A Flexible Approach to Finding Representative Pattern Sets

Guimei Liu, Haojun Zhang, and Limsoon Wong

Abstract—Frequent pattern mining often produces an enormous number of frequent patterns, which imposes a great challenge on visualizing, understanding and further analysis of the generated patterns. This calls for finding a small number of representative patterns to best approximate all other patterns. In this paper, we develop an algorithm called MinRPset to find a minimum representative pattern set with error guarantee. MinRPset produces the smallest solution that we can possibly have in practice under the given problem setting, and it takes a reasonable amount of time to finish when the number of frequent closed patterns is below one million. MinRPset is very space-consuming and time-consuming on some dense datasets when the number of frequent closed patterns is large. To solve this problem, we propose another algorithm called FlexRPset, which provides one extra parameter K to allow users to make a trade-off between result size and efficiency. We adopt an incremental approach to let the users make the trade-off conveniently. Our experiment results show that MinRPset and FlexRPset produce fewer representative patterns than RPlocal—an efficient algorithm that is developed for solving the same problem.

Index Terms—representative patterns, frequent pattern summarization

1 Introduction

Frequent pattern mining is an important problem in the data mining area. It was first introduced by Agrawal et al. in 1993 [1]. Frequent pattern mining is usually performed on a transaction database $D = \{t_1, t_2, ..., t_n\}$, where t_j is a transaction containing a set of items, $j \in [1, n]$. Let $I = \{i_1, i_2, ..., i_m\}$ be the set of distinct items appearing in D. A pattern X is a set of items in I, that is, $X \subseteq I$. If a transaction $t \in D$ contains all the items of a pattern X, then we say t supports X and t is a supporting transaction of X. Let $T(X)$ be the set of transactions in D supporting pattern X. The support of X, denoted as $supp(X)$, is defined as $|T(X)|$. If the support of a pattern X is larger than a user-specified threshold min_sup, then X is called a frequent pattern. Given a transaction database D and a minimum support threshold min_sup, the task of frequent pattern mining is to find all the frequent patterns in D with respect to min_sup.

Many efficient algorithms have been developed for mining frequent patterns [2]. Now the focus has shifted from how to efficiently mine frequent patterns to how to effectively utilize them. Frequent patterns have the anti-monotone property: if a pattern is frequent, then all of its subsets must be frequent too. On dense datasets and/or when the minimum support is low, long patterns can be frequent. All the subsets of these frequent long patterns are frequent too based on the anti-monotone property. This leads to an explosion in the number of frequent patterns. The huge quantity of patterns can easily become a bottleneck for understanding and further analyzing frequent patterns.

It has been observed that the complete set of frequent patterns often contains a lot of redundancy. Many frequent patterns have similar items and supporting transactions. It is desirable to group similar patterns together and represent them using one single pattern. The concept of frequent closed pattern is proposed for this purpose [3]. Let X be a pattern and S be the set of patterns appearing in the same set of transactions as X, that is, $S = \{Y | T(Y) = T(X)\}$. The longest pattern in S is called a closed pattern, and all the other patterns in S are subsets of it. The closed pattern of S is selected to represent all the patterns in S. The set of frequent closed patterns is a lossless representation of the complete set of frequent patterns. That is, all the frequent patterns and their exact support can be recovered from the set of frequent closed patterns. The number of frequent closed patterns can be much smaller than the total number of frequent patterns, but it can still be tens of thousands or even more.

Frequent closed patterns group patterns supported by exactly the same set of transactions together. This condition is too restrictive. Xin et al. [4] relax this condition to further reduce pattern set size. They propose the concept of δ-covered to generalize the concept of frequent closed pattern. A pattern X_1 is δ-covered by another pattern X_2 if X_1 is a subset of X_2 and $(supp(X_1) - supp(X_2))/supp(X_1) \le \delta$. The goal is to find a minimum set of representative patterns that can δ-cover all frequent patterns. When δ=0, the problem corresponds to finding all frequent closed patterns.

• *Guimei Liu, Haojun Zhang and Limsoon Wong are with the Department of Computer Science, National University of Singapore, Singapore.*
E-mail: {liugm, zhanghao, wongls}@comp.nus.edu.sg

Xin et al. show that the problem can be mapped to a set cover problem. They develop two algorithms, RPglobal and RPlocal, to solve the problem. RPglobal first generates the set of patterns that can be δ-covered by each pattern, and then employs the well-known greedy algorithm [5] for the set cover problem to find representative patterns. The optimality of RPglobal is determined by the optimality of the greedy algorithm, so the solution produced by RPglobal is almost the best solution we can possibly have in practice. However, RPglobal is very time-consuming and space-consuming. It is feasible only when the number of frequent patterns is not large. RPlocal is developed based on FPclose [6]. It integrates frequent pattern mining with representative pattern finding. RPlocal is very efficient, but it produces more representative patterns than RPglobal.

In this paper, we analyze the bottlenecks for finding a minimum representative pattern set and develop two algorithms, MinRPset and FlexRPset, to solve the problem. Algorithm MinRPset is similar to RPglobal, but it utilizes several techniques to reduce running time and memory usage. In particular, MinRPset uses a tree structure called CFP-tree [7] to store frequent patterns compactly. The CFP-tree structure also supports efficient retrieval of patterns that are δ-covered by a given pattern. Our experiment results show that MinRPset is only several times slower than RPlocal in most of the cases, while RPglobal is often several orders of magnitude slower than RPlocal. Algorithm FlexRPset is developed based on MinRPset. It provides one extra parameter K, which allows users to make a trade-off between efficiency and the number of representative patterns selected. When $K = \infty$, FlexRPset approaches MinRPset. With the decrease of K, FlexRPset becomes faster, but it produces more representative patterns. When K=1, FlexRPset is often faster than RPlocal, and it still produces fewer representative patterns than RPlocal in almost all cases. We use an incremental approach to allow users to make a trade-off between running time and result size conveniently.

In MinRPset and FlexRPset, a representative pattern can represent its subsets only. To further reduce the number of representative patterns, we drop this condition to allow a representative pattern to represent more patterns. Our experiment results show that this relaxation can further reduce the number of representative patterns greatly.

The rest of the paper is organized as follows. Section 2 introduces related work. Section 3 gives the formal problem definition. The two algorithms, MinRPset and FlexRPset, are described in Section 4 and Section 5 respectively. The relaxation of the covered condition is discussed in Section 6. Experiment results are reported in Section 7. Finally, Section 8 concludes the paper.

2 Related Work

The number of frequent patterns can be very large. Besides frequent closed patterns, several other concepts, such as generators [8], disjunction-free generators [9], δ-free sets [10], non-derivable patterns [11], maximal patterns [12], top-k frequent closed patterns [13] and redundancy-aware top-k patterns [14], have been proposed to reduce pattern set size. The number of generators is larger than that of closed patterns. Furthermore, the set of generators itself is not lossless. It requires a border to be lossless [9], so does the set of disjunction-free generators and δ-free sets. The number of non-derivable patterns can also be larger than that of closed patterns on some datasets. The number of maximal patterns is much smaller than the number of closed patterns. All frequent patterns can be recovered from maximal patterns, but their support information is lost. Another work that also ignores the support information is [15]. It selects k patterns that best cover a collection of patterns.

Frequent closed patterns preserve the exact support of all frequent patterns. In many applications, knowing the approximate support of frequent patterns is sufficient. Several approaches have been proposed to make a trade-off between pattern set size and the precision of pattern support. The work by Xin et al. [4] described in Section 1 is one such approach. Another approach proposed by Pei et al. [16] uses absolute error bound. It uses heuristic algorithms to mine a minimal condensed pattern-base, which is a superset of the maximal pattern set. All frequent patterns and their support can be restored from a condensed pattern-base with error guarantee.

Yan et al. [17] use profiles to summarize patterns. A profile consists of a master pattern, a support and a probability distribution vector, which contains the probability of the items in the master pattern. The set of patterns represented by a profile are subsets of the master pattern, and their support is calculated by multiplying the support of the profile and the probability of the corresponding items. To summarize a collection of patterns using k profiles, Yan et al. partition the patterns into k clusters, and use a profile to describe each cluster. There are several drawbacks with this profile-based approach: 1) It makes contradictory assumptions. On one hand, the patterns represented by the same profile are supposed to be similar in both item composition and supporting transactions, thus the items in the same profile are expected to be strongly correlated. On the other hand, based on how the support of patterns are calculated from a profile, the items in the same profile are expected to be independent. It is hard to make a balance between the two contradicting requirements. 2) There is no error guarantee on the estimated support of patterns. 3) The proposed algorithm for generating profiles is very slow because it needs to scan the original dataset repeatedly. 4) The boundary between frequent patterns and infrequent patterns cannot be determined using profiles.

Several improvements have been made to the profile-based approach. Jin et al. [18] have developed a regression-based approach to minimize restoration error. They cluster patterns based on restoration errors instead of similarity between patterns, thus their approach can achieve lower restoration error. However, there is still no error guarantee on the restored support. CP-summary [19] uses conditional independence to reduce restoration error. It adds one more component to each profile: a pattern base, and the new profile is called c-profile. The items in a c-profile are expected to be independent with respect to the pattern base. CP-summary provides error guarantee on estimated support. However, patterns of a c-profile often share little similarity, so a c-profile is not representative of its patterns any more.

Profiles can be considered as generalizations of closed patterns. Wang et al. [20] make generalization on another concise representation of frequent patterns—non-derivable patterns. They use Markov Random Field (MRF) to summarize frequent patterns. The support of a pattern is estimated from its subsets, which is similar to non-derivable patterns. Markov Random Field model is not as intuitive as profiles, and it is also expensive to learn. It does not provide error guarantee on estimated support either.

The above approaches aim to summarize frequent patterns. Mampaey et al. [21] aim to summarize data instead with a collection of non-redundant patterns. A probabilistic maximum entropy model is used in their approach.

3 Problem Statement

We follow the problem definition in [4]. The distance between two patterns is defined based on their supporting transaction sets.

Definition 1 ($D(X_1, X_2)$): Given two patterns X_1 and X_2, the distance between them is defined as $D(X_1, X_2) = 1 - \frac{|T(X_1) \cap T(X_2)|}{|T(X_1) \cup T(X_2)|}$.

Definition 2 (ϵ-covered): Given a real number $\epsilon \in [0, 1)$ and two patterns X_1 and X_2, we say X_1 is ϵ-covered by X_2 if $X_1 \subseteq X_2$ and $D(X_1, X_2) \leq \epsilon$.

In the above definition, condition $X_1 \subseteq X_2$ ensures that the two patterns have similar items, and condition $D(X_1, X_2) \leq \epsilon$ ensures that the two patterns have similar supporting transaction sets and similar support. Based on the definition, a pattern ϵ-covers itself.

Lemma 1: Given two patterns X_1 and X_2, if pattern X_1 is ϵ-covered by pattern X_2 and we use $supp(X_2)$ to approximate $supp(X_1)$, then the relative error $\frac{supp(X_1) - supp(X_2)}{supp(X_1)}$ is no larger than ϵ.

Proof: $\frac{supp(X_1) - supp(X_2)}{supp(X_1)} = 1 - \frac{supp(X_2)}{supp(X_1)} = 1 - \frac{|T(X_2)|}{|T(X_1)|} \leq 1 - \frac{|T(X_1) \cap T(X_2)|}{|T(X_1) \cup T(X_2)|} \leq \epsilon$. □

Lemma 2: If a frequent pattern X_1 is ϵ-covered by pattern X_2, then $supp(X_2) \geq min\_sup \cdot (1 - \epsilon)$.

Proof: Based on Lemma 1, $1 - \frac{supp(X_2)}{supp(X_1)} \leq \epsilon$, so we have $supp(X_2) \geq supp(X_1) \cdot (1 - \epsilon) \geq min\_sup \cdot (1 - \epsilon)$. □

Our goal here is to select a minimum set of patterns that can ϵ-cover all frequent patterns. The selected patterns are called representative patterns. We do not require representative patterns to be frequent. Based on Lemma 2, the support of representative patterns must be no less than $min\_sup \cdot (1 - \epsilon)$.

When ϵ=0, the representative patterns are frequent closed patterns. Restoring the support of a non-representative pattern is the same as restoring the support of a non-closed pattern. That is, to get the support of a non-representative pattern X, we search the supersets of X in the representative pattern set, and then use the highest support of the supersets to approximate $supp(X)$. Based on Lemma 1, the restoration error is bounded by ϵ.

In the next two sections, we describe two algorithms to find representative patterns.

4 The MinRPset Algorithm

Let $\mathcal{F}$ be the set of frequent patterns in a dataset D with respect to threshold min_sup, and $\hat{\mathcal{F}}$ be the set of patterns with support no less than $min\_sup \cdot (1 - \epsilon)$ in D. Obviously, $\mathcal{F} \subseteq \hat{\mathcal{F}}$. Given a pattern $X \in \hat{\mathcal{F}}$, we use $C(X)$ to denote the set of frequent patterns that can be ϵ-covered by X. We have $C(X) \subseteq F$. If X is frequent, we have $X \in C(X)$.

A straightforward algorithm for finding a minimum representative pattern set works as follows. First we mine all patterns in $\hat{F}$, and then we generate $C(X)$—the set of frequent patterns that X covers—for every pattern $X \in \hat{\mathcal{F}}$. We get $|\hat{\mathcal{F}}|$ sets. The elements of these sets are frequent patterns in $\mathcal{F}$. Let $S = \{C(X) | X \in \hat{\mathcal{F}}\}$. Finding a minimum representative pattern set is now equivalent to finding a minimum number of sets in S that can cover all the frequent patterns in $\mathcal{F}$. This is a set cover problem, and it is NP-hard. We use the well-known greedy algorithm [5] to solve the problem, which achieves an approximation ratio of $\sum_{i=1}^{k} \frac{1}{i}$, where k is the maximal size of the sets in S. We call this simple algorithm MinRPset.

The greedy algorithm is essentially the best-possible polynomial time approximation algorithm for the set cover problem. Our experiment results have shown that it usually takes little time to finish. Generating $C(X)$s is the main bottleneck of the MinRPset algorithm when $\mathcal{F}$ and $\hat{\mathcal{F}}$ are large because we need to find $C(X)$s over a large $\mathcal{F}$ for a large number of patterns in $\hat{\mathcal{F}}$. We use the following techniques to improve the efficiency of MinRPset: 1) consider closed patterns only; 2) use a structure called CFP-tree to find $C(X)$s efficiently; and 3) use a light-weight compression technique to compress $C(X)$s.

4.1 Considering closed patterns only

A pattern is closed if it is more frequent than all of its supersets. If a pattern X_1 is non-closed, then there exists another pattern X_2 such that $X_1 \subset X_2$ and $supp(X_2) = supp(X_1)$.

Lemma 3: Given two patterns X_1 and X_2 such that $X_1 \subseteq X_2$ and $supp(X_1) = supp(X_2)$, if X_2 is ϵ-covered by a pattern X, then X_1 must be ϵ-covered by X too. The above lemma directly follows from Definition 2. It implies that instead of covering all frequent patterns, we can cover frequent closed patterns only, which leads to the following lemma.

Lemma 4: Let $\mathcal{F}$ be the set of frequent patterns in a dataset D with respect to a threshold min_sup. If a set of patterns R ϵ-covers all the frequent closed patterns in $\mathcal{F}$, then R ϵ-covers all the frequent patterns in $\mathcal{F}$.

Lemma 5: Given two patterns X_1 and X_2 such that $X_1 \subseteq X_2$ and $supp(X_1) = supp(X_2)$, if a pattern X is ϵ-covered by X_1, then X must be ϵ-covered by X_2 too.
This lemma also directly follows from Definition 2. It suggests that we can use closed patterns only to cover all frequent patterns.

The number of frequent closed patterns can be orders of magnitude smaller than the total number of frequent patterns. Consider only closed patterns improves the efficiency of the MinRPset algorithm in two aspects. On one hand, it reduces the size of individual $C(X)$s since now they contain only frequent closed patterns. On the other hand, it reduces the number of patterns whose $C(X)$ needs to be generated as now we need to generate $C(X)$s for closed patterns only.

4.2 Using CFP-tree to find $C(X)$s efficiently

The CFP-tree structure is specially designed for storing and querying frequent patterns [7]. It resembles a set-enumeration tree [22]. We use an example dataset D in Table 1 to illuminate its structure. Table 2 shows all the frequent patterns in D when $min\_sup = 3$. The CFP-tree constructed from these frequent patterns is shown in Figure 1. The CFP-tree is constructed using a pattern-growth approach. The root node contains all frequent items sorted in ascending frequency order. Each item i in the root node points to a subtree, and this subtree stores all the frequent patterns discovered from item i's conditional database.

Each node in a CFP-tree is a variable-length array. If a node contains multiple entries, then each entry contains exactly one item. If a node has only one entry, then it is called a *singleton node*. Singleton nodes can contain more than one item. For example, node 2 in Figure 1 is a singleton node with two items m and a. An entry E stores several pieces of information: (1) m items ($m \geq 1$), (2) the support of E, (3) a pointer pointing to the child node of E and (4) the id of the entry that is assigned using preordering. In the rest

TABLE 1
An example dataset D

TID	Transactions
1	a, c, e, f, m, p
2	b, e, v
3	a, b, f, m, p
4	d, e, f, h, p
5	a, c, d, m, v
6	a, c, h, m, s
7	a, f, m, p, u
8	a, b, d, f, g

TABLE 2
Frequent patterns (min_sup=3)

ID	Itemsets	ID	itemsets	ID	itemsets
1	a:6	9	ac:3	17	acm:3
2	b:3	10	af:4	18	afm:3
3	c:3	11	am:5	19	afp:3
4	d:3	12	ap:3	20	amp:3
5	e:3	13	cm:3	21	fmp:3
6	f:5	14	fm:3	22	afmp:3
7	m:5	15	fp:4		
8	p:4	16	mp:3		

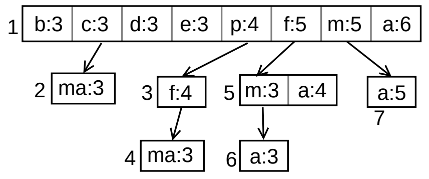

Fig. 1. CFP-tree constructed on the frequent patterns in Table 2

of this paper, we use $E.items$, $E.support$, $E.child$ and $E.preorder$ to denote the above fields.

The CFP-tree structure allows different patterns to share the storage of their prefixes as well as suffixes. Prefix sharing is easy to understand. For example, patterns $\{f, m\}$ and $\{f, a\}$ share the same prefix $\{f\}$ in Figure 1. In a multi-entry CFP-tree node, only the items after an entry E can appear in the subtree pointed by E, and these items are called *candidate extensions* of E. Suffix sharing occurs when a candidate extension of an entry E occurs in the same set of transactions as the pattern represented by E. Let i be a candidate extension of E and X be a pattern represented by E. If $supp(X) = supp(X \cup \{i\})$, then for any pattern Z, we must have $supp(X \cup Z) = supp(X \cup \{i\} \cup Z)$. In other words, X and $X \cup \{i\}$ have the same extensions. In CFP-tree, a singleton node containing item i is created to enable the sharing between X and $X \cup \{i\}$. In the root node of Figure 1, item f is a candidate extension of item p. Patterns $\{p\}$ and $\{p, f\}$ have the same support, so a singleton node containing item f is created, which is node 3, to allow $\{p\}$ and $\{p, f\}$ to share the same subtree.

Every entry in a CFP-tree represents one or more patterns with the same support, and these patterns contain the items on the path from the root to the entry. Items contained in singleton nodes are optional.

Let E be an entry, X_m be the set of items in the multiple-entry nodes and X_s be the set of items in the singleton nodes on the path from the root to the parent of E respectively. The set of patterns represented by E is $\{X_m \cup Y \cup Z | Y \subseteq X_s, Z \subseteq E.items, Z \neq \emptyset\}$. The longest pattern represented by E is $X_m \cup X_s \cup E.items$. Let us look at an example. In Figure 1, node 4 contains only one entry. For this entry, we have $X_m = \{p\}$, $X_s = \{f\}$ and $E.items = \{m, a\}$. Hence node 4 represents 6 patterns: $\{p, m\}$, $\{p, a\}$, $\{p, m, a\}$, $\{p, f, m\}$, $\{p, f, a\}$ and $\{p, f, m, a\}$. We use $E.pattern$ to denote the longest pattern represented by E.

The above feature makes CFP-tree a very compact structure for storing frequent patterns. The number of entries in a CFP-tree is much smaller than the total number of patterns stored in the tree. For each entry, we consider its longest pattern only based on Lemma 3 and Lemma 5. For an entry E, only its longest pattern can be closed. Other patterns of E that are shorter than the longest pattern cannot be closed based on the definition of closed patterns. If the longest pattern of an entry is not closed, then we call the entry a *non-closed entry*.

The CFP-tree structure has the following property.

Property 1: In a multiple-entry node, the item of an entry E can appear in the subtrees pointed by entries before E, but it cannot appear in the subtrees pointed by entries after E.

For example, in the root node of Figure 1, item p is allowed to appear in the subtrees pointed by entries b, c, d and e, but it is not allowed to appear in the subtrees pointed by entries f, m and a. This property implies the following lemma.

Lemma 6: In a CFP-tree, the supersets of a pattern cannot appear on the right of the pattern. They appear either on the left of the pattern or in the subtree pointed by the pattern.

4.2.1 Finding one $C(X)$

Given a pattern X, $C(X)$ contains the subsets of X that can be ϵ-covered by X. CFP-tree supports efficient retrieval of subsets of patterns. To find the subsets of a pattern X, we simply traverse the CFP-tree and match the items of the entries against X. For an entry E in a multiple-entry node, if its item appears in X, then entry E represents some subsets of X and the search is continued on its subtree. Otherwise, entry E and its subtree is skipped because all the patterns in the subtree of E contain $E.items \notin X$, and these patterns cannot be subsets of X. For example, in the root node of Figure 1, to search for the subsets of $\{c, m, a\}$, we need to visit the subtrees pointed by c, m and a only.

Algorithm 1 shows the pseudo-codes for retrieving $C(X)$. Initially, $cnode$ is the root node of the CFP-tree. Parameter Y contains the set of items to be searched in $cnode$. It is set to X initially. Once an entry E is visited, the item of E is removed from Y when Y is passed to the subtree of E (line 8, 18). The item of E is also excluded when Y is passed to the entries after E (line 21). This is because the item of E cannot appear in the subtrees pointed by entries after E based on Property 1.

Algorithm 1 Search_CX Algorithm

```
Input:
    $cnode$ is a CFP-tree node; //$cnode$ is the root node initially.
    $Y$ is the set of items to be searched in $cnode$; //$Y$=$X$ initially.
    $supp(X)$ is the support of $X$;
Output:
    $C(X)$;
Description:
 1: if $cnode$ contains only one entry $E$ then
 2:    if $E.support == supp(X)$ AND $E.pattern \subset X$ AND $E$ is on the right of $X$ then
 3:       Mark $E$ as non-closed;
 4:    if $E$ is not marked as non-closed AND $E$ is frequent then
 5:       if $E.items \bigcap Y \neq \emptyset$ AND $E.support \leq \frac{supp(X)}{(1-\epsilon)}$ then
 6:          Put $E.preorder$ into $C(X)$;
 7:       if $E.child \neq NULL$ AND $Y - E.items \neq \emptyset$ then
 8:          Search_CX($E.child$, $Y - E.items$, $supp(X)$);
 9: else if $cnode$ contains multiple entries then
10:    for each entry $E \in cnode$ from left to right do
11:       if $E.items \in Y$ AND $E$ is frequent then
12:          if $E.support == supp(X)$ AND $E.pattern \subset X$ AND $E$ is on the right of $X$ then
13:             Mark $E$ as non-closed;
14:          if $E$ is not marked as non-closed then
15:             if $E.support \leq \frac{supp(X)}{(1-\epsilon)}$ AND $E$ is more frequent than its child entries then
16:                Put $E.preorder$ into $C(X)$;
17:             if $E.child \neq NULL$ AND $Y - E.items \neq \emptyset$ then
18:                Search_CX($E.child$, $Y - E.items$, $supp(X)$);
19:             if $supp(E.pattern \cup Y) > \frac{supp(X)}{(1-\epsilon)}$ then
20:                return ;
21:             $Y$=$Y - E.items$;
```

During the search of $C(X)$s, we also mark non-closed patterns. If the longest pattern of E is a proper subset of X, $E.support$=$supp(X)$ and E occurs on the right of X, then E is marked as non-closed (line 2-3, 12-13), and it is skipped in subsequent search. For example, when we search for the subsets of pattern $\{p, f, m, a\}$ using Algorithm 1 in Figure 1, we find that one subset $\{f, m\}$ has the same support as $\{p, f, m, a\}$ and $\{f, m\}$ occurs on the right of $\{p, f, m, a\}$. Hence the entry of $\{f, m\}$, which is the first entry in node 5, is marked as non-closed. All the patterns in the subtree pointed by this entry cannot be closed either because for every pattern Z in the subtree, $Z' = Z \cup \{p, a\}$ is a proper superset of Z and it has the same support as Z. Entry $\{f, m\}$ and its subtree are skipped in subsequent traversal.

The *early termination* technique. If a pattern is ϵ-covered by X, then its support must be no larger than $\frac{supp(X)}{(1-\epsilon)}$ based on Definition 2. We use this requirement to further improve the efficiency of Algorithm 1.

Given an entry E in a multiple-entry node, after we visit the subtree of E, if we find $supp(E.pattern \cup Y) > \frac{supp(X)}{(1-\epsilon)}$, where Y is the set of items that is passed to E, then there is no need to visit the subtrees pointed by entries after E (line 19-20). The reason being that all the subsets of X in these subtrees must be subsets of $(E.pattern \cup Y)$, and their support must be larger than $\frac{supp(X)}{(1-\epsilon)}$ too based on the anti-monotone property. We call this pruning technique *early termination*.

Let us look at an example of early termination. Let ϵ=0.25 and $X = \{p, f, m, a\}$. In the root node of Figure 1, only the last four entries and their subtrees need to be visited to get $C(X)$. At entry m, we have $E.pattern = \{m\}$ and $Y = \{m, a\}$. After visiting the subtree pointed by m, we get $supp(\{m, a\})$=5, which is larger than $\frac{supp(X)}{(1-\epsilon)} = \frac{3}{1-0.25} = 4$. Therefore, $\{m, a\}$ cannot be covered by $X = \{p, f, m, a\}$ because its support is too high. The subsets of X that occur on the right of $\{m, a\}$ cannot be covered by X either as they are subsets of $\{m, a\}$, and their support must be no less than 5. Hence entry a can be safely skipped.

4.2.2 Finding $C(X)$s of all closed patterns

Algorithm 2 shows the pseudo-codes for generating all $C(X)$s. It traverses the CFP-tree in depth-first order from left to right. Using this traversal order, the supersets of a pattern X that are on the left of X are visited before X. If the support of X is the same as one of these supersets, then X should be marked as non-closed when $Search\_CX$ is called for that superset. If X is not marked as non-closed when X is visited, it means that X is more frequent than all its supersets on its left. Based on Lemma 6, the supersets of a pattern appear either on the left of the pattern or in the subtree pointed by the pattern. If X is also more frequent than its child entries, then X must be closed. The conditions listed at line 2 and line 3 ensure that Algorithm 2 generates $C(X)$s for only closed patterns.

Algorithm 2 DFS_Search_CXs Algorithm

```
Input:
    $cnode$ is a CFP-tree node; //$cnode$ is the root node initially.
Output:
    $C(X)$s;
Description:
1: for each entry $E \in cnode$ from left to right do
2:     if $E$ is not marked as non-closed then
3:         if $E$ is more frequent than its child entries then
4:             $X$=$E.pattern$;
5:             $C(X)$ = Search_CX($root$, $X$, $E.support$);
6:         if $E.child \neq NULL$ then
7:             DFS_Search_CXs($E.child$);
```

In Algorithm 2, if an entry E is marked as non-closed because it has the same support as one of its supersets on its left, then all the patterns in the subtree pointed by E are non-closed. We can safely skip E and its subtree in subsequent traversal (line 2). The same pruning is done in Algorithm 1 (line 4, 14). This pruning technique has been used in almost all frequent closed pattern mining algorithms to prune non-closed patterns [6], [23].

4.3 Compressing $C(X)$s

In a CFP-tree, each entry E has an id, and it is denoted as $E.preorder$. In Algorithm 1, we put the id of entry E into $C(X)$ if E is ϵ-covered by X (line 6, 16). Each id takes 4 bytes. The total number of $C(X)$s generated by Algorithm 2 grows with the number of frequent (closed) patterns. When the number of frequent closed patterns is large, the total size of $C(X)$s can be very large. If the main memory cannot accommodate all $C(X)$s, the greedy set cover algorithm becomes very slow.

To alleviate this problem, we compress $C(X)$s using a light-weight compression technique [24]. Each entry id occupies one or more bytes depending on its value. To reduce the number of bytes needed for storing entries ids, we sort the entry ids in ascending order and store the differences between consecutive ids instead. Our experiment results show that this compression technique can reduce the space needed for storing $C(X)$s by about three quarters in many cases.

Algorithm 3 MinRPset Algorithm

```
Description:
1: Mine patterns with support $\geq min\_sup \cdot (1-\epsilon)$ and store
   them in a CFP-tree; let $root$ be the root node of the tree;
2: DFS_Search_CXs($root$);
3: Remove non-closed entries from $C(X)$s;
4: Apply the greedy set cover algorithm on $C(X)$s to find
   representative patterns and output them;
```

Algorithm 3 shows the pseudo-codes of MinRPset, and it calls Algorithm 2 to find $C(X)$s. Note that we store all patterns, including non-closed patterns, with support no less than $min\_sup \cdot (1-\epsilon)$ in a CFP-tree (line 1). Non-closed patterns are identified during the search of $C(X)$s. Hence it is possible that some $C(X)$s contains some non-closed entries. These non-closed entries are removed from $C(X)$s (line 3) before the greedy set cover algorithm is applied.

4.4 Complexity Analysis

Let $\hat{F}$ be the set of patterns with support no less than $min\_sup \cdot (1-\epsilon)$. The first step of Algorithm 3 constructs the CFP-tree storing all patterns in $\hat{F}$, and its complexity is the same as that of a depth-first frequent pattern mining algorithm since the CFP-tree is constructed using a depth-first frequent pattern mining algorithm [7].

Let $\hat{F_c} = \{X | X \in \hat{F} \wedge X\ is\ closed\}$. The second step generates $C(X)$ for every pattern $X \in \hat{F_c}$ by calling Algorithm 1. In the worst case, Algorithm 1 needs to check all the $2^{|X|}-1$ subsets of X to see whether they are ϵ-covered by X. Hence the worst case time complexity of the second step is $O(\sum_{X\in\hat{F_c}}(2^{|X|}-1))$. In practice, the cost of this step is much lower than the worst case cost because the early termination technique can prune many of the subsets, especially when ϵ is small. Furthermore, we consider closed subsets of X only, which also reduces the number of subsets to be checked.

The third step removes non-closed entries from $C(X)$s. This can be done by scanning all the $C(X)$s once. Hence, the time complexity of the third step is $O(\sum_{X\in\hat{F_c}}|C(X)|)$. The fourth step is the greedy set cover algorithm, which can be implemented in time complexity of $O(\sum_{X\in\hat{F_c}}|C(X)|)$ too.

The cost of the second step is higher than that of the last two steps because the size of $C(X)$ is usually much smaller than $2^{|X|}-1$. In the next section, we describe an algorithm, which aims to reduce the cost and the output size of the second step. The cost of the last two steps is reduced consequently too.

5 The FlexRPset Algorithm

When the number of frequent patterns is large on a dataset, the MinRPset algorithm may become very slow since it needs to search subsets over a large CFP-tree for a large number of patterns. Furthermore, the set of $C(X)$s may become too large to fit into the main memory. To solve this problem, instead of searching $C(X)$s for all closed patterns, we can selectively generate $C(X)$s such that every frequent pattern is covered a sufficient number of times, in the hope that the greedy set cover algorithm can still find a near-optimal solution. Intuitively, the fewer the number of $C(X)$s generated, the more efficient the algorithm is. This is the basic idea of the FlexRPset algorithm.

The FlexRPset algorithm uses a parameter K to control the minimum number of times that a frequent pattern needs to be covered. Algorithm 4 shows how FlexRPset selectively generates $C(X)s$. The other steps of FlexRPset are the same as those of MinRPset.

Algorithm 4 still uses the depth-first order to traverse a CFP-tree from left to right. It traverses the subtree of an entry E first (line 3-4) before it processes E (line 5-8), which means that when E is processed, all the supersets of E have been processed already based on Lemma 6, and E cannot be covered any more except by itself. If E is frequent and it is covered less than K times, then we generate $C(E.pattern)$ to cover E (the first condition at line 6). If E has already be covered at least K times when E is visited, then we look at the ancestor entries of E. For an ancestor entry E' of E, most of its supersets are already processed too when E is visited, hence not many remaining entries can cover E'. If E' is frequent, E' can be ϵ-covered by E and E' is covered less than K times,

Algorithm 4 Flex_Search_CXs Algorithm

```
Input:
    $cnode$ is a CFP-tree node; //$cnode$ is the root node initially.
    $K$ is the minimum number of times that a frequent closed pattern needs to be covered;
Output:
    $C(X)$s;
Description:
1: for each entry $E \in cnode$ from left to right do
2:     if $E$ is not marked as non-closed then
3:         if $E.child \neq NULL$ then
4:             Flex_Search_CXs($E.child$);
5:         if $E$ is more frequent than its child entries then
6:             if ($E$ is frequent AND $E$ is covered less than $K$ times) OR ($\exists$ an ancestor entry $E'$ of $E$ such that $E'$ is frequent, $E'$ can be $\epsilon$-covered by $E$ and $E'$ is covered less than $K$ times) then
7:                 $X$=$E.pattern$;
8:                 $C(X)$ = Search_CX($root$, $X$, $E.support$);
```

then we also generate $C(E.pattern)$ to cover E' (the second condition at line 6).

We use the CFP-tree shown in Figure 1 to illustrate how Algorithm 4 works. Let ϵ=0.25 and K=1. Algorithm 4 traverses the CFP-tree in post-order from left to right. The first entry b in the root node is first visited. Since it has not be covered, so $C(\{b\})$ is generated and it contains only pattern $\{b\}$ itself. Next, node 2 is visited, and it contains only one entry. This entry has not be covered, so $C(\{c,m,a\})$ is generated using Algorithm 1 and it contains only pattern $\{c,m,a\}$ itself. Among other subsets of $\{c,m,a\}$, pattern $\{c\}$ is excluded because it has the same support as its child entry and it is not closed; pattern $\{m\}$ is excluded for the same reason; pattern $\{m,a\}$ and $\{a\}$ cannot be covered by $\{c,m,a\}$ because their support is too high. Next, the second entry in the root node is visited. This entry has the same support as its child node, so it is skipped (line 5). Next, $C(\{d\})$ and $C(\{e\})$ are generated and both of them contain only the pattern itself. Then node 4 is visited. It contains only one entry and this entry has not be covered before. $C(\{p,f,m,a\})$ is generated, and it contains $\{p,f\}$ and $\{f,a\}$. At the same time, the first entry in node 5 is marked as non-closed. Next node 3 is visited. Node 3 has already been covered by $\{p,f,m,a\}$ and the only ancestor entry of node 3 is not closed, so node 3 is skipped. In the remaining traversal, $C(\{f,a\})$ is generated in order to cover $\{f\}$ and $C(\{m,a\})$ is generated to cover itself and pattern $\{a\}$.

With the increase of parameter K, more information are gathered, hence less representative patterns are generated. However, the running time of FlexRPset becomes longer. How can users make a trade-off between running time and result size conveniently? We observe that the $C(X)$s generated at a smaller K value can be re-used at a larger K value. This leads to the incremental FlexRPset algorithm. It starts from K=1 and works like FlexRPset. If the user is happy with the number of representative patterns generated at K=1, then it stops. Otherwise, it increases K to 10 and generates $C(X)$s in a way similar to Algorithm 4. The only difference is that the set of $C(X)$s generated at K=1 are taken into consideration. During the traversal of the CFP-tree, the times that an entry E is covered is decided not only by the $C(X)$s generated in the current traversal of the CFP-tree, but also by the $C(X)$s generated at K=1. The greedy set cover algorithm is then applied on all the $C(X)$s that have been generated to find representative patterns. This process is repeated until either users are happy with the number of representative patterns, or K has reached a certain limit. In every iteration, we increase K by 10 times so that a significant number of new $C(X)$s are generated in each iteration, which in turn often leads to a considerable reduction in the number of representative patterns.

6 Relaxing the ϵ-covered condition

In exploratory data analysis, users often need to inspect individual patterns manually [25]. In such situation, it is desirable to keep the number of representative patterns as few as possible to reduce human efforts. Definition 2 allows a pattern to cover its subsets only. If we remove this constraint and allow a pattern X to cover a pattern Y as long as $D(X,Y) \le \epsilon$, then the number of representative patterns can be further reduced. However, without the subset constraint, it is impossible to estimate the support of a non-representative pattern X from representative patterns as we do not know which representative pattern covers X. Fortunately, this is not a problem for MinRPset and FlexRPset because in these two algorithms, all frequent patterns are stored in a CFP-tree compactly, and users can retrieve support of patterns from the CFP-tree directly. In this section, we relax Definition 2 by removing condition $X_1 \subseteq X_2$ to further reduce the number of representative patterns. We also discuss how to modify MinRPset and FlexRPset to select representative patterns based on the relaxed definition.

Definition 3 (ϵ^*-cover): Given a real number $\epsilon \in [0,1)$ and two patterns X_1 and X_2, we say X_1 and X_2 ϵ^*-cover each other if $D(X_1,X_2) \le \epsilon$.

Obviously, if X_1 ϵ-covers X_2, then X_1 and X_2 must ϵ^*-cover each other. The reverse is not true, but the following lemma holds.

Lemma 7: If two patterns X_1 and X_2 ϵ^*-cover each other, then $X = X_1 \cup X_2$ ϵ-covers both X_1 and X_2.

Proof: $X = X_1 \cup X_2$, so we have $T(X) = T(X_1) \cap T(X_2)$. Therefore, $D(X,X_1) = 1 - \frac{|T(X)|}{|T(X_1)|} = 1 - \frac{|T(X_1)\cap T(X_2)|}{|T(X_1)|} \le 1 - \frac{|T(X_1)\cap T(X_2)|}{|T(X_1 \cup T(X_2)|} = D(X_1,X_2) \le \epsilon$. That is, X ϵ-covers X_1. Similarly, we can prove X ϵ-covers X_2. □

The above lemma indicates that if two patterns ϵ^*-cover each other, then they are likely to have similar item composition as they have a common superset that ϵ-covers both of them. Based on the above lemma and Lemma 2, given a frequent pattern X_1, if X_2 ϵ^*-covers X_1, then we must have $supp(X_2) \ge supp(X_1 \cup X_2) \ge supp(X_1)\cdot(1-\epsilon) \ge min\_sup \cdot (1-\epsilon)$.

Let $C^*(X)$ be the set of patterns that are ϵ^*-covered by X and $C(X)$ be the set of patterns that are ϵ-covered by X. We have $C(X) \subseteq C^*(X)$. Given a pattern $X' \in C^*(X)$, X' can be in three cases:

1) $X' \subseteq X$. In this case, $X' \in C(X)$. Let $C_1^*(X) = \{X' | X' \in C^*(X), X' \subseteq X\}$.
2) $X' \supset X$. In this case, $X \in C(X')$. Let $C_2^*(X) = \{X' | X' \in C^*(X), X' \supset X\}$.
3) Otherwise, based on Lemma 7, we have $X' \in C(X' \cup X)$ and $X \in C(X' \cup X)$. Let $C_3^*(X) = \{X' | X' \in C^*(X), X' - X \neq \emptyset, X - X' \neq \emptyset\}$.

We have $C^*(X) = C_1^*(X) \cup C_2^*(X) \cup C_3^*(X)$. Apparently, $C_1^*(X) = C(X)$. To generate $C_2^*(X)$, we need to find all patterns Y such that $X \in C(Y)$, and then put all such Y into $C_2^*(X)$. To generate $C_3^*(X)$, we need to find all patterns Y such that $X \in C(Y)$, and then put other patterns in $C(Y)$ that are not subsets or supersets of X into $C_3^*(X)$. However, not every such pattern in $C(Y)$ can be put into $C_3^*(X)$. A pattern in $C(Y)$ needs to satisfy some support constraint to be put into $C_3^*(X)$.

Lemma 8: Two patterns X and X' ϵ^*-cover each other *if and only if* $supp(X') \le \frac{2-\epsilon}{1-\epsilon}\cdot supp(Y) - supp(X)$, where $Y = X \cup X'$.

Proof: $supp(X') \le \frac{2-\epsilon}{1-\epsilon}\cdot supp(Y) - supp(X)$

$\Leftrightarrow \frac{supp(X)+supp(X')}{supp(Y)} \le \frac{2-\epsilon}{1-\epsilon}$

$\Leftrightarrow \frac{supp(X)+supp(X')-supp(Y)}{supp(Y)} \le \frac{1}{1-\epsilon}$

$\Leftrightarrow \frac{|T(X)\cup T(X')|}{|T(X)\cap T(X')|} \le \frac{1}{1-\epsilon} \Leftrightarrow \frac{|T(X)\cap T(X')|}{|T(X)\cup T(X')|} \ge 1-\epsilon$

$\Leftrightarrow 1 - \frac{|T(X)\cap T(X')|}{|T(X)\cup T(X')|} \le \epsilon$ □

Algorithm 5 shows the pseudo-codes for generating $C^*(X)$s from $C(X)$s. We first generate $C_2^*(X)$s from $C(X)$s (line 1-3), and then generate $C_3^*(X)$s from $C(X)$s and $C_2^*(X)$s (line 6-12). Given a pattern Y such that $X \in C(Y)$, for a pattern in $C(Y)$ to be in $C_3^*(X)$, its support cannot be larger than $(supp(Y)\cdot(\frac{2-\epsilon}{1-\epsilon}) - supp(X))$ based on Lemma 8. Therefore, we sort the patterns in $C(Y)$ in ascending support order (line 4-5). When we check the patterns in $C(Y)$ and we reach a pattern with support larger than $(supp(Y)\cdot(\frac{2-\epsilon}{1-\epsilon}) - supp(X))$, we stop the checking on $C(Y)$ (line 9-10).

In Algorithm MinRPset (Algorithm 3), if we add one extra line "$Gen\_C^*(X)(C(X)s)$" after line 3, and replace $C(X)$ with $C^*(X)$ at line 4, then we get the algorithm for finding representative patterns based on ϵ^*-cover. We denote this algorithm as MinRPset∗. Similar modification can be done to FlexRPset, and we denote the resultant algorithm as FlexRPset∗.

Complexity Analysis. Let $\hat{F}_c$ be the set of closed patterns with support no less than $min\_sup \cdot (1-\epsilon)$. For MinRPset∗, the input of

Algorithm 5 Gen_$C^*(X)$ Algorithm

```
Input:
    $C(X)$s;
Output:
    $C^*(X)$s;
Description:
 1: for each $C(X)$ do
 2:     for each pattern $X' \in C(X)$ do
 3:         Add $X$ to $C_2^*(X')$;
 4: for each $C(X)$ do
 5:     Sort the patterns in $C(X)$ in ascending support order;
 6: for each $C(X)$ do
 7:     for each pattern $Y$ in $C_2^*(X)$ do
 8:         for each pattern $X'$ in $C(Y)$ do
 9:             if $supp(X') > supp(Y) \cdot (\frac{2-\epsilon}{1-\epsilon}) - supp(X)$ then
10:                 break;
11:             else if $X' - X \neq \emptyset$ AND $X' - X \neq \emptyset$ AND
                $X' \notin C_3^*(X)$ then
12:                 Add $X'$ to $C_3^*(X)$;
```

Algorithm 5 is $S = \{C(X)|X \in \hat{F_c}\}$. The time complexity for generating $C_2^*(X)$s (lines 1-3) is $O(\sum_{X\in\hat{F_c}} C(X))$, for sorting $C(X)$s (lines 4-5) is $O(\sum_{X\in\hat{F_c}} |C(X)|log|C(X)|)$, and for generating $C_3^*(X)$s (lines 6-12) is $O(\sum_{X\in\hat{F_c}} C(X)^2)$. Hence the time complexity of Algorithm 5 is $O(\sum_{X\in\hat{F_c}} C(X)^2)$, and the time complexity of MinRPset$*$ is $O(\sum_{X\in\hat{F_c}}(2^{|X|} - 1 + C(X)^2)) + T_{FP}$, where T_{FP} is the cost for constructing the CFP-tree using a depth-first frequent pattern mining algorithm. The complexity of FlexRPset$*$ is upper bounded by that of MinRPset$*$.

MinRPset$*$ produces fewer representative patterns than MinRPset in the cost of higher computation cost as it requires an extra step—Algorithm 5—to generate $C^*(X)$s. The size of $C^*(X)$s is usually much larger than that of $C(X)$s, so the cost of the greedy set cover algorithm increases too. FlexRPset$*$ is less costly than MinRPset$*$ but it produces more representative patterns than MinRPset$*$. Our experiment results show that FlexRPset$*$ with a small K value often runs faster than MinRPset and produces fewer representative patterns than MinRPset. However, from the representative patterns produced by FlexRPset$*$ or MinRPset$*$, we cannot estimate the support of non-representative patterns even though a non-representative pattern is still similar to its representative pattern. On the contrary, the support of non-representative patterns can be estimated with bounded error from the representative patterns generated by MinRPset as discussed at the end of Section 3. This feature may be useful when storage space is a critical resource as we can store representative patterns only.

7 Experiments

Our experiments were conducted on a PC with 2.66Ghz CPU and 2.00GB memory. Our algorithms were implemented using C++. We downloaded the source codes of RPlocal from the IlliMine system package [26]. All source codes were compiled using Microsoft Visual Studio 2005.

7.1 Datasets

The datasets used in the experiments are shown in Table 3. Dataset *BMS-POS* is contributed by Blue Martini Software for KDD cup 2000 [27]. Other datasets are obtained from the FIMI repository (http://fimi.ua.ac.be/data/). Table 3 shows the number of transactions (#trans) and items (#items), the maximal and average length of transactions (MaxTL, AvgTL) on the datasets.

TABLE 3
Datasets

dataset	#trans	#items	MaxTL	AvgTL
accidents	340,183	468	52	33.81
BMS-POS	515,597	1,657	164	6.53
chess	3,196	75	37	37.00
connect	67,557	129	43	43.00
kosarak	990,002	41,270	2,497	8.10
mushroom	8,124	119	23	23.00
pumsb	49,046	2,113	74	74.00
pumsb_star	49,046	2,088	63	50.48
retail	88,162	16,470	76	10.31
webdocs	1,692,082	5,267,656	71,472	177.23

7.2 Results on ϵ-covered

The first experiment compares MinRPset and FlexRPset with RPlocal. Table 4 shows the number of frequent closed patterns (column "#closedPats"), number of representative patterns generated by RPlocal, MinRPset and FlexRPset with K=10. The number of representative patterns generated by RPlocal is more than 10 times smaller than the number of closed patterns on six datasets. MinRPset is able to further reduce the number of representative patterns on all datasets. In particular, MinRPset reduces the number of representative patterns by 67.4% on *pumsb*. The number of representative patterns generated by FlexRPset with K=10 is close to that by MinRPset. On dataset *retail*, RPlocal produces more representative patterns than the number of closed patterns, while MinRPset and FlexRPset always produce fewer representative patterns than the number of closed patterns.

Let N be the number of representative patterns generated by an algorithm. Figure 2 shows the ratio of N to the number of representative patterns generated by RPlocal when min_sup is varied. Obviously, the ratio is always 1 for RPlocal. Parameter ϵ is set to 0.1 on all datasets. MinRPset always produces fewer representative patterns than RPlocal. On *retail*, the number of representative patterns generated by FlexRPset with K=1 and MinRPset is almost identical. The same result is observed on *accidents*, *BMS-POS* and *webdocs*. On *kosarak* and *connect*, the number of representative patterns produced by FlexRPset with K=10 is pretty

TABLE 4
Comparison with RPlocal on number of representative patterns ($\epsilon = 0.1$)

dataset	min_sup	#closedPats	RPlocal	MinRPset	FlexRPset	MinRPset∗	FlexRPset∗
		(ratio to RPlocal)		(ratio to RPlocal)	K=10	(ratio to RPlocal)	K=10
accidents	0.1	9,958,684 (43.6)	228,321	174,011 (0.762)	174,938	145,707 (0.638)	148,271
BMS-POS	0.0004	928,526 (1.93)	481,772	469,783 (0.975)	469,783	468,059 (0.972)	468,059
chess	0.4	1,361,158 (47.8)	28,500	17,263 (0.606)	18,897	7,883 (0.277)	11,751
connect	0.2	1,482,862 (23.0)	64,602	49,186 (0.761)	49,596	27,481 (0.425)	30,224
kosarak	0.002	35,865 (8.29)	4,324	4,200 (0.971)	4,210	3,794 (0.877)	3,820
mushroom	0.001	147,906 (1.13)	130,729	119,211 (0.912)	119,138	116,778 (0.893)	116,778
pumsb	0.7	241,197 (92.1)	2,618	854 (0.326)	1,273	218 (0.083)	754
pumsb_star	0.1	1,512,866 (54.7)	27,660	19,767 (0.715)	20,494	12,006 (0.434)	13,731
retail	0.0001	189,078 (0.730)	258,860	170,152(0.657)	170,144	168,767 (0.652)	168,767
webdocs	0.11	101,600 (10.7)	9,466	7,485 (0.791)	7,522	5,756 (0.608)	5,811

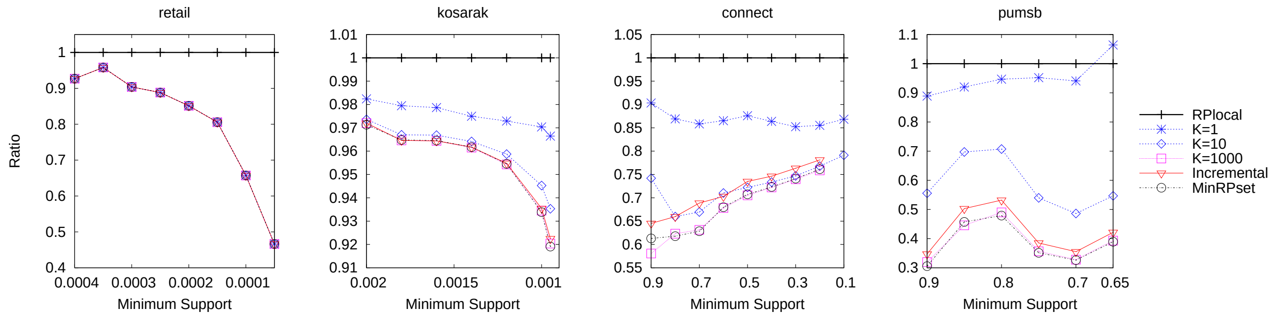

Fig. 2. Number of representative patterns when varying min_sup. ϵ=0.1

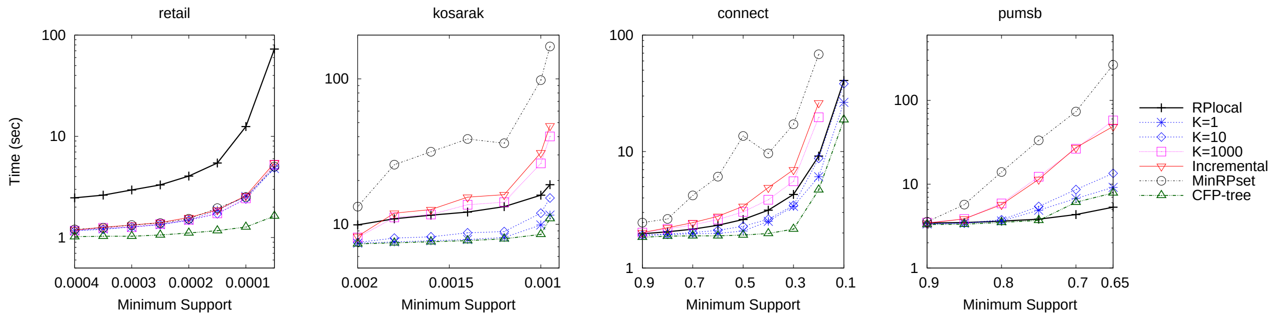

Fig. 3. Running time when varying min_sup. ϵ=0.1

close to that generated by MinRPset. The same result is observed on *mushroom* and *pumsb_star*. On *pumsb* and *chess*, FlexRPset approaches MinRPset only when K is large.

Figure 3 shows the running time of the several algorithms when min_sup is varied. The running time of MinRPset and FlexRPset includes the time for constructing CFP-tree. We use a separate curve to show the time for constructing CFP-tree, denoted as “CFP-tree”. The running time of all algorithms increases with the decrease of min_sup. MinRPset is faster than RPlocal on *retail*, *BMS-POS* and *webdocs* and slower than RPlocal on other datasets. On *kosarak* and *connect*, FlexRPset with a small K value is faster than RPlocal. With the increase of K, FlexRPset becomes slower. On other datasets, RPlocal is the fastest, and the running time of FlexRPset with K=1 is close to that of RPlocal on these datasets.

Figure 4 compares the number of representative patterns generated by the several algorithms when ϵ is varied. When ϵ increases, MinRPset generally achieves greater reduction in the number of representative patterns. However, its running time increases quickly too as shown in Figure 5. The running time of RPlocal is relatively stable with respect to ϵ, so is the running time of FlexRPset when $K \leq 10$. When K becomes larger, the running time of FlexRPset increases more quickly with ϵ.

For the incremental FlexRPset algorithm, we set the maximum value of K to 1000. The incremental algorithm tries 4 values of K incrementally: 1, 10, 100 and 1000. Figure 2 and Figure 4 show that the number of representative patterns generated by the incremental FlexRPset algorithm with K=1000 (denoted as “Incremental” in the figures) is slightly more than that generated by FlexRPset with K=1000. The reason

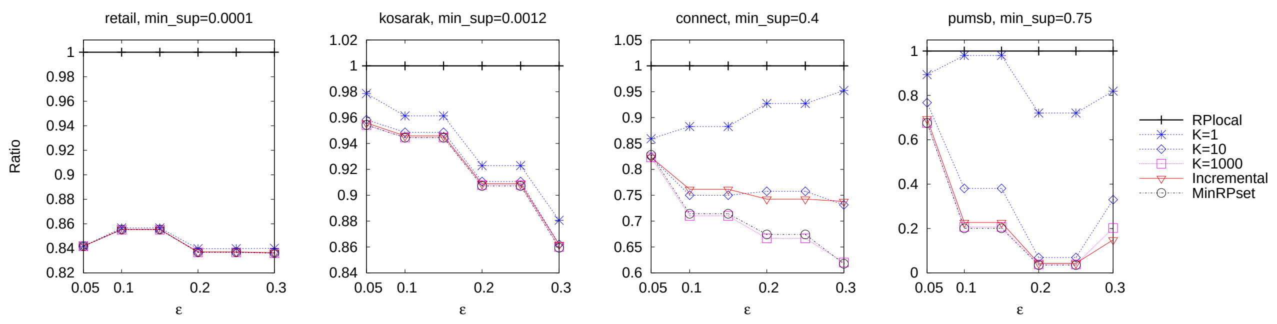

Fig. 4. Number of representative patterns when varying ϵ

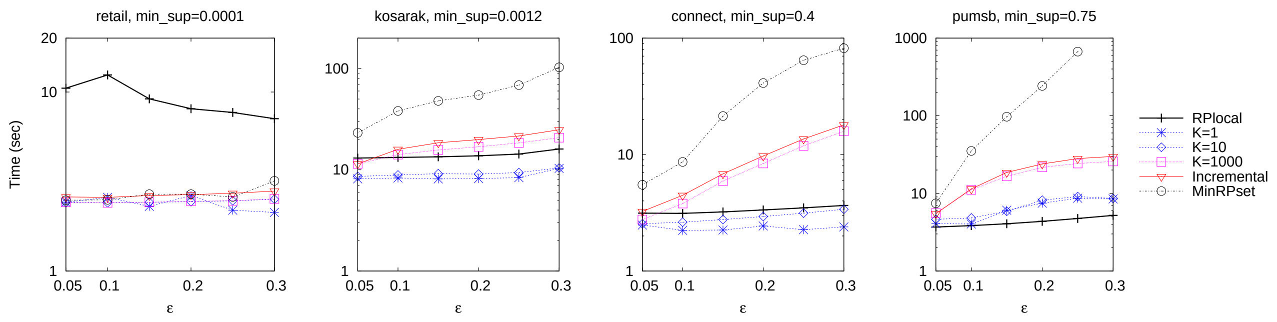

Fig. 5. Running time when varying ϵ

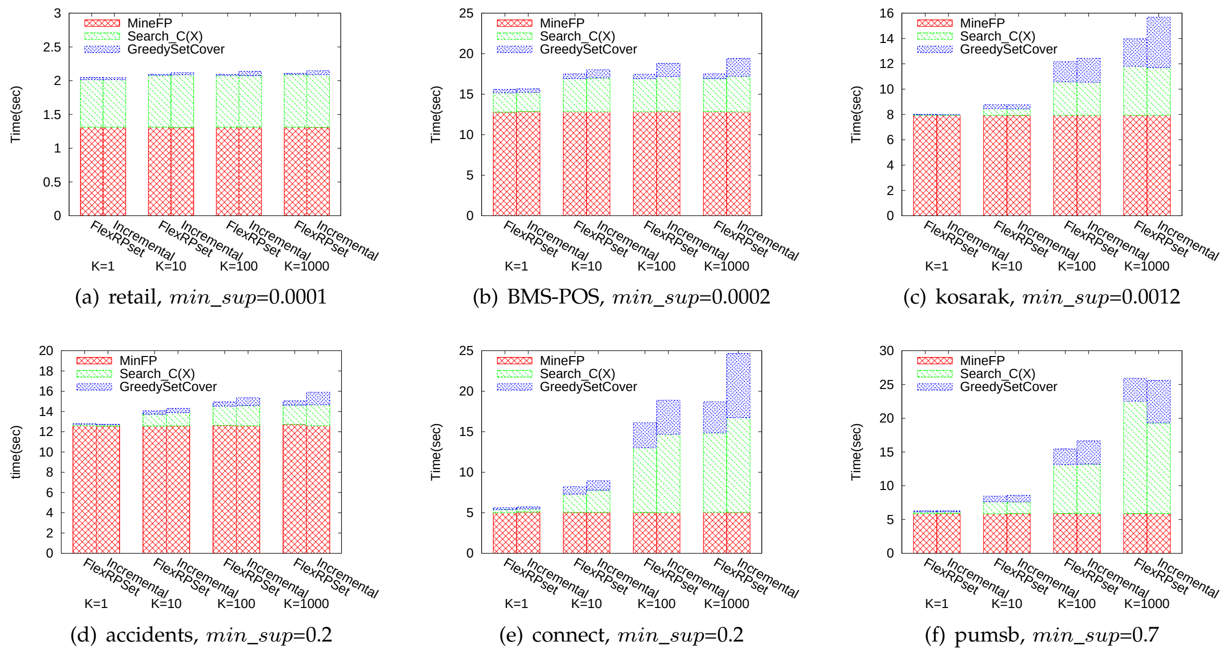

Fig. 6. Decomposition of the running time of FlexRPset and incremental FlexRPset. For each value of K, the incremental algorithm tries the following $log_{10}(K) + 1$ values incrementally: 1, 10, $\cdots$, K. On all datasets, ϵ=0.1.

being that the two algorithms use different strategies to selectively generate $C(X)$s. The strategy used by FlexRPset seems to be more effective in reducing the final number of representative patterns.

Figure 3 and Figure 5 show that the running time of the incremental algorithm is slightly longer than that of FlexRPset with K=1000. We decompose the running time of the two algorithms into three parts: frequent pattern mining (denoted as "MineFP" in the figures), generating $C(X)$s (denoted as "Search_C(X)") and finding a minimum set cover (denoted as "GreedySetCover"). Figure 6 shows that the most costly steps in the two algorithms are frequent pattern mining and generating $C(X)$s. These

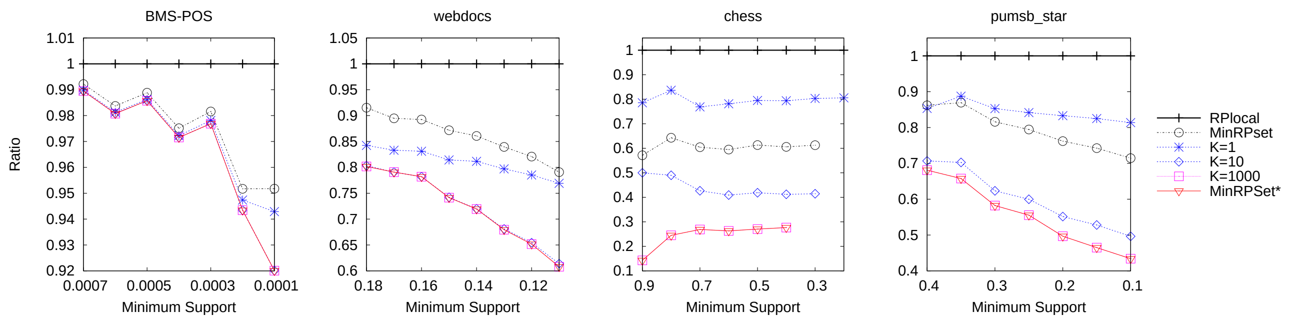

Fig. 7. Number of representative patterns when varying min_sup for $\epsilon*$-covered. ϵ=0.1

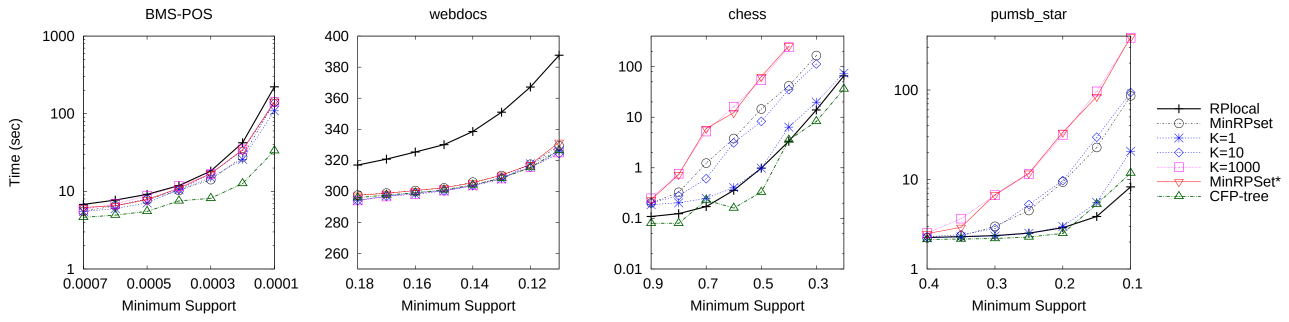

Fig. 8. Running time when varying min_sup for $\epsilon*$-covered. ϵ=0.1

TABLE 5
The effect of the early termination technique.

dataset	min_sup	ϵ	W/O(sec)	With(sec)	speedup
accidents	0.2	0.1	11.31	2.97	3.81
accidents	0.2	0.05	9.03	1.84	4.90
BMS-POS	0.0001	0.1	131.14	31.87	4.11
BMS-POS	0.0001	0.05	115.87	28.81	4.02
chess	0.3	0.1	284.76	51.50	5.53
chess	0.3	0.05	209.98	25.73	8.16
connect	0.2	0.1	94.05	28.69	3.28
connect	0.2	0.05	77.67	11.30	6.88
kosarak	0.0009	0.1	318.90	280.37	1.14
kosarak	0.0009	0.05	62.83	42.73	1.47
mushroom	0.001	0.2	2.89	0.266	10.87
mushroom	0.001	0.1	2.86	0.234	12.22
mushroom	0.001	0.05	2.828	0.219	12.91
pumsb	0.65	0.1	82.77	67.09	1.23
pumsb	0.65	0.05	33.905	16.58	2.05
pumsb_star	0.1	0.1	97.62	39.97	2.44
pumsb_star	0.1	0.05	77.69	22.52	3.45
retail	0.00003	0.1	6.64	5.86	1.13
retail	0.00003	0.05	6.66	5.86	1.14
webdocs	0.08	0.1	13.70	9.37	1.46
webdocs	0.08	0.05	7.97	2.70	2.95

two steps are shared among different K values in the incremental algorithm, hence the incremental algorithm has similar running time with FlexRPset for the two steps. The incremental algorithm needs to call the greedy set cover algorithm multiple times, one for each K value, so it has longer running time for this part. Overall, the incremental algorithm is just slightly slower than FlexRPset in most of the cases except on *pumsb* at K=1000. The two algorithms use different strategies to selectively generate $C(X)$s. On *pumsb*, the incremental algorithm generates fewer $C(X)$s than FlexRPset when K=1000, so its cost for $Search\_C(X)$, as well as its overall cost, is lower.

7.3 Effect of the early termination technique

In Algorithm 1, we use an early termination technique (described at the end of Section 4.2.1) to improve the efficiency of Algorithm 1. Table 5 shows the effect of the early termination technique on the generating time of $C(X)$s in MinRPSet. Columns "W/O (sec)" and "With (sec)" are the generation time of $C(X)$s without and with the early termination technique respectively. The last column is the speedup achieved by using the early termination technique. On most of the datasets, the early termination technique achieves several times of speedup. On *mushroom*, the speedup is more than 10 times. The early termination technique is more effective when ϵ is smaller. This is because when ϵ is smaller, fewer subsets of X can be ϵ-covered by X, and more subsets of X that do not satisfy the support constraint are pruned by the early termination technique.

7.4 Results on ϵ^*-cover

In this experiment, we study the performance of MinRPset$*$ and FlexRPset$*$. They are compared with RPlocal and MinRPset. The last two columns in Table 4 show the number of representative patterns produced by MinRPset$*$ and FlexRPset$*$ with K=10, which is fewer than that by MinRPset and RPlocal on all datasets.

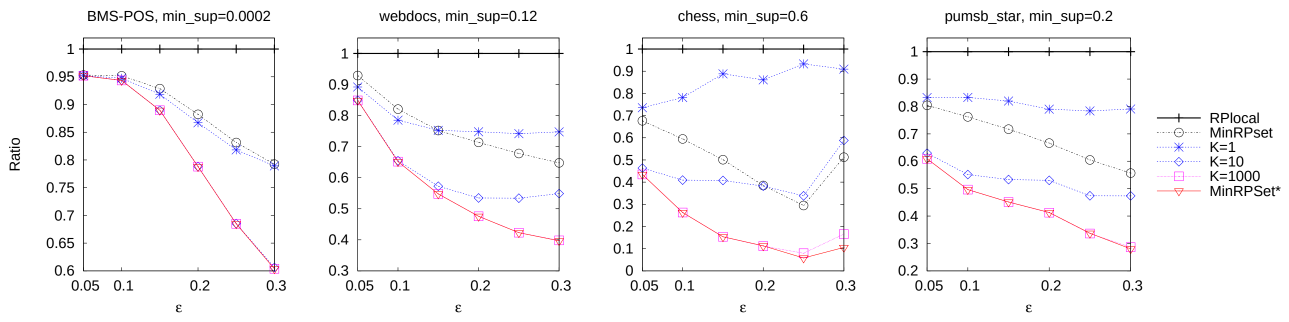

Fig. 9. Number of representative patterns when varying ϵ for ϵ∗-covered.

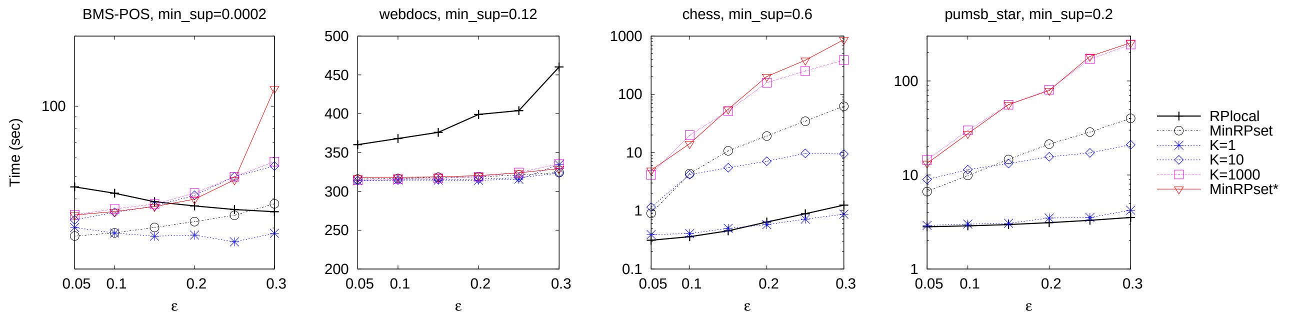

Fig. 10. Running time when varying ϵ for ϵ∗-covered.

Figure 7 shows the ratio of the number of representative patterns generated by different algorithms to that of RPlocal when min_sup is varied. On *BMS-POS*, the number of representative patterns generated by MinRPset, MinRPset∗ and FlexRPset∗ with different K values is very close. We observed the same result on *retail*. On *webdocs*, FlexRPset∗ with K=1 already generates fewer representative patterns than MinRPset. The same result is observed on *kosarak* and *accidents*. On other datasets, FlexRPset∗ with K=10 produces fewer representative patterns than MinRPset.

Figure 8 shows the running time of the algorithms when min_sup is varied. MinRPset∗ has similar running time with MinRPset and FlexRPset∗ on *BMS-POS* and *webdocs*. Similar result is observed on *retail* and *mushroom*. MinRPset∗ is slower than MinRPset on other datasets. The extra cost is mainly spent on generating $C^*(X)$s from $C(X)$s. When K=1, the running time of FlexRPset∗ is close to that of RPlocal in most cases.

Figure 9 and Figure 10 show the performance of MinRPset∗ and FlexRPset∗ when ϵ is varied. Similar to MinRPset and FlexRPset, MinRPset∗ and FlexRPset∗ achieve bigger reduction in the number of representative patterns when ϵ increases, but their running time also increases.

8 Discussion and Conclusion

In this paper, we have described two algorithms, MinRPset and FlexRPset, for finding minimum representative pattern sets. Both algorithms first mine frequent patterns, and then find representative patterns in a post-processing step, while RPlocal integrates frequent pattern mining with representative pattern finding. Due to the use of the post-processing strategy, MinRPset and FlexRPset have the following additional benefits besides producing fewer representative patterns: 1) Users may not know what value should be used for ϵ at the beginning. The post-processing strategy allows users to try different ϵ values without mining frequent patterns multiple times. This is especially beneficial on very large datasets. 2) In MinRPset and FlexRPset, it is easy to keep record of the set of patterns covered by each representative pattern. This information is useful for users to inspect individual representative patterns in more details. 3) We can relax the conditions on ϵ-covered to further reduce the number of representative patterns as what we have done in Section 6.

MinRPset and FlexRPset have some drawbacks. On some dense datasets, MinRPset and FlexRPset with a large K value are often much slower than RPlocal. Furthermore, the greedy set cover algorithm needs to hold all the $C(X)$s and their inversion in memory, otherwise it becomes very slow. When the number of frequent closed patterns reaches several millions, MinRPset and FlexRPset with a large K value become infeasible on some dense datasets because all the $C(X)$s and their inversion are too large to fit into memory. RPlocal and FlexRPset with a small K value are less memory-demanding, and they can still work in such situation. The incremental approach is particularly helpful in such situation as it allows users to have a result as early as possible.

In summary, both MinRPset and FlexRPset generate fewer representative patterns than previous work RPlocal. MinRPset is often more expensive than RPlocal. FlexRPset takes one extra parameter K, which allows users to make a trade-off between result size and running time. Users can make the trade-off conveniently using the incremental approach. When K is small, FlexRPset is usually faster than or has similar running time with RPlocal. We also allow users to relax the conditions in the problem definition to further reduce the number of representative patterns. Hence our approach is a very flexible approach to finding representative patterns.

9 Acknowledgment

This work is supported in part by Singapore Agency for Science, Technology and Research grant SERC 102 101 0030.

References

[1] R. Agrawal, T. Imielinski, and A. N. Swami, "Mining association rules between sets of items in large databases," in *SIGMOD Conference*, 1993, pp. 207–216.
[2] B. Goethals and M. J. Zaki, "Advances in frequent itemset mining implementations: Introduction to fimi03," in *Proc. of the ICDM 2003 Workshop on Frequent Itemset Mining Implementations*, 2003.
[3] N. Pasquier, Y. Bastide, R. Taouil, and L. Lakhal, "Discovering frequent closed itemsets for association rules," in *ICDT*, 1999, pp. 398–416.
[4] D. Xin, J. Han, X. Yan, and H. Cheng, "Mining compressed frequent-pattern sets," in *VLDB*, 2005, pp. 709–720.
[5] V. Chvatal, "A greedy heuristic for the set-covering problem," *Mathematics of Operations Research*, vol. 4, no. 3, pp. 233–235, 1979.
[6] G. Grahne and J. Zhu, "Efficiently using prefix-trees in mining frequent itemsets," in *FIMI*, 2003.
[7] G. Liu, H. Lu, and J. X. Yu, "Cfp-tree: A compact disk-based structure for storing and querying frequent itemsets," *Inf. Syst.*, vol. 32, no. 2, pp. 295–319, 2007.
[8] Y. Bastide, N. Pasquier, R. Taouil, G. Stumme, and L. Lakhal, "Mining minimal non-redundant association rules using frequent closed itemsets," in *Proc. of Computational Logic Conference*, 2000, pp. 972–986.
[9] A. Bykowski and C. Rigotti, "A condensed representation to find frequent patterns," in *PODS*, 2001.
[10] J.-F. Boulicaut, A. Bykowski, and C. Rigotti, "Free-sets: A condensed representation of boolean data for the approximation of frequency queries," *Data Mining and Knowledge Discovery*, vol. 7, no. 1, pp. 5–22, 2003.
[11] T. Calders and B. Goethals, "Mining all non-derivable frequent itemsets," in *PKDD*, 2002, pp. 74–85.
[12] R. J. Bayardo, "Efficiently mining long patterns from databases," in *SIGMOD Conference*, 1998, pp. 85–93.
[13] J. Wang, J. Han, Y. Lu, and P. Tzvetkov, "Tfp: An efficient algorithm for mining top-k frequent closed itemsets," *IEEE Trans. Knowl. Data Eng.*, vol. 17, no. 5, pp. 652–664, 2005.
[14] D. Xin, H. Cheng, X. Yan, and J. Han, "Extracting redundancy-aware top-k patterns," in *KDD*, 2006, pp. 444–453.
[15] F. N. Afrati, A. Gionis, and H. Mannila, "Approximating a collection of frequent sets," in *KDD*, 2004, pp. 12–19.
[16] J. Pei, G. Dong, W. Zou, and J. Han, "Mining condensed frequent-pattern bases," *Knowledge and Information Systems*, vol. 6, no. 5, pp. 570–594, 2004.
[17] X. Yan, H. Cheng, J. Han, and D. Xin, "Summarizing itemset patterns: a profile-based approach," in *KDD*, 2005, pp. 314–323.
[18] R. Jin, M. Abu-Ata, Y. Xiang, and N. Ruan, "Effective and efficient itemset pattern summarization: regression-based approaches," in *KDD*, 2008, pp. 399–407.
[19] A. K. Poernomo and V. Gopalkrishnan, "Cp-summary: a concise representation for browsing frequent itemsets," in *KDD*, 2009, pp. 687–696.
[20] C. Wang and S. Parthasarathy, "Summarizing itemset patterns using probabilistic models," in *KDD*, 2006, pp. 730–735.
[21] M. Mampaey, N. Tatti, and J. Vreeken, "Tell me what i need to know: succinctly summarizing data with itemsets," in *KDD*, 2011, pp. 573–581.
[22] R. Rymon, "Search through systematic set enumeration," in *KR*, 1992, pp. 539–550.
[23] J. Wang, J. Han, and J. Pei, "Closet+: searching for the best strategies for mining frequent closed itemsets," in *KDD*, 2003, pp. 236–245.
[24] T. Westmann, D. Kossmann, S. Helmer, and G. Moerkotte, "The implementation and performance of compressed databases," *SIGMOD Record*, vol. 29, no. 3, pp. 55–67, 2000.
[25] K. Zhao, B. Liu, J. Benkler, and W. Xiao, "Opportunity map: identifying causes of failure—a deployed data mining system," in *SIGKDD*, 2006, pp. 892–901.
[26] "Illimine system package. http://illimine.cs.uiuc.edu/download/."
[27] R. Kohavi, C. E. Brodley, B. Frasca, L. Mason, and Z. Zheng, "Kdd-cup 2000 organizers' report: Peeling the onion," *SIGKDD Explorations*, vol. 2, no. 2, pp. 86–98, 2000.

Guimei Liu is a senior research fellow at School of Computing, National University of Singapore. She obtained her BSc in Computer Science from Peking University, China, and PhD from Hong Kong University of Science & Technology. Her main research interests are frequent pattern mining, using patterns for exploratory data analysis, graph mining and bioinformatics.

Haojun Zhang is a research assistance and a part-time Master student at School of Computing, National University of Singapore. He obtained his Bachelor in Computer Science from National University of Singapore.

Limsoon Wong is a professor of computer science and pathology at The National University of Singapore. He currently works mostly on knowledge discovery technologies and their application to biomedicine. He has also done significant research in database query language theory and finite model theory, as well as significant development work in broad-scale data integration systems. He serves on the editorial boards of Journal of Bioinformatics and Computational Biology (ICP), Bioinformatics (OUP), and Drug Discovery Today (Elsevier).. He received his BSc(Eng) from Imperial College London and his PhD from The University of Pennsylvania.