4}$ on most datasets. The most effective optimization to reduce memory usage is O_4 (using compact utility-tables).

Dataset	HUSRM	HUSRM$_1$	HUSRM$_{1,2}$	HUSRM$_{1,2,3}$	HUSRM$_{1,2,3,4}$
BIBLE	**1,022**	1,177	1,195	1,211	1,346
FIFA	**1,060**	1,089	1,091	1,169	1,293
KOSARAK	**502**	587	594	629	1,008
SIGN	**1,053**	1,052	1,053	1053	1,670

Table I-5 Comparison of maximum memory usage (megabytes)

5 Conclusion

To address the lack of confidence measure in high-utility sequential pattern mining, we defined the problem of high-utility sequential rule mining and studied its properties. We proposed an efficient algorithm named HUSRM (High-Utility Sequential Rule Miner) to mine these rules. HUSRM

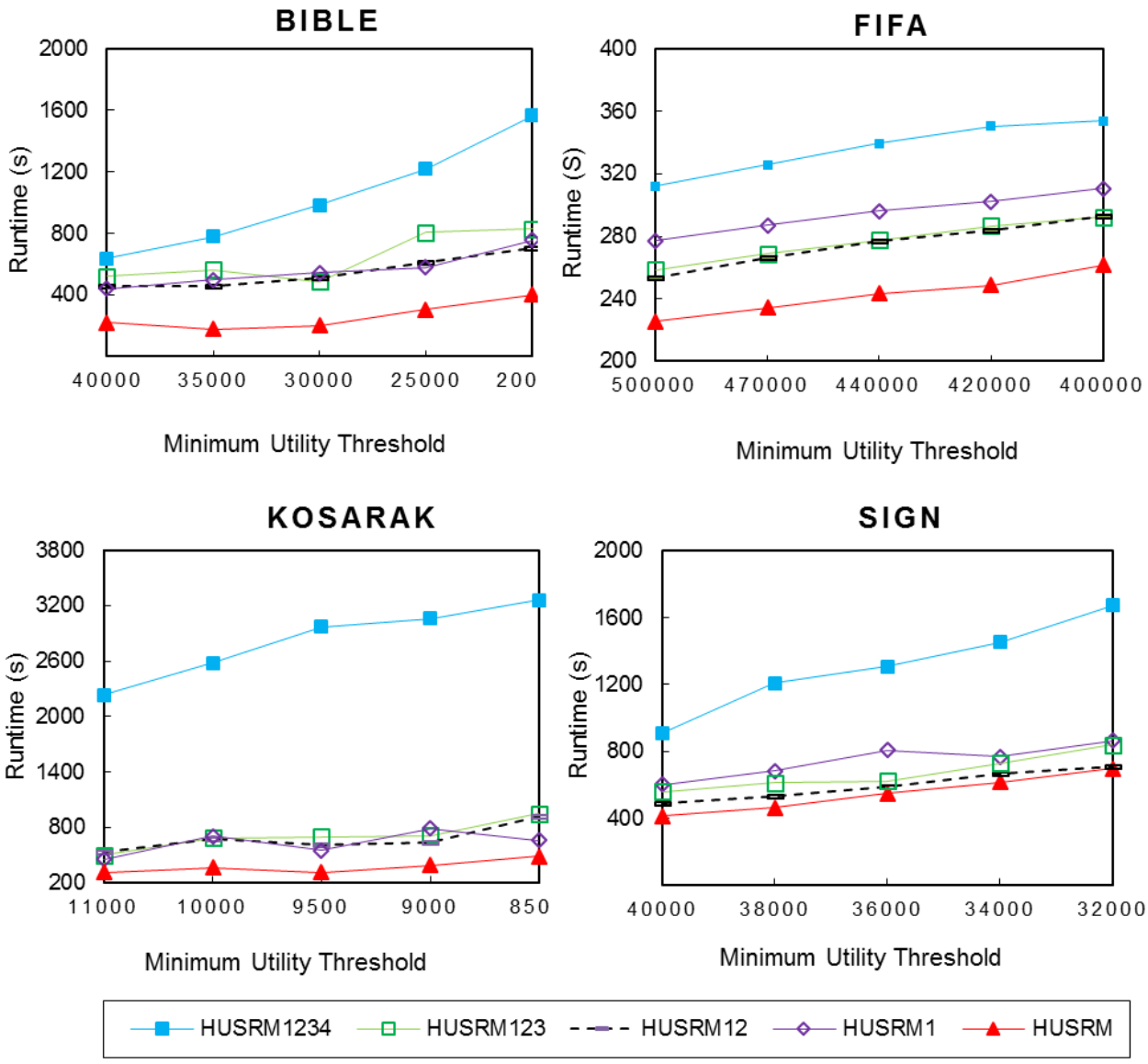

Figure I-2 Comparison of execution times (seconds)

is a one-phase algorithm that relies on a new data structure called compact utility-table and include several novel optimizations to mine rules efficiently. An extensive experimental study with four datasets shows that HUSRM is very efficient and that its optimizations respectively improve its execution time by up to 25 times and its memory usage by up to 50%.

CHAPITRE II

EFIM: EFFICIENT HIGH-UTILITY ITEMSET MINING

UN ALGORITHME EFFICIENT POUR LA DÉCOUVERTE DES ITEMSETS PROFITABLES

Abstract

High-utility itemset mining (HUIM) is an important data mining task with wide applications. However, it remains computationally expensive. In this paper, we address this issue by proposing an algorithm named EFIM (EFficient high-utility Itemset Mining), which introduces several new ideas to more efficiently discovers high-utility itemsets. It relies on two new upper-bounds named *sub-tree utility* and *local utility* to more effectively prune the search space. It also introduces a novel array-based utility counting technique named *Fast Utility Counting* to calculate these upper-bounds in linear time and space. Moreover, to reduce the cost of database scans, EFIM proposes efficient database projection and transaction merging techniques. An extensive experimental study on various datasets shows that EFIM is in general two to three orders of magnitude faster and consumes up to eight times less memory than the state-of-art algorithms d^{2}HUP, HUI-Miner, HUP-Miner, FHM and UP-Growth+, and that EFIM has excellent scalability and performs well on dense datasets.

Keywords: high-utility mining, itemset mining, pattern mining.

1 Introduction

Frequent Itemset Mining (FIM) [1] is a popular data mining task that is essential to a wide range of applications. Given a transaction database, FIM consists of discovering frequent itemsets. i.e. groups of items (itemsets) appearing frequently in transactions [1,6,31]. However, an important limitation of FIM is that it assumes that each item cannot appear more than once in each transaction and that all items have the same importance (weight, unit profit or value). These assumptions often do not hold in real applications. For example, consider a database of customer transactions containing information about the quantities of items in each transaction and the unit profit of each item. FIM algorithms would discard this information and may thus discover many frequent itemsets generating a low profit and fail to discover itemsets that generate a high profit.

To address these issues, the problem of *High-Utility Itemset Mining* (HUIM) has been defined [2, 5, 11, 12, 15, 18, 19, 27, 30]. As opposed to FIM [1, 6, 31], HUIM considers the case where

items can appear more than once in each transaction and where each item has a weight (e.g. unit profit). Therefore, it can be used to discover itemsets having a high-utility (e.g. high profit), that is *High-Utility Itemsets*. HUIM is an important research topic in data mining having a wide range of applications such as cross-marketing, click stream analysis and biomedical applications [2, 5, 11, 18]. Moreover, HUIM has inspired several important data mining tasks such as high-utility sequential pattern mining [21, 32].

The problem of HUIM is widely recognized as more difficult than the problem of FIM. In FIM, the *downward-closure property* states that the support (frequency) of an itemset is *anti-monotonic* [1], that is the supersets of an infrequent itemset are infrequent and subsets of a frequent itemset are frequent. This property is very powerful to prune the search space. In HUIM, the utility of an itemset is neither monotonic or anti-monotonic. That is, a HUI may have a superset or subset with lower, equal or higher utility [2, 5, 11, 18]. Thus techniques to prune the search space developed in FIM based on the anti-monotonicity of the support cannot be directly applied to HUIM.

Many studies have been proposed to develop efficient HUIM algorithms [2, 5, 11, 12, 15, 18, 19, 27, 30]. A popular approach to HUIM is to discover high-utility itemsets in two phases using the Transaction-Weigthed-Downward Closure model. This approach is adopted by algorithms such as IHUP [2], PB [27], Two-Phase [15], BAHUI [30], UP-Growth [18] and UP-Growth+ [19]. These algorithms first generate a set of candidate high-utility itemsets by overestimating their utility in Phase 1. Then, in Phase 2, the algorithms scan the database to calculate the exact utility of candidates and filter low-utility itemsets. However, the two-phase model suffers from the problem of generating a huge amount of candidates and repeatedly scanning the database. Recently, to avoid the problem of candidate generation, more efficient approaches were proposed in the HUI-Miner [11] and d^{2}HUP [28] algorithms to mine high-utility itemsets directly using a single phase, thus avoiding the problem of candidate generation. The d^{2}HUP and HUI-Miner algorithms were reported to be respectively up to 10 and 100 times faster than state-of-the-art two-phase algorithms [11, 28]. Then, improved versions of HUI-Miner named HUP-Miner [12] and FHM [5] were proposed. FHM and HUP-Miner were shown to be up to 6 times faster than HUI-Miner, and are to our knowledge the current best algorithms for HUIM. However, despite all these research efforts, the task of high-utility itemset mining remains very computationally expensive.

In this paper, we address this need for more efficient HUIM algorithms by proposing a one-phase algorithm named EFIM (EFficient high-utility Itemset Mining), which introduces several novel ideas to greatly decrease the time and memory required for HUIM.

— First, we propose two new techniques to reduce the cost of database scans. EFIM performs database projection and merges transactions that are identical in each projected database using a linear time and space implementation. Both operations reduce the size of the database as larger itemsets are explored, and thus considerably decrease the cost of database scans.
— Second, we propose two new upper-bounds on the utility of itemsets named *sub-tree utility* and *local utility* to more effectively prune the search space, and show that these upper-bounds are more effective at pruning the search space than the TWU and remaining utility upper-bounds used in previous work.
— Third, we introduce a novel array-based utility counting technique named *fast utility counting* (FAC) to calculate these upper-bounds in linear time and space.

We conducted an extensive experimental study to compare the performance of EFIM with the state-of-the-art algorithms d^{2}HUP, HUI-Miner, HUP-Miner, FHM and UP-Growth+ on various datasets. Results show that EFIM is in general two to three orders of magnitude faster than these algorithms, consumes up to eight times less memory, has excellent scalability and performs well on dense datasets.

The rest of this paper is organized as follows. Sections 2, 3, 4, 5 and 6 respectively presents the problem of HUIM, the related work, the EFIM algorithm, the experimental evaluation and the conclusion.

2 Problem Statement

We first introduce the problem of high-utility itemset mining.

Definition 1 (Transaction database). Let I be a finite set of items (symbols). An itemset X is a finite set of items such that $X \subseteq I$. A *transaction database* is a multiset of transactions $D = \{T_1, T_2, ..., T_n\}$ such that for each transaction T_c, $T_c \subseteq I$ and T_c has a unique identifier c called its

Table II-1 A transaction database

TID	Transaction
T_1	$(a,1)(c,1)(d,1)$
T_2	$(a,2)(c,6)(e,2)(g,5)$
T_3	$(a,1)(b,2)(c,1)(d,6)(e,1)(f,5)$
T_4	$(b,4)(c,3)(d,3)(e,1)$
T_5	$(b,2)(c,2)(e,1)(g,2)$

Table II-2 External utility values

Item	a	b	c	d	e	f	g
Profit	5	2	1	2	3	1	1

TID (Transaction ID). Each item $i \in I$ is associated with a positive number $p(i)$, called its *external utility* (e.g. unit profit). Every item i appearing in a transaction T_c has a positive number $q(i, T_c)$, called its *internal utility* (e.g. purchase quantity).

Example 1. Consider the database in Table II-1, which will be used as the running example. It contains five transactions ($T_1, T_2..., T_5$). Transaction T_2 indicates that items a, c, e and g appear in this transaction with an internal utility of respectively 2, 6, 2 and 5. Table II-2 indicates that the external utility of these items are respectively 5, 1, 3 and 1.

Definition 2 (Utility of an item/itemset in a transaction). The *utility of an item i in a transaction* T_c is denoted as $u(i, T_c)$ and defined as $p(i) \times q(i, T_c)$. An itemset X is a set of items such that $X \subseteq I$. The *utility of an itemset X in a transaction* T_c is denoted as $u(X, T_c)$ and defined as $u(X, T_c) = \sum_{i \in X} u(i, T_c)$.

Example 2. The utility of item a in T_2 is $u(a, T_2) = 5 \times 2 = 10$. The utility of the itemset $\{a, c\}$ in T_2 is $u(\{a, c\}, T_2) = u(a, T_2) + u(c, T_2) = 5 \times 2 + 1 \times 6 = 16$.

Definition 3 (Utility of an itemset in a database). The *utility of an itemset X* is denoted as $u(X)$ and defined as $u(X) = \sum_{T_c \in g(X)} u(X, T_c)$, where $g(X)$ is the set of transactions containing X.

Example 3. The utility of the itemset $\{a, c\}$ is $u(\{a, c\}) = u(a, T_1)+u(a, T_2)+u(a, T_3)+u(c, T_1)+ u(c, T_2) + u(c, T_3) = 5 + 10 + 5 + 1 + 6 + 1 = 28$.

Definition 4 (Problem definition). An itemset X is a *high-utility itemset* if its utility $u(X)$ is no less than a user-specified minimum utility threshold $minutil$ given by the user. Otherwise, X is a *low-utility itemset*. The *problem of high-utility itemset mining* is to discover all high-utility itemsets.

Example 4. If $minutil = 30$, the high-utility itemsets in the database of our running example are $\{b, d\}$, $\{a, c, e\}$, $\{b, c, d\}$, $\{b, c, e\}$, $\{b, d, e\}$, $\{b, c, d, e\}$ with respectively a utility of 30, 31, 34, 31, 36, 40 and 30.

3 Related Work

HUIM is harder than FIM since the utility measure is not monotonic or anti-monotonic [2, 15, 18], i.e. the utility of an itemset may be lower, equal or higher than the utility of its subsets. Thus, strategies used in FIM to prune the search space based on the anti-monotonicity of the frequency cannot be applied to the utility measure to discover high-utility itemsets. Several HUIM algorithms circumvent this problem by overestimating the utility of itemsets using the *Transaction-Weighted Utilization* (TWU) measure [2, 15, 18, 19, 27, 30], which is anti-monotonic, and defined as follows.

Definition 5 (Transaction weighted utilization). The *transaction utility* of a transaction T_c is the sum of the utilities of items from T_c in that transaction. i.e. $TU(T_c) = \sum_{x \in T_c} u(x, T_c)$. The *transaction-weighted utilization* (TWU) of an itemset X is defined as the sum of the transaction utilities of transactions containing X, i.e. $TWU(X) = \sum_{T_c \in g(X)} TU(T_c)$.

Example 5. The TU of transactions T_1, T_2, T_3, T_4 and T_5 for our running example are respectively 8, 27, 30, 20 and 11. The TWU of single items a, b, c, d, e, f and g are respectively 65, 61, 96, 58, 88, 30 and 38. Consider item a. $TWU(a) = TU(T_1) + TU(T_2) + TU(T_3) = 8 + 27 + 30 = 65$.

The following properties of the TWU are used to prune the search space.

Property 1 (Overestimation using the TWU). Let be an itemset X. The TWU of X is no less than its utility ($TWU(X) \geq u(X)$). Moreover, the TWU of X is no less than the utility of its supersets ($TWU(X) \geq u(Y) \forall Y \supset X$) [15].

Property 2 (Pruning search space using TWU). For any itemset X, if $TWU(X) < minutil$, then X is a low-utility itemset as well as all its supersets [15].

Algorithms such as IHUP [2], PB [27], Two-Phase [15], BAHUI [30], UP-Growth [18] and UP-Growth+ [19] utilize Property 2 to prune the search space. They operate in two phases. In the first phase, they identify candidate high-utility itemsets by calculating their TWUs. In the second phase, they scan the database to calculate the exact utility of all candidates to filter low-utility itemsets. UP-Growth is one of the fastest two-phase algorithm. It was shown to be up to 1,000 times faster than Two-phase and IHUP. More recent two-phase algorithms such as PB and BAHUI only provide a small speed improvement over Two-Phase or UP-Growth.

Recently, algorithms that mine high-utility itemsets using a single phase were proposed to avoid the problem of candidate generation. The d^{2}HUP and HUI-Miner algorithms were reported to be respectively up to 10 and 100 times faster than UP-Growth [11, 28]. Then, improved versions of HUI-Miner named HUP-Miner [12] and FHM [5] were proposed to reduce the number of join operations performed by HUI-Miner. FHM and HUP-Miner were shown to be up to 6 times faster than HUI-Miner, and are to our knowledge the current best algorithm for HUIM. In HUI-Miner, HUP-Miner and FHM, each itemset is associated with a structure named *utility-list* [5, 11]. Utility-lists allow calculating the utility of an itemset by making join operations with utility-lists of smaller itemsets. Utility-lists are defined as follows.

Definition 6 (Remaining utility). Let $\succ$ be a total order on items from I, and X be an itemset. The *remaining utility* of X in a transaction T_c is defined as $re(X, T_c) = \sum_{i \in T_c \wedge i \succ x \forall x \in X} u(i, T_c)$.

Definition 7 (Utility-list). The *utility-list* of an itemset X in a database D is a set of tuples such that there is a tuple $(c, iutil, rutil)$ for each transaction T_c containing X. The $iutil$ and $rutil$ elements of a tuple respectively are the utility of X in T_c ($u(X, T_c)$) and the remaining utility of X in T_c ($re(X, T_c)$).

Example 6. Assume the lexicographical order (i.e. $e \succ d \succ c \succ b \succ a$). The utility-list of $\{a, e\}$ is $\{(T_2, 16, 5), (T_3, 8, 5)\}$.

To discover high-utility itemsets, HUI-Miner, HUP-Miner and FHM perform a database scan to create utility-lists of patterns containing single items. Then, larger patterns are obtained by joining utility-lists of smaller patterns (see [11, 12] for details). Pruning the search space is done using the following properties.

Definition 8 (Remaining utility upper-bound). Let X be an itemset. Let the *extensions* of X be the itemsets that can be obtained by appending an item i to X such that $i \succ x$, $\forall x \in X$. The *remaining utility upper-bound* of X is defined as $reu(X) = u(X) + re(X)$, and can be computed by summing the $iutil$ and $rutil$ values in the utility-list of X.

Property 3 (Pruning search space using utility-lists). If $reu(X) < minutil$, then X is a low-utility itemset as well as all its extensions [11].

One-phase algorithms are faster than previous algorithms because they discover itemsets in one phase, thus avoiding the problem of candidate generation found in previous algorithms. However, mining HUIs remains a very computationally expensive tasks. For example, HUI-Miner, HUP-Miner, and FHM still suffer from a high space and time complexity. The size of each utility-list is in the worst case $O(n)$, where n is the number of transactions (when a utility-list contains an entry for each transaction). The complexity of building a utility-list is also very high [5]. In general, it requires to join three utility-lists of smaller itemsets, which thus requires $O(n^3)$ time in the worst case. FHM and HUP-Miner introduces strategies to reduce the number of join operations. However, joining utility-lists remain the main performance bottleneck.

4 The EFIM Algorithm

We next present our proposal, the EFIM algorithm. It is a one-phase algorithm, which introduces several novel ideas to reduce the time and memory required for HUIM. Subsection 4.1 introduces definitions related to the depth-first search of itemsets. Subsection 4.2 and 4.3 respectively explain how EFIM reduces the cost of database scans using an efficient implementation of database projection and transaction merging. Subsection 4.4 present two new upper-bounds used by EFIM to prune the search space. Subsection 4.5 presents a new array-based utility counting technique named

Fast Utility Counting to efficiently calculate these upper-bounds in linear time and space. Finally, subsection 4.6 gives the pseudocode of EFIM.

4.1 The Search Space

Let $\succ$ be any total order on items from I. According to this order, the search space of all itemsets 2^I can be represented as a *set-enumeration tree* [29]. For example, the set-enumeration tree of $I = \{a, b, c, d\}$ for the lexicographical order is shown in Fig. II-1. The EFIM algorithm explores this search space using a depth-first search starting from the root (the empty set). During this depth-first search, for any itemset α, EFIM recursively appends one item at a time to α according to the $\succ$ order, to generate larger itemsets. In our implementation, the $\succ$ order is defined as the order of increasing TWU because it generally reduces the search space for HUIM [2, 5, 11, 18, 19]. However, we henceforth assume that $\succ$ is the lexicographical order, for the sake of simplicity. We next introduce three definitions related to the depth-first search exploration of itemsets.

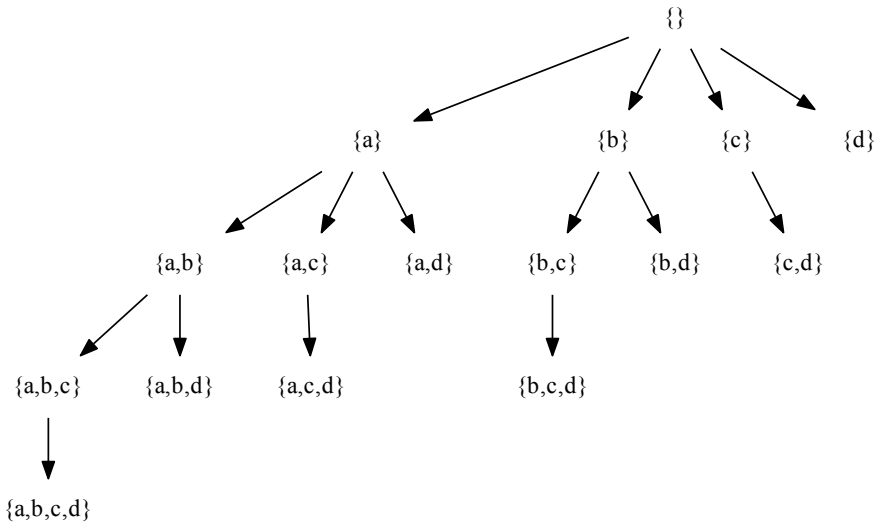

Figure II-1 Set-enumeration tree for $I = \{a, b, c, d\}$

Definition 9 (Items that can extend an itemset). Let be an itemset α. Let $E(\alpha)$ denote the set of all items that can be used to extend α according to the depth-first search, that is $E(\alpha) = \{z | z \in I \wedge z \succ x, \forall x \in \alpha\}$.

Definition 10 (Extension of an itemset). Let be an itemset α. An itemset Z is an *extension* of α (appears in a sub-tree of α in the set-enumeration tree) if $Z = \alpha \cup W$ for an itemset $W \in 2^{E(\alpha)}$ such that $W \neq \emptyset$.

Definition 11 (Single-item extension of an itemset). Let be an itemset α. An itemset Z is a *single-item extension* of α (is a child of α in the set-enumeration tree) if $Z = \alpha \cup \{z\}$ for an item $z \in E(\alpha)$.

Example 7. Consider the database of our running example and $\alpha = \{d\}$. The set $E(\alpha)$ is $\{e, f, g\}$. Single-item extensions of α are $\{d, e\}$, $\{d, f\}$ and $\{d, g\}$. Other extensions of α are $\{d, e, f\}$, $\{d, f, g\}$ and $\{d, e, f, g\}$.

4.2 Reducing the Cost of Database Scans using Projections

As we will later explain, EFIM performs database scans to calculate the utility of itemsets and upper-bounds on their utility. To reduce the cost of database scans, it is desirable to reduce the database size. An observation is that when an itemset α is considered during the depth-first search, all items $x \notin E(\alpha)$ can be ignored when scanning the database to calculate the utility of itemsets in the sub-tree of α, or upper-bounds on their utility. A database without these items is called a *projected database*.

Definition 12 (Projected transaction). The *projection of a transaction T using an itemset α* is denoted as α-*T* and defined as α-$T = \{i | i \in T \wedge i \in E(\alpha)\}$.

Definition 13 (Projected database). The *projection of a database D using an itemset α* is denoted as α-*D* and defined as the multiset α-$D = \{\alpha\text{-}T | T \in D \wedge \alpha\text{-}T \neq \emptyset\}$.

Example 8. Consider database D of the running example and $\alpha = \{b\}$. The projected database α-*D* contains three transactions: α-$T_3 = \{c, d, e, f\}$, α-$T_4 = \{c, d, e\}$ and α-$T_5 = \{c, e, g\}$.

Database projections generally greatly reduce the cost of database scans since transactions become smaller as larger itemsets are explored. However, an important issue is how to implement

database projection efficiently. A naive and inefficient approach is to make physical copies of transactions for each projection. An efficient approach used in EFIM is to sort each transaction in the original database according to the $\succ$ total order beforehand. Then, a projection is performed as a *pseudo-projection*, that is each projected transaction is represented by an offset pointer on the corresponding original transaction. The complexity of calculating the projection α-*D* of a database *D* is linear in time and space with respect to the number of transactions.

The proposed database projection technique is a generalization of the concept of database projection used in frequent pattern mining [13, 31] for the case of transactions with internal/external utility values. Note that FP-growth based HUIM algorithms [2, 18, 19, 27] and hyper-link based HUIM algorithms [28] also perform projections but differently than the EFIM algorithm since they use different database representations (tree and hyperlink representations).

4.3 Reducing the Cost of Database Scans by Transaction Merging

To further reduce the cost of database scans, EFIM also introduce an efficient transaction merging technique. It is based on the observation that transaction databases often contain identical transactions. The technique consists of identifying these transactions and to replace them with single transactions.

Definition 14 (Identical transactions). A transaction T_a is *identical to* a transaction T_b if it contains the same items as T_b (i.e. $T_a = T_b$) (but not necessarily the same internal utility values).

Definition 15 (Transaction merging). *Transaction merging* consists of replacing a set of identical transactions $Tr_1, Tr_2, ...Tr_m$ in a database D by a single new transaction $T_M = Tr_1 = Tr_2 = ... = Tr_m$ where the quantity of each item $i \in T_M$ is defined as $q(i, T_M) = \sum_{k=1...m} q(i, Tr_k)$.

Merging identical transactions reduce the size of the database. But this reduction is small if the database contains few identical transactions. To achieve higher database reduction, we also merge transactions in projected databases. This generally achieves a much higher reduction because projected transactions are smaller than original transactions, and thus are more likely to be identical.

Definition 16 (Projected transaction merging). *Projected transaction merging* consists of replacing a set of identical transactions $Tr_1, Tr_2, ...Tr_m$ in a database α-D by a single new transaction $T_M = Tr_1 = Tr_2 = ... = Tr_m$ where the quantity of each item $i \in T_M$ is defined as $q(i, T_M) = \sum_{k=1...m} q(i, Tr_k)$.

Example 9. Consider database D of our running example and $\alpha = \{c\}$. The projected database α-D contains transactions α-$T_1 = \{d\}$, α-$T_2 = \{e, g\}$, α-$T_3 = \{d, e, f\}$, α-$T_4 = \{d, e\}$ and α-$T_5 = \{e, g\}$. Transactions α-T_2 and α-T_5 can be replaced by a new transaction $T_M = \{e, g\}$ where $q(e, T_M) = 3$ and $q(g, T_M) = 7$.

Transaction merging is obviously desirable. However, a key problem is to implement it efficiently. The naive approach to identify identical transactions is to compare all transactions with each other. But this is inefficient ($O(n^2)$, where n is the number of transactions). To find identical transactions in $O(n)$ time, we propose the following approach. We initially sort the original database according to a new total order $\succ_T$ on transactions. Sorting is achieved in $O(n\ log(n))$ time, and is performed only once.

Definition 17 (Total order on transactions). The $\succ_T$ order is defined as the lexicographical order when the transactions are read backwards. Formally, let be two transactions $T_a = \{i_1, i_2, ...i_m\}$ and $T_b = \{j_1, j_2, ...j_k\}$. The total order $\succ_T$ is defined by four cases. The first case is that $T_b \succ T_a$ if both transactions are identical and T_b is greater than the TID of T_a. The second case is that $T_b \succ_T T_a$ if $k > m$ and $i_{m-x} = j_{k-x}$ for any integer x such that $0 \leq x < m$. The third case is that $T_b \succ_T T_a$ if there exists an integer x such that $0 \leq x < \min(m, k)$, where $j_{k-x} \succ i_{m-x}$ and $i_{m-y} = j_{k-y}$ for all integer y such that $x < y < \min(m, k)$. The fourth case is that otherwise $T_a \succ_T T_b$.

Example 10. Let be transactions $T_x = \{b, c\}$, $T_y = \{a, b, c\}$ and $T_z = \{a, b, e\}$. We have that $T_z \succ_T T_y \succ_T T_x$.

A database sorted according to the $\succ_T$ order provides the following property.

Property 4 (Order of identical transactions in a $\succ_T$ sorted database). Let be a $\succ_T$ sorted database D and an itemset α. Identical transactions appear consecutively in the projected database α-D.

Proof. Because (1) transactions are sorted in lexicographical order when read backwards and (2) projections always removes the smallest items of a transactions according to the lexicographical order, it is clear that the property holds.

Using the above property, all identical transactions in a (projected) database can be identified by only comparing each transaction with the next transaction in the database. Thus, using this scheme, transaction merging can be done very efficiently by scanning a (projected) database only once (linear time).

It is interesting to note that transaction merging as proposed in EFIM is not performed in any other one-phase HUIM algorithms. The reason is that it cannot be implemented efficiently in utility-list based algorithms such as HUP-Miner, HUI-Miner and FHM, and hyperlink-based algorithms such as d^{2}HUP due to their database representations.

4.4 Pruning Search Space using Sub-tree Utility and Local Utility

To propose an efficient HUIM algorithm, a key problem is to design an effective mechanism for pruning itemsets in the search space. For this purpose, we introduce in EFIM two new upper-bounds on the utility of itemsets named *sub-tree utility* and *local utility*. As we will explain, these upper-bounds have similarities to the remaining utility and TWU upper-bounds. But a key difference with these upper-bounds is that the proposed upper-bounds are defined w.r.t the sub-tree of an itemset α in the search-enumeration tree. Moreover, as we will explain, some items can be ignored when calculating these upper-bounds. This makes the proposed upper-bounds more effective at pruning the search space.

The proposed local utility upper-bound is defined as follows.

Definition 18 (Local utility). Let be an itemset α and an item $z \in E(\alpha)$. The *Local Utility* of z w.r.t. α is $lu(\alpha, z) = \sum_{T \in g(\alpha \cup \{z\})} [u(\alpha, T) + re(\alpha, T)]$.

Example 11. Consider the running example and $\alpha = \{a\}$. We have that $lu(\alpha, c) = (8 + 27 + 30) = 65$, $lu(\alpha, d) = 30$ and $lu(\alpha, e) = 57$.

The following theorem of the local utility is proposed in EFIM to prune the search space.

Property 5 (Overestimation using the local utility). Let be an itemset α and an item $z \in E(\alpha)$. Let Z be an extension of α such that $z \in Z$. The relationship $lu(\alpha, z) \geq u(Z)$ holds.

Proof. Let Y denotes the itemset $\alpha \cup \{z\}$. The utility of Z is equal to $u(Z) = \sum_{T \in g(Z)} u(Z, T)$ $= \sum_{T \in g(Z)} [u(\alpha, T) + u(Z \setminus \alpha, T)]$. The local-utility of α w.r.t to z is equal to $lu(\alpha, z) = \sum_{T \in g(Y)} [u(\alpha, T) + re(\alpha, T)]$. Because $g(Z) \subseteq g(Y)$ and $Z \setminus Y \subseteq E(\alpha)$, it follows that $u(Z \setminus \alpha, T) \leq re(\alpha, T)$ and thus that $lu(\alpha, z) \geq u(Z)$.

Theorem 4.1 (Pruning an item from all sub-trees using the local utility). Let be an itemset α and an item $z \in E(\alpha)$. If $lu(\alpha, z) < minutil$, then all extensions of α containing z are low-utility. In other words, item z can be ignored when exploring all sub-trees of α.

Thus, by using Theorem 4.1, some items can be pruned from all sub-trees of an itemset α, which reduces the number of itemsets to be considered. To further reduce the search space, we also identify whole sub-trees of α that can be pruned by proposing another upper-bound named the sub-tree utility, which is defined as follows.

Definition 19 (Sub-tree utility). Let be an itemset α and an item z that can extend α according to the depth-first search ($z \in E(\alpha)$). The *Sub-tree Utility* of z w.r.t. α is $su(\alpha, z) = \sum_{T \in g(\alpha \cup \{z\})} [u(\alpha, T) + u(z, T) + \sum_{i \in T \wedge i \in E(\alpha \cup \{z\}} u(i, T)]$.

Example 12. Consider the running example and $\alpha = \{a\}$. We have that $su(\alpha, c) = (5 + 1 + 2)$ $+(10 + 6 + 11)$ $+(5 + 1 + 20) = 61$, $su(\alpha, d) = 25$ and $su(\alpha, e) = 34$.

The following theorem of the sub-tree utility is proposed in EFIM to prune the search space.

Property 6 (Overestimation using the sub-tree utility). Let be an itemset α and an item $z \in E(\alpha)$. The relationship $su(\alpha, z) \geq u(\alpha \cup \{z\})$ holds. And more generally, $su(\alpha, z) \geq u(Z)$ holds for any extension Z of $\alpha \cup \{z\}$.

Proof. Suppose α is an itemset and $Y = \alpha \cup \{z\}$ is an extension of α. The utility of Y is equal to $u(Y) = \sum_{T \in g(Y)} u(Y, T)$. The sub-tree utility of α w.r.t. z is equal to $su(\alpha, z) = \sum_{T \in g(Y)} [u(\alpha, T) + u(z, T) + \sum_{i \in T \wedge i \in E(\alpha \cup \{z\})} u(i, T)] = \sum_{T \in g(Y)} u(Y, T)$ $+ \sum_{T \in g(Y)} \sum_{i \in T \wedge i \in E(\alpha \cup \{z\})} u(i, T) \geq \sum_{T \in g(Y)} u(Y, T)$. Thus, $su(\alpha, z) \geq u(Y)$. Now let's consider another itemset Z that is an extension of Y. The utility of Z is equal to $u(Z) = \sum_{T \in g(Z)} u(Y, T)$ $+ \sum_{T \in g(Z)} u(Z \setminus Y, T)$. Because $g(Z) \subseteq g(Y)$ and $Z \setminus Y \subseteq E(\alpha)$, it follows that $su(\alpha, z) \geq u(Z)$.

Theorem 4.2 (Pruning a sub-tree using the sub-tree utility). Let be an itemset α and an item $z \in E(\alpha)$. If $su(\alpha, z) < minutil$, then the single item extension $\alpha \cup \{z\}$ and its extensions are low-utility. In other words, the sub-tree of $\alpha \cup \{z\}$ in the set-enumeration tree can be pruned.

The relationships between the proposed upper-bounds and the main ones used in previous work are the following.

Property 7 (Relationships between upper-bounds). Let be an itemset α, an item z and an itemset $Y = \alpha \cup \{z\}$. The relationship $TWU(Y) \geq lu(\alpha, z) \geq reu(Y) = su(\alpha, z)$ holds.

Proof. $TWU(Y) = \sum_{T \in g(Y)} TU(T)$ and $lu(\alpha \cup \{z\}) = \sum_{T \in g(Y)} [u(\alpha, T) + re(\alpha, T)]$. Since $u(\alpha, T) + re(\alpha, T)$ cannot be greater than $TU(T)$ for any transaction T, the relationship $TWU(Y) \geq lu(\alpha \cup \{z\})$ holds. Moreover, because $reu(Y) = \sum_{T \in g(Y)} [u(Y, T) + re(\alpha \cup \{z\}, T)]$ and $re(\alpha, T) \geq re(\alpha \cup \{z\}, T)$, the relationship $lu(\alpha, z) \geq reu(Y)$ holds. Lastly, by definition $su(\alpha, z) = \sum_{T \in g(Y)} [u(Y, T) + \sum_{i \in T \wedge i \in E(Y)} u(i, T)]$ and $re(Y, T)$ is equal to $\sum_{i \in T \wedge i \in E(Y)} u(i, T)$, the relationship $reu(Y) = su(\alpha, z)$ holds.

Given, the above relationship, it can be seen that the proposed local utility upper-bound is a tighter upper-bound on the utility of Y and its extensions compared to the TWU, which is commonly used in two-phase HUIM algorithms. Thus the local utility can be more effective for pruning the search space.

About the su upper-bound, one can ask what is the difference between this upper-bound and the reu upper-bound of HUI-Miner and FHM since they are mathematically equivalent. A major

difference is that su is calculated when the depth-first search is at itemset α in the search tree rather than at the child itemset Y. Thus, if $su(\alpha, z) < minutil$, EFIM prunes the whole sub-tree of z including node Y rather than only pruning the descendant nodes of Y. Thus, using su instead of reu is more effective for pruning the search space.

Moreover, we make the su upper-bound even tighter by redefining it as follows.

Definition 20 (Primary and secondary items). Let be an itemset α. The *primary items of* α is the set of items defined as $Primary(\alpha) = \{z | z \in E(\alpha) \wedge su(\alpha, z) \geq minutil\}$. The *secondary items of* α is the set of items defined as $Secondary(\alpha) = \{z | z \in E(\alpha) \wedge lu(\alpha, z) \geq minutil\}$. Because $lu(\alpha, z) \geq su(\alpha, z)$, $Primary(\alpha) \subseteq Secondary(\alpha)$.

Example 13. Consider the running example and $\alpha = \{a\}$. $Primary(\alpha) = \{c, e\}$. $Secondary(\alpha) = \{c, d, e\}$. This means that w.r.t. α, only the sub-trees rooted at nodes $\alpha \cup \{c\}$ and $\alpha \cup \{e\}$ should be explored. Furthermore, in these subtrees, no items other than c, d and e should be considered.

The redefined (tighter) su upper-bound is defined as follows.

Definition 21 (Redefined Sub-tree utility). Let be an itemset α and an item z. The redefined sub-tree utility of item z w.r.t. itemset α is defined as: $su(\alpha, z) = \sum_{T \in g(\alpha \cup \{z\})} [u(\alpha, T) + u(z, T) + \sum_{i \in T \wedge i \in E(\alpha \cup \{z\}) \wedge i \in Secondary(\alpha)} u(i, T)]$.

The difference between the su upper-bound and the redefined su upper-bound is that in the latter, items not in $Secondary(\alpha)$ will not be included in the calculation of the su upper-bound. Thus, this redefined upper-bound is always less than or equal to the original su upper-bound and the reu upper-bound. It can be easily proven that the redefined su upper-bound preserves the pruning Theorem of the su upper bound. In the rest of the paper, it is assumed that the redefined su upper-bound is used instead of the original su upper-bound.

Lastly, it is interesting to note that the redefined su upper-bound cannot be applied in vertical algorithms such as HUI-Miner and FHM, since transactions are not represented explicitly. These

latter algorithms store the remaining utility of an itemset in each transaction in their utility-list structure but cannot reduce the remaining utility at further stages of the depth-first search, as it would require to scan the original transactions. The next section explains how EFIM calculates its upper-bounds efficiently, using its horizontal database representation.

4.5 Calculating Upper-Bounds Efficiently using Utility-Bins

In the previous subsection, we introduced two new upper-bounds on the utility of itemsets to prune the search space. We now present a novel efficient array-based approach to compute these upper-bounds in linear time and space that we call Fast Utility Counting (FUC). It relies on a novel array structure called utility-bin.

Definition 22 (Utility-Bin). Let be the set of items I appearing in a database D. A *utility-bin array* U for database D is an array of length $|I|$, having an entry denoted as $U[z]$ for each item $z \in I$. Each entry is called a *utility-bin* and is used to store a utility value (an integer in our implementation, initialized to 0).

A utility-bin array can be used to efficiently calculate the following upper-bounds in $O(n)$ time (recall that n is the number of transactions), as follows.

Calculating the TWU of all items. A utility-bin array U is initialized. Then, for each transaction T of the database, the utility-bin $U[z]$ for each item $z \in T$ is updated as $U[z] = U[z] + TU(T)$. At the end of the database scan, for each item $k \in I$, the utility-bin $U[k]$ contains $TWU(k)$.

Calculating the sub-tree utility w.r.t. an itemset α. A utility-bin array U is initialized. Then, for each transaction T of the database, the utility-bin $U[z]$ for each item $z \in T \cap E(\alpha)$ is updated as $U[z] = U[z] + u(\alpha, T) + u(z, T) + \sum_{i \in T \wedge i \succ z} u(i, T)$. Thereafter, we have $U[k] = su(\alpha, k)\ \forall k \in I$.

Calculating the local utility w.r.t. an itemset α. A utility-bin array U is initialized. Then, for each transaction T of the database, the utility-bin $U[z]$ for each item $z \in T \cap E(\alpha)$ is updated as $U[z] = U[z] + u(\alpha, T) + re(\alpha, T)$. Thereafter, we have $U[k] = lu(\alpha, k)\ \forall k \in I$.

Thus, by the above approach, the three upper-bounds can be calculated for all items that can extend an itemset α with only one (projected) database scan. Furthermore, it can be observed that utility-bins are a very compact data structure ($O(|I|)$ size). To utilize utility-bins more efficiently, we propose three optimizations. First, all items in the database are renamed as consecutive integers. Then, in a utility-bin array U, the utility-bin $U[i]$ for an item i is stored in the i-th position of the array. This allows to access the utility-bin of an item in $O(1)$ time. Second, it is possible to reuse the same utility-bin array multiple times by reinitializing it with zero values before each use. This avoids creating multiple arrays and thus greatly reduces memory usage. In our implementation, only three utility-bin arrays are created, to respectively calculate the TWU, sub-tree utility and local utility. Third, when reinitializing a utility-bin array to calculate the sub-tree utility or the local utility of single-item extensions of an itemset α, only utility-bins corresponding to items in $E(\alpha)$ are reset to 0, for faster reinitialization of the utility-bin array.

4.6 The Proposed Algorithm

In this subsection, we present the EFIM algorithm, which combines all the ideas presented in the previous section. The main procedure (Algorithm 4) takes as input a transaction database and the $minutil$ threshold. The algorithm initially considers that the current itemset α is the empty set. The algorithm then scans the database once to calculate the local utility of each item w.r.t. α, using a utility-bin array. Note that in the case where $\alpha = \emptyset$, the local utility of an item is its TWU. Then, the local utility of each item is compared with $minutil$ to obtain the secondary items w.r.t to α, that is items that should be considered in extensions of α. Then, these items are sorted by ascending order of TWU and that order is thereafter used as the $\succ$ order (as suggested in [2, 5, 11]). The database is then scanned once to remove all items that are not secondary items w.r.t to α since they cannot be part of any high-utility itemsets by Theorem 4.1. If a transaction becomes empty, it is removed from the database. Then, the database is scanned again to sort transactions by the $\succ_T$ order to allow $O(n)$ transaction merging, thereafter. Then, the algorithm scans the database again to calculate the sub-tree utility of each secondary item w.r.t. α, using a utility-bin array. Thereafter, the algorithm calls the recursive procedure $Search$ to perform the depth first search starting from α.

Algorithm 4: The EFIM algorithm

input : D: a transaction database, $minutil$: a user-specified threshold

output: the set of high-utility itemsets

1 $\alpha = \emptyset$;

2 Calculate $lu(\alpha, i)$ for all items $i \in I$ by scanning D, using a utility-bin array;

3 $Secondary(\alpha) = \{i | i \in I \wedge lu(\alpha, i) \geq minutil\}$;

4 Let $\succ$ be the total order of TWU ascending values on $Secondary(\alpha)$;

5 Scan D to remove each item $i \notin Secondary(\alpha)$ from the transactions, and delete empty transactions;

6 Sort transactions in D according to $\succ_T$;

7 Calculate the sub-tree utility $su(\alpha, i)$ of each item $i \in Secondary(\alpha)$ by scanning D, using a utility-bin array;

8 $Primary(\alpha) = \{i | i \in Secondary(\alpha) \wedge su(\alpha, i) \geq minutil\}$;

9 `Search` (α, D, $Primary(\alpha)$, $Secondary(\alpha)$, $minutil$);

The $Search$ procedure (Algorithm 5) takes as parameters the current itemset to be extended α, the α projected database, the primary and secondary items w.r.t α and the $minutil$ threshold. The procedure performs a loop to consider each single-item extension of α of the form $\beta = \alpha \cup \{i\}$, where i is a primary item w.r.t α (since only these single-item extensions of α should be explored according to Theorem 4.2). For each such extension β, a database scan is performed to calculate the utility of β and at the same time construct the β projected database. Note that transaction merging is performed whilst the β projected database is constructed. If the utility of β is no less than $minutil$, β is output as a high-utility itemset. Then, the database is scanned again to calculate the sub-tree and local utility w.r.t β of each item z that could extend β (the secondary items w.r.t to α), using two utility-bin arrays. This allows determining the primary and secondary items of β. Then, the $Search$ procedure is recursively called with β to continue the depth-first search by extending β. Based on properties and theorems presented in previous sections, it can be seen that when EFIM terminates, all and only the high-utility itemsets have been output.

Complexity. The complexity of EFIM can be analyzed as follows. In terms of time, a $O(n\, log(n))$

sort is performed once. Then, to process each primary itemset α encountered during the depth-first search, EFIM performs database projection, transaction merging and upper-bound calculation in $O(n)$ time. In terms of space, utility-bin arrays are created once and require $O(|I|)$ space. The database projection performed for each primary itemset α requires at most $O(n)$ space.

Algorithm 5: The $Search$ procedure

input : α: an itemset, α-*D*: the α projected database, $Primary(\alpha)$: the primary items of α, $Secondary(\alpha)$: the secondary items of α, the $minutil$ threshold

output: the set of high-utility itemsets that are extensions of α

1 **foreach** *item* $i \in Primary(\alpha)$ **do**
2 $\beta = \alpha \cup \{i\}$;
3 Scan α-*D* to calculate $u(\beta)$ and create β-*D*; `// uses transaction merging`
4 **if** $u(\beta) \geq minutil$ **then** output β;
5 Calculate $su(\beta, z)$ and $lu(\beta, z)$ for all item $z \in Secondary(\alpha)$ by scanning β-*D* once, using two utility-bin arrays;
6 $Primary(\beta) = \{z \in Secondary(\alpha) | su(\beta, z) \geq minutil\}$;
7 $Secondary(\beta) = \{z \in Secondary(\alpha) | lu(\beta, z) \geq minutil\}$;
8 `Search` (β, β-*D*, $Primary(\beta)$, $Secondary(\beta)$, $minutil$);
9 **end**

5 Experimental Results

We performed several experiments to evaluate the performance of the proposed EFIM algorithm. Experiments were carried out on a computer with a fourth generation 64 bit core i7 processor running Windows 8.1 and 16 GB of RAM. We compared the performance of EFIM with the state-of-the-art algorithms UP-Growth+, HUP-Miner, d^{2}HUP, HUI-Miner and FHM. Moreover, to evaluate the influence of the design decisions in EFIM, we also compared it with two versions of EFIM named EFIM(nop) and EFIM(lu) where transaction merging and search space pruning using the sub-tree utility were respectively deactivated.

Algorithms were implemented in Java and memory measurements were done using the Java

API. Experiments were performed using a set of standard datasets used in the HUIM literature for evaluating HUIM algorithms, namely (*Accident*, *BMS*, *Chess*, *Connect*, *Foodmart* and *Mushroom*). Table 4 summarizes their characteristics. *Foodmart* contains real external/internal utility values. For other datasets, external/internal utility values have been respectively generated in the $[1, 000]$ and $[1, 5]$ intervals using a log-normal distribution, as done in previous state-of-the-art HUIM studies [2, 5, 11, 19]. The datasets and the source code of the compared algorithms can be downloaded at `http://goo.gl/9ay6Vw`.

Table II-3 Dataset characteristics

Dataset	# Transactions	# Distinct items	Avg. trans. length
Accident	340,183	468	33.8
BMS	59,601	497	4.8
Chess	3,196	75	37.0
Connect	67,557	129	43.0
Foodmart	4,141	1,559	4.4
Mushroom	8,124	119	23.0

Influence of the $minutil$ threshold on execution time. We first compare execution times of the various algorithms. We ran the algorithms on each dataset while decreasing the $minutil$ threshold until algorithms were too slow, ran out of memory or a clear winner was observed. Execution times are shown in Fig. II-2. Note that for UP-Growth+, no result is shown for the connect dataset and that some results are missing for the chess dataset because UP-Growth+ exceeded the 16 GB memory limit. It can be observed that EFIM clearly outperforms UP-Growth+, HUP-Miner, d^{2}HUP, HUI-Miner and FHM on all datasets. EFIM is in general about two to three orders of magnitude faster than these algorithms. For *Accident*, *BMS*, *Chess*, *Connect*, *Foodmart* and *Mushroom*, EFIM is respectively up to 15,334, 2, 33,028, –, 17 and 3,855 times faster than UP-Growth+, 154, 741, 323, 22,636, 2 and 85 times faster than HUP-Miner, 89, 1,490, 109, 2,587, 1 and 15 times faster than d^{2}HUP, 236, 2,370, 482, 10,586, 3 and 110 times faster than HUI-Miner and 145, 227, 321, 6,606, 1 and 90 times faster than FHM. The main reasons why EFIM performs so well are, as we will show in the following experiments, (1) its proposed upper-bounds that allows EFIM to prune a larger

part of the search space compared to other algorithms, and (2) its proposed transaction merging technique for reducing the cost of database scans. Beside, its efficient calculation of upper-bounds in linear time using utility-bins also contribute to the time efficiency of EFIM.

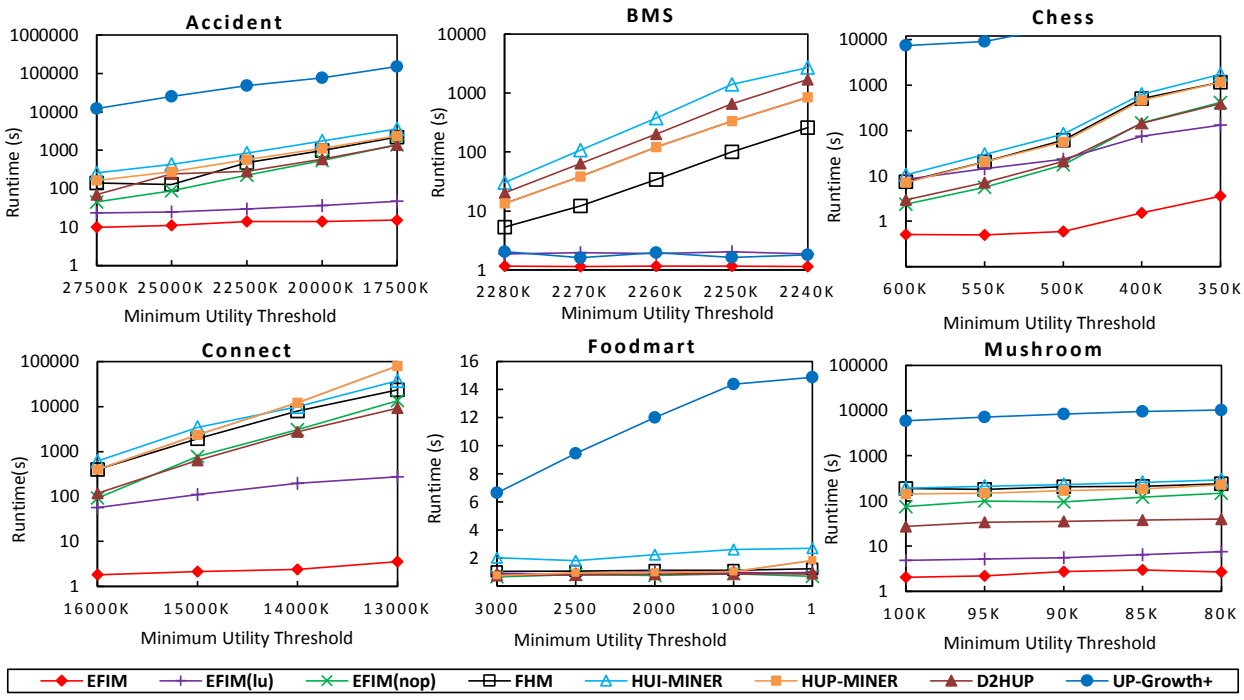

Figure II-2 Execution times on different datasets

Influence of the $minutil$ **threshold on memory usage.** In terms of memory usage, EFIM also clearly outperforms the other algorithms as shown in Table II-4. For *Accident*, *BMS*, *Chess*, *Connect*, *Foodmart* and *Mushroom*, EFIM uses 1.8, 4.4, 14.9, 27.0, 1.3 and 6.5 times less memory than the second fastest algorithm (d^{2}HUP). Moreover, EFIM uses 1.6, 9.2, 4.6, 8.1, 3.2 and 3.1 times less memory than the third fastest algorithm (FHM). It is also interesting that EFIM utilizes less than 100 MB on four out of the six datasets, and never more than 1 GB, while other algorithms often exceed 1 GB.

A reason why EFIM is so memory efficient is that it use a simple database representation, which does not requires to maintain much information in memory (only pointers for pseudo-projections). Other algorithms relies on more complex structures such as tree-structures (e.g. UPGrowth+) and list-structures (e.g. HUP-Miner, HUI-Miner and FHM), which requires additional memory for pointers between nodes and for maintaining additional information. For example, it can be eas-

ily seen that for any itemset α the size of the α projected database of EFIM is smaller than the size of the α utility-list of HUP-Miner, HUI-Miner and FHM. Both structures contains entries representing transactions where α occurs. However, EFIM stores two fields per entry (transaction id and pointer), while the utility list stores three (transaction id, *rutil* and *iutil* values). Moreover, a projected database generally contains less entries than the corresponding utility-list because of transaction merging. Another reason for the high memory efficiency of EFIM compared to utility list based algorithms is that the number of projected databases created by EFIM is less than the number of utility-lists, because EFIM visits less nodes of the search-enumeration tree (as we will show later). EFIM is also more efficient than two-phase algorithms such as UPGrowth+ since it is a one-phase algorithm (it does not need to maintain candidates in memory).

Lastly, another important characteristic of EFIM in terms of memory efficiency is that it reuses some of its data structures. As explained in section 4.5, EFIM uses a very efficient mechanism called Fast Array Counting for calculating upper-bounds. FAC only requires to create three arrays that are then reused to calculate the upper-bounds of each itemset considered during the depth-first search.

Table II-4 Comparison of maximum memory usage (MB)

Dataset	HUI-MINER	FHM	EFIM	UP-Growth+	HUP-Miner	d^{2}HUP
Accident	1,656	1,480	**895**	765	1,787	1,691
BMS	210	590	**64**	64	758	282
Chess	405	305	**65**	–	406	970
Connect	2,565	3,141	**385**	–	1,204	1,734
Foodmart	808	211	**64**	819	68	84
Mushroom	194	224	**71**	1,507	196	468

Influence of transaction merging on execution time. In terms of optimizations, the proposed transaction merging technique used in EFIM sometimes greatly increases its performance in terms of execution time. This allows EFIM to perform very well on dense datasets (*Chess*, *Connect* and *Mushroom*). For example, for *Connect* and $minutil = 13$M, EFIM terminates in 3 seconds while

HUP-Miner, d^{2}HUP, HUI-Miner and FHM respectively run for 22, 2, 10 and 6 hours. On dense datasets, transaction merging is very effective as projected transactions are more likely to be identical. This can be clearly seen by comparing the runtime of EFIM and EFIM(nop). On *Chess*, *Connect* and *Mushroom*, EFIM is up to 116, 3,790 and 55 and times faster than EFIM(nop). For other datasets, transaction merging also reduces execution times but by a lesser amount (EFIM is up to 90, 2 and 2 times than EFIM(nop) on *Accident*, *BMS* and *Foodmart*). It is also interesting to note that transaction merging cannot be implemented efficiently in utility-list based algorithms such as HUP-Miner, HUI-Miner and FHM, due to their vertical database representation, and also for hyperlink-based algorithms such as the d^{2}HUP algorithm.

Comparison of the number of visited nodes. We also performed an experiment to compare the ability at pruning the search space of EFIM to other algorithm. Table II-5 shows the number of nodes of the search-enumeration tree (itemsets) visited by EFIM, UP-Growth+, HUP-Miner, d^{2}HUP, HUI-Miner and FHM for the lowest $minutil$ values on the same datasets. It can be observed that EFIM is much more effective at pruning the search space than the other algorithms, thanks to its proposed sub-tree utility and local utility upper-bounds. For example, it can be observed that EFIM visits respectively up to 282, 636,000, 682,000, 6,000,800 and 658,000 times less nodes than UP-Growth+, HUP-Miner, d^{2}HUP, HUI-Miner and FHM.

Table II-5 Comparison of visited node count

Dataset	HUI-MINER	FHM	EFIM	UP-Growth+	HUP-Miner	d^{2}HUP
Accident	131,300	128,135	**51,883**	3,234,611	113,608	119,427
BMS	2,205,782,168	212,800,883	**323**	91,195	205,556,936	220,323,377
Chess	6,311,753	6,271,900	**2,875,166**	–	6,099,484	5,967,414
Connect	3,444,785	3,420,253	**1,366,893**	–	3,385,134	3,051,789
Foodmart	55,172,950	1,880,740	**233,231**	233,231	1,258,820	233,231
Mushroom	3,329,191	3,089,819	**2,453,683**	13,779,114	3,054,253	2,919,842

Influence of the number of transactions on execution time. Lastly, we also compared the execution time of EFIM with UP-Growth+, HUP-Miner, d^{2}HUP, HUI-Miner and FHM while vary-

ing the number of transactions in each dataset to assess the scalability of the algorithms. For this experiment, algorithms were run on the same datasets using the lowest $minutil$ values used in previous experiments, while varying the number of transactions from 25% to 100%. Results are shown in Fig. II-3. It can be observed that EFIM's runtime linearly increases w.r.t to the number of transactions, while runtimes of UP-Growth+, HUP-Miner, d^{2}HUP, HUI-Miner and FHM exponentially increase for some datasets. Thus, it can be concluded that EFIM has on overall the best scalability.The reason for this excellent scalability is that most operations performed by EFIM for an itemset α are performed in linear time. Thus, the complexity of EFIM is mostly linear for each node visited in the search space. Moreover, thanks to the proposed sub-tree utility and local-utility upper-bounds, EFIM can prune a large part of the search space.

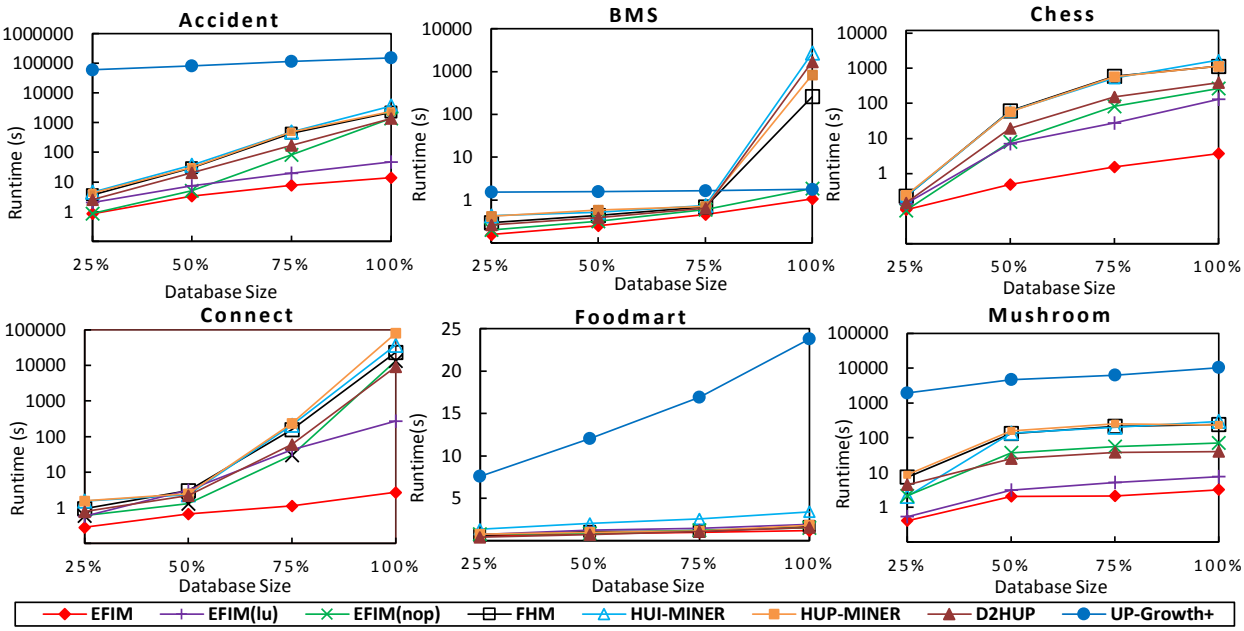

Figure II-3 Scalability on different datasets

6 Conclusion

High-utility itemset mining is an important data mining tasks with numerous applications. However, it remains very time consuming. To address this issue, we have presented a novel algorithm for high-utility itemset mining named EFIM. It relies on two new upper-bounds named *sub-tree utility* and *local utility* to prune the search space. It also introduces a novel array-based utility counting

approach named *Fast Utility Counting* to calculate these upper-bounds in linear time and space. Moreover, to reduce the cost of database scans, EFIM introduces techniques for database projection and transaction merging, also performed in linear time and space. An extensive experimental study on various datasets shows that EFIM is in general two to three orders of magnitude faster and consumes up to eight times less memory than the state-of-art algorithms UP-Growth+, HUP-Miner, d^2HUP, HUI-Miner and FHM. Moreover, results show that EFIM has excellent scalability and performs well on dense datasets. The source code of all algorithms and datasets used in the experiments can be downloaded at `http://goo.gl/9ay6Vw`.

For future work, we will extend EFIM for popular variations of the HUIM problem such as closed+ high-utility itemset mining [20] and top-k high-utility itemset [14].

CONCLUSION GÉNÉRALE

Le travail de recherche présenté dans cette thèse porte sur la découverte de motifs profitables. Deux articles ont été présentés, introduisant chacun un algorithme efficient pour la découverte de motifs profitables.

Le premier, HUSRM, s'applique à des bases de données contenant des séquences de transactions réalisées par des consommateurs. Il découvre des règles séquentielles profitables de la forme $X \rightarrow Y$ indiquant que si un consommateur achète un ensemble d'articles X, il achètera ensuite l'ensemble d'articles Y, avec une certaine probabilité (confiance). HUSRM est à notre connaissance le premier algorithme de découverte de motifs séquentiels profitables tenant compte de la confiance qu'un motif soit suivi. Afin de découvrir les règles séquentielles de façon efficiente, HUSRM associe une nouvelle structure de données appelée *compact utility-table* à chaque règle. Elle permet de calculer efficacement le profit, la fréquence, la confiance d'une règle et permet aussi d'élaguer l'espace de recherche. De plus, HUSRM inclut cinq optimisations. Une évaluation expérimentale avec plusieurs jeux de données montre que HUSRM est 25 fois plus rapide et consomme moins de mémoire qu'une version sans optimisations.

Le deuxième algorithme, EFIM, s'applique à un ensemble de transactions, non ordonnées dans le temps, et découvre des ensembles d'articles (*itemsets*) rapportant un profit élevé. Pour ce problème, plusieurs algorithmes existent [2,5,11,15,18,27,30]. La contribution de EFIM est de fournir un nouvel algorithme plus efficient en temps d'exécution et mémoire pour découvrir les itemsets profitables. Afin d'y parvenir, nous avons introduit plusieurs idées dans EFIM. Tout d'abord, pour réduire le coût de parcourir la base de données, l'algorithme effectue la fusion et la projection de transactions. Deuxièmement, pour élaguer une plus grande partie de l'espace de recherche, de nouvelles bornes supérieures nommées *subtree-utility* et *local-utility* ont été introduites. Troisièmement, une nouvelle structure de données nommée *utility-bins* a été introduite pour calculer ces bornes supérieures de façon efficiente. Une étude expérimentale avec plusieurs jeux de données a montré que EFIM est jusqu'à 7000 fois plus rapide et consomme moins de mémoire que les algorithmes FHM [5], d^{2}HUP [28], HUI-Miner [11] et HUP-Miner [12], souvent reconnus comme

étant les plus performants de la littérature.

Le travail présenté offre plusieurs possibilités pour des travaux futurs. Premièrement, un projet intéressant est d'utiliser les règles extraites par HUSRM pour concevoir un système de recommandation de produits visant à maximiser le profit plutôt que maximiser le nombre de recommandations menant à un achat. Deuxièmement, les deux algorithmes proposés pourraient être étendus pour tenir compte des articles vendus à perte [8] et des périodes de vente de chaque article [9].

RÉFÉRENCES

[1] Agrawal, R., Srikant, R.: Fast algorithms for mining association rules in large databases. Proc. Int. Conf. Very Large Databases, pp. 487–499 (1994)

[2] Ahmed, C. F., Tanbeer, S. K., Jeong, B.-S., Lee, Y.-K.: Efficient Tree Structures for High-utility Pattern Mining in Incremental Databases. IEEE Trans. Knowl. Data Eng. 21(12), 1708–1721 (2009)

[3] Fournier-Viger, P., Wu, C.-W., Tseng, V.S., Cao, L., Nkambou, R.: Mining Partially-Ordered Sequential Rules Common to Multiple Sequences. IEEE Trans. Knowl. Data Eng., 14 pages (to appear)

[4] Fournier-Viger, P., Gueniche, T., Zida, S., Tseng, V. S. (2014). ERMiner: Sequential Rule Mining using Equivalence Classes. Proc. 13th Intern. Symposium on Intelligent Data Analysis, Springer, LNCS 8819, pp. 108–119 (2014)

[5] Fournier-Viger, P., Wu, C.-W., Zida, S., Tseng, V. S.: FHM: Faster High-Utility Itemset Mining using Estimated Utility Co-occurrence Pruning. Proc. 21st Intern. Symp. Methodologies Intell. Systems, Springer, pp. 83–92 (2014)

[6] Jiawei Han, Jian Pei, Yiwen Yin, Runying Mao: Mining Frequent Patterns without Candidate Generation: A Frequent-Pattern Tree Approach. Data Min. Knowl. Discov. 8(1): 53-87 (2004)

[7] Fournier-Viger, P., Gomariz, A., Gueniche, T., Soltani, A., Wu., C., Tseng, V. S.: SPMF: a Java Open-Source Pattern Mining Library. Journal of Machine Learning Research. 15, 3389–3393 (2014)

[8] Fournier-Viger, P.: FHN: Efficient Mining of High-Utility Itemsets with Negative Unit Profits. Proc. 10th International Conference on Advanced Data Mining and Applications (ADMA 2014), Springer LNCS 8933, pp. 16–29 (2014)

[9] Fournier-Viger, P., Zida, S.: FOSHU: Faster On-Shelf High Utility Itemset Mining– with or without negative unit profit. Proc. 30th Symposium on Applied Computing (ACM SAC 2015). ACM Press, pp. 857-864 (2015)

[10] Lin, C.-W., Hong, T.-P., Lu., W.-H.: An effective tree structure for mining high utility itemsets. Expert Systems with Applications. 38(6), 7419–7424 (2011)

[11] Liu, M., Qu, J.:Mining High Utility Itemsets without Candidate Generation. Proc. 22nd ACM Intern. Conf. on Info. Know. Management, pp. 55–64 (2012)

[12] Krishnamoorthy, S.: Pruning strategies for mining high utility itemsets.Expert Systems with Applications, 42(5), 2371-2381 (2015)

[13] Pei, J., Han, J., Mortazavi-Asl, B., Wang, J., Pinto, H., Chen, Q., Dayal, U., Hsu, M.: Mining sequential patterns by pattern-growth: the PrefixSpan approach. IEEE Trans. Known. Data Engin. 16(11), 1424–1440 (2004)

[14] Tseng, V., Wu, C., Fournier-Viger, P., Yu, P.: Efficient Algorithms for Mining Top-K High Utility Itemsets. IEEE Trans. Knowl. Data Eng., 14 pages, to appear.

[15] Liu, Y., Liao, W., Choudhary, A.: A two-phase algorithm for fast discovery of high utility itemsets. Proc. 9th Pacific-Asia Conf. on Knowledge Discovery and Data Mining, pp. 689–695 (2005)

[16] Lo, D., Khoo, S.-C., Wong, L.. Non-redundant sequential rules - Theory and algorithm. Information Systems 34(4-5), 438-453 (2009)

[17] Pham, T. T., Luo, J., Hong, T. P., Vo, B.: An efficient method for mining non-redundant sequential rules using attributed prefix-trees. Engineering Applications of Artificial Intelligence 32, 88–99 (2014)

[18] Tseng, V. S., Wu, C. W., Shie, B. E., Yu, P. S.: UP-Growth: an efficient algorithm for high utility itemset mining. Proc. 16th ACM SIGKDD intern. conf. on Knowledge discovery and data mining, pp. 253–262 (2010).

[19] Tseng, V. S., Shie, B.-E., Wu, C.-W., Yu., P. S.: Efficient Algorithms for Mining High Utility Itemsets from Transactional Databases. IEEE Trans. Knowl. Data Eng. 25(8), 1772–1786 (2013)

[20] Tseng, V., Wu, C., Fournier-Viger, P., Yu, P.: Efficient Algorithms for Mining the Concise and Lossless Representation of Closed+ High Utility Itemsets. IEEE Trans. Knowl. Data Eng., 27(3), 726–739 (2015)

[21] Yin, J., Zheng, Z., Cao, L.: USpan: An Efficient Algorithm for Mining High Utility Sequential Patterns. Proc. 18th ACM SIGKDD Intern. Conf. on Knowledge Discovery and Data Mining, pp. 660–668 (2012)

[22] Yin, J., Zheng, Z., Cao, L., Song, Y., Wei, W.: Efficiently Mining Top-K High Utility Sequential Patterns. IEEE 13th International Conference on Data Mining, pp. 1259–1264 (2013)

[23] Han and Kamber (2011), Data Mining: Concepts and Techniques, 3rd edition, Morgan Kaufmann Publishers, ISBN 1-55860-901-6.

[24] Zaki, M. (2013). Data mining and analysis: fundamental concepts and algorithms.

[25] Tan, Steinbach and Kumar (2006), Introduction to Data Mining, Pearson education, ISBN-10: 0321321367.

[26] Witten, I. H. (Ian H.) (2011) Data mining : practical machine learning tools and techniques 3rd ed.

[27] Lan, G. C., Hong, T. P., Tseng, V. S.: An efficient projection-based indexing approach for mining high utility itemsets. Knowl. and Inform. Syst. 38(1), 85–107 (2014)

[28] Liu, J., Wang, K., Fung, B.: Direct discovery of high utility itemsets without candidate generation. Proc. 12th IEEE Intern. Conf. Data Mining, pp. 984–989 (2012)

[29] Rymon, R.: Search through systematic set enumeration. Proc. Third Intern. Conf. Principles of Knowledge Representation and Reasoning, pp. 539–50 (1992)

[30] Song, W., Liu, Y., Li, J.: BAHUI: Fast and Memory Efficient Mining of High Utility Itemsets Based on Bitmap. Intern. Journal of Data Warehousing and Mining. 10(1), 1–15 (2014)

[31] Uno, T., Kiyomi, M., Arimura, H.: LCM ver. 2: Efficient Mining Algorithms for Frequent/Closed/Maximal Itemsets. Proc. ICDM'04 Workshop on Frequent Itemset Mining Implementations, CEUR (2004)

[32] Zida, S., Fournier-Viger, P., Wu, C.-W., Lin, J. C. W., Tseng, V.S.: Efficient Mining of High Utility Sequential Rules. Proc. 11th Intern. Conf. Machine Learning and Data Mining. 15 pages (2015)

[33] Zida, S., Fournier-Viger, P., Lin, J. C.-W., Wu, C.-W., Tseng, V.S.: EFIM: A Highly Efficient Algorithm for High-Utility Itemset Mining. Proc. 14th Mexican Intern. Conference on Artificial Intelligence. 17 pages (2015)