Intelligent Data Analysis 12 (2008) 321–338
IOS Press

Index-CloseMiner: An improved algorithm for mining frequent closed itemset

Wei Song[a,*], Bingru Yang[b] and Zhangyan Xu[c]
[a]*College of Information Engineering, North China University of Technology, Beijing, 100144, China*
[b]*School of Information Engineering, University of Science and Technology Beijing, Beijing, 100083, China*
[c]*Department of Computer, Guangxi Normal University, Guilin, 541004, China*

Abstract. The set of frequent closed itemsets determines exactly the complete set of all frequent itemsets and is usually much smaller than the latter. This paper proposes an improved algorithm for mining frequent closed itemsets. Firstly, the index array is proposed, which is used for discovering those items that always appear together. Then, by using bitmap, an algorithm for computing index array is presented. Thirdly, based on the heuristic information provided by index array, frequent items, which co-occur together and share the same support, are merged together. Thus, initial generators are calculated. Finally, based on index array, reduced pre-set and reduced post-set are proposed. It is proved that the reduced pre-set and reduced post-set not only retain the function of pre-set and post-set, but also have smaller sizes. Therefore, the redundant items in pre-set and post-set are deleted, thus making it possible to save a lot of work related to inclusion check. The experimental results show that the proposed algorithm is efficient especially on dense dataset.

Keywords: Data mining, association rule, frequent closed itemset, index array, subsume index

1. Introduction

1.1. Motivation

Frequent Itemset Mining (FIM) is one of the major problems in many data mining applications. It started as a phase in the discovery of association rules [1], but has been generalized independent of these to many other patterns. For example, frequent sequences [2], episodes [3], and frequent subgraphs [4].

The problem of FIM can be stated as follows: Let $I = \{i_1, i_2, \ldots, i_M\}$ be a finite set of items and D be a dataset containing N transactions, where each transaction $t \in D$ is a list of distinct items $t = \{i_1, i_2, \ldots, i_{|t|}\}$, $i_j \in I (1 \leqslant j \leqslant |t|)$. Let X be a *k-itemset*, where $X = \{i_1, i_2, \ldots, i_k\}$ is a set of k distinct items. Given a k-itemset X, let *supp*(X) be its *support*, defined as the number of transactions in D that include X. Mining all the frequent itemsets from D requires discovering all the itemsets having a support higher than (or equal to) a given threshold *min_supp*. This requires browsing the huge search space given by the power set of I.

The FIM problem has been extensively studied in the last years. Several variations to the original Apriori algorithm [5], as well as completely different approaches, have been proposed [6–9]. Most of the

*Corresponding author. E-mail: sgyzfr@yahoo.com.cn.

well-studied frequent pattern mining algorithms, including Apriori [5], ECLAT [8], and FP-Growth [9], mine the complete set of frequent itemsets. These algorithms may have good performance when the support threshold is high and the pattern space is sparse. However, when the support threshold drops low, the number of frequent itemsets goes up dramatically, the performance of these algorithms deteriorates quickly because of the generation of a huge number of itemsets. Moreover, the effectiveness of the mining of the complete set degrades because it generates numerous redundant itemsets.

There are mainly two current solutions to address this problem. The first one is to mine only the maximal frequent itemsets [10], which are typically orders of magnitude fewer than all frequent itemsets. While mining maximal sets help understand the long itemsets in dense domains, they lead to a loss of information; since subset frequency is not available. Thus, maximal sets are not suitable for generating rules.

The second is to mine only the Frequent Closed Itemsets (FCI) [11]. Closed sets are *lossless* in the sense that they uniquely determine the set of all frequent itemsets and their exact frequency. At the same time closed sets can themselves be orders of magnitude smaller than all frequent sets, especially on dense databases. More importantly, association rules extracted from closed itemsets have been proven to be more meaningful for analysts, because all redundancies are discarded [12,13]. In this paper, we study efficient algorithm for mining frequent closed itemsets.

1.2. Related work

In 1999, Pasquier et al. proposed to mine only closed set of frequent itemsets instead of complete set [11]. They developed an Apriori-based algorithm A-Close that employs breadth-first search to find FCI. A-Close constructs a generator set to integrate the pruning step to limit the search space. But A-Close still suffers from redundant scans of datasets and high costs of itemset matching inherent to breadth-first search. Other "generate-and-test" algorithms include Close [14] and Titanic [15].

Pei et al. proposed the algorithm CLOSET [16] based on FP-Growth. Using a depth-first traversal of the search space, CLOSET tries to split the extraction context, stored in a global FP-tree, into smaller sub-contexts and to recursively applies the FCI mining process on these sub-contexts. The mining process also heavily relies on the search space pruning. This pruning is also based on statistical metrics in conjunction with introduced heuristics. CLOSET suffers from inefficiency caused by recursive building of "conditional FP-trees" that consume lots of CPU time and memory. Some improvements or alternatives of this algorithm were proposed, mainly CLOSET+ [17], Afopt-Close [18] and FP-Close [19], while respecting its driving idea.

Unlike methods which exploit only the (closed) itemset search space, ChARM [20] uses a vertical format, and simultaneously explores both the closed itemsets search space and that of transactions thanks to an introduced data structure called IT-tree (Itemset-Tidset tree). Each node in the IT-tree contains an FCI candidate and the list of the transactions to which it belongs, i.e., tidset. ChARM explores the search space in a depth-first manner, without splitting the extraction context into smaller sub-contexts. However, it generates each time a single candidate. Then, it tries to test whether it is an FCI or not, using tidset intersections and subsumption checking. In addition, the *diffset* technique is used to reduce the size of the row id lists and the computational complexity. Since different paths can lead to the same closed itemset, ChARM exploits a hash table to quickly individuate all the already mined closed itemsets Y that subsumes a given frequent itemset X. CloseMiner [21] is a variant of ChARM. The main idea of CloseMiner is to group the complete set of itemsets into non-overlapping clusters and each cluster is uniquely identified by a closed tidset.

Both LCM [22] and DCI-Closed [23,24] algorithms inherit the use of tidsets adopted in ChARM. They traverse the search space in a depth-first manner. After discovering an FCI Z, they generate a new generator *gen* by extending Z with a frequent item i, $i \notin Z$. Using a total order relation on frequent items, LCM and DCI-Closed verify if *gen* infringes this order by performing tests using only the tidset of Z and those of the frequent items. If *gen* is not discarded, *gen* is an order-preserving generator of a new FCI. Then, its closure is computed using the previously mentioned tidsets. These algorithms do not need to store, in main memory, the set containing the previously mined FCIs. The differences between LCM and DCI-Closed are the strategies for taking closure and the adopted data structures for storing the extraction context in main memory.

1.3. Our contributions

By introducing the order-preserving property of generators, DCI-Closed [23,24] devises a general and memory-efficient technique to avoid duplicate generation. It is proved that, for each closed itemset, it is possible to devise one and only one sequence of order-preserving generators. In order to check whether a given generator is order-preserving or not, DCI-Closed introduces the definitions of pre-set and post-set. The sequences of order-preserving generators are obtained by checking whether the tidset of candidate generators are subsumed by the tidsets of items in pre-set and post-set. However, there are redundant elements in both pre-set and post-set, especially when the datasets contain a large number of items. Thus, these checking operations are time-consuming sometimes.

To solve the above problems, we propose an improved algorithm Index-CloseMiner for frequent closed itemset mining. The contributions of this paper are listed as follows:

1. The *index array* is proposed, which is used for discovering those itemsets that always co-occur together. Then, by using bitmap, an algorithm for computing index array is presented.
2. Based on index array, we can identify some closed itemsets directly. These closed itemsets are used as initial order-preserving generators in our Index-ClosedMiner algorithm.
3. Based on index array, reduced pre-set and reduced post-set are proposed. It is proved that the reduced pre-set and reduced post-set retain the function of original ones used in DCI-Closed. Thus, the redundant operations on duplication checks are avoided greatly.

The experimental results show that the proposed algorithm is efficient especially on dense dataset.

The remaining of the paper is organized as follows. In Section 2, we briefly revisit the problem definition of frequent closed itemset mining. In Section 3, we present the definition of index array and the corresponding algorithm for generating index array. In Section 4, we devise algorithm Index-CloseMiner by exploiting the heuristic information provided by index array. A thorough performance study of Index-CloseMiner in comparison with several recently developed efficient algorithms is reported in Section 5. We conclude this study in Section 6.

2. Problem statement

Let T and X, $T \subseteq D$ and $X \subseteq I$, be subsets of all the transactions and items appearing in D, respectively. The concept of closed itemset is based on the two following functions, f and g:

$$f(T) = \{i \in I | \forall t \in T, i \in t\}$$

$$g(X) = \{t \in D | \forall i \in X, i \in t\}$$

TID	Items
1	A B C D E F
2	A E G
3	B C E F
4	A B C D F
5	B C E F

Fig. 1. The example database.

Function f associates with T the items common to all objects $t \in T$ and g associates with X the objects related to all items $i \in X$.

Definition 1. An itemset X is said to be closed if and only if

$$c(X) = f(g(X)) = f \circ g(X) = X$$

where the composite function $c = f \circ g$ is called the Galois operator or closure operator. A closed itemset X is frequent if its support is lager or equal to the given support threshold *min_supp*.

The closure operator defines a set of equivalence classes over the lattice of frequent itemsets: Two itemsets belong to the same equivalence class if and only if they have the same closure, i.e., they are supported by the same set of transactions. We can also show that an itemset X is closed if and only if no supersets of X with the same support exist. Therefore, mining the maximal elements of all the equivalence classes corresponds to mine all the closed itemsets.

An example database is given in Fig. 1. For convenience, we write an itemset $\{A, B, C\}$ as *ABC*, and a set of transaction identifiers $\{2,4,5\}$ as 245. In the example database, $f(14) = f(1) \cap f(4)$ =*ABCDEF*$\cap$ *ABCDF*=*ABCDF*, and g(*BCF*)=$g(B) \cap g(C) \cap g(F)$ = 1345$\cap$1345$\cap$1345=1345. Since c(*BCF*)=$f \circ g$(*BCF*)=f(1345)=*BCF*, *BCF* is a closed itemset.

3. Generation of index array

To reduce the search space as well as the sizes of pre-set and post-set, we introduce the definition of index array.

Definition 2. An *index array* is an array with size $m1$, where $m1$ is the number of frequent 1-itemset. Each element of the array corresponds to a tuple (*item*, *subsume*), where *item* is an item, *subsume*(*item*)=$\{j \in I | j \neq item \wedge g(item) \subseteq g(j)\}$. For each element of index array, we call *item* the *representative item*, and *subsume*(*item*) the *subsume index*.

The *subsume index* of *item* is an itemset, whose meaning is if $j \in$*subsume* (*item*), the tidset of *item* is the subset of the tidset of j. For example, in the example dataset (shown in Fig. 1), the subsume index of D is *ABCF*. By using subsume index, those itemsets co-occur together and share the same support, can be merged together.

The pseudocode for generating index array is shown in Algorithm 1.

In Algorithm 1, the database *D* is first scanned once to determine the frequent single items (Step 1). In Step 2, frequent items are sorted in certain order (for example, lexicographic order), and the sorted frequent items are assigned to the elements of index array as representative items one by one (Steps 3–4). In Step 5, the bitmap representation of database *D* is built. That is, for a transaction T, if

Algorithm 1 Generating index array

Input: dataset $\boldsymbol{D}$, *min_supp*

Output: index array

1: Scan database $\boldsymbol{D}$ once. Delete infrequent items;
2: Sort frequent single items in certain order as $a_1, a_2, \ldots, a_m$;
3: **for** each element *index*[*j*] of index array **do**
4: *index*[*j*].*item*=a_j;
5: Represent the database $\boldsymbol{D}$ with bitmap
6: **for** each element *index*[*j*] in index array **do**
7: *index*[*j*].*subsume*=$\varnothing$;
8: *candidate* = $\bigcap_{m \in \{index[j].item\}} t$;
9: Store the item corresponding to the i_k^{th} position in candidate to *index*[*j*].*subsume* (excluding *index*[*j*].*item*), if the value of the i_k^{th} bit in candidate is set;
10: **for** each item *i* in *index*[*j*].*subsume* **do**
11: **if** *supp*(*index* [*j*].*item*)==*supp*(*i*) **then**
12: delete *index* [*k*] with *index*[*k*].*item*=*i*;
13: **end for**
14: **Write Out** the index array;

1-frequent itemset i is contained by T, then the bit corresponding to i in T will be set. The index array is calculated by the main loop (Steps 6–13). New candidate of subsume index is formed by intersecting all transactions containing item *index*[j].*item* (Steps 7–8). Then the subsume index is obtained according to Definition 2 (Step 9). Next, the heuristic information, provided by supports of a frequent item i and items in *subsume*(i), is used for pruning (Steps 10–12). This operation can avoid the generation of duplicate initial generators (See Section 4.1). The correctness of this pruning strategy is confirmed by the following Theorem 1.

Theorem 1. If item $j \in$*subsume*(i) and *supp*(i)=*supp*(j), then $i \cup$*subsume*(i) = $j \cup$*subsume*(j).

Proof. Since $j \in$*subsume*(i), according to Definition 2, we know that $g(i) \subseteq g(j)$. Furthermore, since *supp*(i)=*supp*(j), we have $|g(i)| = |g(j)|$. Thus, $g(i) = g(j)$.

For $\forall x \in i \cup$*subsume*(i), there are two cases:

1. if $x = i$, we have $g(x) = g(i) = g(j)$, so $g(j) \subseteq g(x)$. That is to say $x \in$*subsume*(j) $\subset$ $j \cup$*subsume*(j).
2. if $x \in$*subsume*(i), according to Definition 2, we know that $g(i) \subseteq g(x)$. Since $g(i) = g(j)$, we have $g(j) \subseteq g(x)$. That is to say $x \in$*subsume*(j) $\subset$ $j \cup$*subsume*(j).

According to the discussions of 1) and 2), we have $i \cup$*subsume*(i) $\subseteq$ $j \cup$*subsume*(j).
Similarly, we can also prove that $j \cup$*subsume*(j) $\subseteq$ $i \cup$*subsume*(i).
Thus, we have $i \cup$*subsume*(i) = $j \cup$*subsume*(j).

Example 1. We use the example database in Fig. 1 to illustrate the basic idea of Algorithm 1.

Suppose the support threshold *min_supp* is 2, after the first scan of database, infrequent item G is deleted. Then the scanned database is represented by bitmap (shown in Fig. 2).

	A	*B*	*C*	*D*	*E*	*F*
1	1	1	1	1	1	1
2	1	0	0	0	1	0
3	0	1	1	0	1	1
4	1	1	1	1	0	1
5	0	1	1	0	1	1

Fig. 2. Bitmap representation of example database.

Then the subsume indices of each frequent 1-itemset are computed one by one. Take item D as example, *candidate*$= \bigcap_{t \in g(D)} t$ =1∩4=111111∩111101=111101, where 1 and 4 are tids. There are five 1 in *candidate*, since the fourth bit corresponds to D itself, the items, corresponds the first, the second, the third and the sixth bit, constitute the subsume index of D, that is *ABCF*. This means that transactions, which support item D, also support items A, B, C, and F. We can iterate this process similarly, and finally the index array is (A, $\emptyset$), (B, *CF*), (D, *ABCF*), (E, $\emptyset$). Note that, since items $C \in$*subsume*(B), $F \in$*subsume*(B), and *supp*(B)=*supp*(C)=*supp*(F), we delete elements of index array with C and F as representative items.

4. The Index-CloseMiner algorithm

The goal of an effective browsing strategy should be to identify exactly a single itemset for each equivalence class. We could in fact mine all the closed itemsets by computing the closure of just this single representative itemset for each equivalence class, without generating any duplicate. The representative itemsets are usually called as *generators*.

The advantages of DCI-Closed algorithm include: 1) It does not need to mine closed itemsets according to a "strict" lexicographic order and to keep previously mined closed itemsets in the main memory to perform duplicate checks. 2) It permits subdividing the search space and produce completely independent sub-problems, which can be solved in whatever order and, thus, also in parallel. These benefits are achieved according to the following definitions and theorems.

Definition 3. [23,24]. A generator of the form $X = Y \cup i$, where Y is a closed itemset and $i \notin Y$, is said to be order-preserving if and only if either $c(X) = X$ or $i \prec (c(X) \backslash X)$.

Given any total order relation *R* defined among items, hereinafter we will always consider an itemset as an *ordered set*, and denote with symbol "$\prec$" the order between two ordered itemsets.

Theorem 2 [23,24]. For each closed itemset $\overline{Y} \neq c(\emptyset)$, there exists a sequence of n items $i_0 \prec i_1 \prec \ldots \prec i_{n-1}, n \geqslant 1$, such that

$<$*gen*$_0$, *gen*$_1$, …, *gen*$_{n-1}$ $>=<$ $Y_0 \cup i_0, Y_1 \cup i_1, \ldots, Y_{n-1} \cup i_{n-1} >$

where the various *gen*$_i$ are order-preserving generators, with $Y_0 = c(\emptyset), \forall j \in [0, n-1], Y_{j+1} = c(Y_j \cup i_j)$, and $Y_n = \overline{Y}$.

Note that the closed itemset $c(\emptyset)$ contains, if any, the items that occur in all the transactions of the database *D*. If no such items exist, then $c(\emptyset) = \emptyset$.

Corollary 1 [23,24]. For each closed itemset $\overline{Y} \neq c(\emptyset)$, the sequence of order-preserving generators of Theorem 2 is unique.

In order to exploit the results of Theorem 2, DCI-Closed introduce the definitions pre-set and post-set to check whether a given generator is order-preserving or not.

Definition 4. [23,24]. Given a generator of the form *gen*=$Y \cup i$, where Y is a closed itemset and $i \notin Y$, *pre-set* (*gen*) is defined as follows:

$$pre - set(gen) = \{j \in I | j \notin gen \wedge j \prec i\}$$

Theorem 3. [23,24]. Let *gen*=$Y \cup i$ be a generator, where Y is a closed itemset and $i \notin Y$. If $\exists j \in$*pre-set*(*gen*) such that g(*gen*)$\subseteq g(j)$, then *gen* is not order-preserving.

Definition 5. [23,24]. Given a generator of the form *gen*=$Y \cup i$, where Y is a closed itemset and $i \nexists Y$, *post-set* (*gen*) is defined as follows:??

$$post - set(gen) = \{j \in I | j \notin gen \wedge i \prec j\}$$

As we can see from the above definitions and theorems, the sequences of order-preserving generators are obtained by checking whether the tidsets of candidate generators are subsumed by the tidsets of items in pre-set and post-set. However, there are a lot of redundant elements in both pre-set and post-set especially when the datasets contain a large number of items. Therefore these checking operations are time-consuming sometimes. The aim of our algorithm is to reduce these redundant operations as greatly as possible.

4.1. Initial generators

In DCI-Closed algorithm, every order-preserving sequence starts from $c(\emptyset)$. In most cases of real-world datasets, $c(\emptyset) = \emptyset$, thus every frequent 1-itemset will be used as candidate generator. By using the following Theorem 4, Index-CloseMiner reduced the number of initial generators greatly. That is because only closed itemsets are used as the initial generators.

Theorem 4. Let X be an itemset, $X \cup$*subsume*(X) is a closed itemset, and *supp*($X \cup$*subsume*(X))=*supp* (X).

Proof. According to Definition 2, for $\forall i \in$*subsume*(X), we have $g(X) \subseteq g(i)$. Thus, g($X \cup$*subsume*(X))=$g(X) \cap g$(*subsume*(X))=$g(X)$. So *supp*($X \cup$*subsume*(X))=*supp*(X). Assume that $X \cup$*subsume*(X) is not a closed itemset, according to the Definition 1, there at least exists an itemset Y, such that $X \cup$*subsume*(X) $\subset Y$ and *supp*($X \cup$*subsume*(X))=*supp*(Y), so we have $g(X \cup$*subsume*(X))=$g(Y)$. Thus, there at least exists a 1-itemset $i \in Y \wedge i \notin X \cup$*subsume*($X$), such that $g(X) = g(X \cup$*subsume*(X))=$g(Y) \subseteq g(i)$. According to Definition 2, $i \in$*subsume*(X). This is in contradiction with the former hypothesis. Thus, $X \cup$*subsume*(X) is a closed itemset.

Note that, in Algorithm 1, we use Theorem 1 to prune elements in index array. The reason is this operation can avoid the generation of duplicate initial generators.

4.2. The reduced pre-set and reduced post-set

In DCI-Closed algorithm, the sequences of order-preserving generators are obtained by checking whether the tidset of candidate generators are subsumed by the tidsets of items in pre-set and post-set. However, there are a lot of redundant elements in both pre-set and post-set especially when the datasets contain a large number of items. Therefore these checking operations are time-consuming sometimes. To reduce the sizes of pre-set and post-set, avoid redundant operations on checking tidset inclusion, we propose the following reduced pre-set and reduced post-set.

Definition 5. Given a generator of the form *gen*$= Y \cup i$, where Y is a closed itemset and $i \notin Y$, the reduced pre-set is defined as follows: *pre-set*$_R$(*gen*)=$\{i | i \in$*pre-set*(*gen*)$\wedge \nexists\ j \in$*pre-set*(*gen*), such that $j \in$*subsume*$(i)\}$.

Definition 6. Given a generator of the form *gen*$= Y \cup i$, where Y is a closed itemset and $i \notin Y$, the reduced post-set is defined as follows: *post-set*$_R$(*gen*)=$\{i | i \in$*post-set*(*gen*) $\wedge \nexists\ j \in$*pre-set*$_R$(*gen*), such that $j \in$*subsume*$(i)\}$.

We can replace the pre-set and post-set of DCI-Closed algorithm with proposed reduced pre-set and reduced post-set, because we have the following theorems.

Theorem 5. Under the framework of Index-CloseMiner, for a generator *gen*, its reduced pre-set (*pre-set*$_R$(*gen*)) retains the function of its original pre-set (*pre-set*(*gen*)).

Proof. Under the framework of Index-CloseMiner, for any $i \in$*pre-set*(*gen*), it is only used to check whether the candidate generator *gen* is order-preserving or not, i.e., whether g(*gen*)$\subseteq g(i)$ holds or not.

1) if $\exists i \in$*pre-set*(*gen*), such that g(*gen*)$\subseteq g(i)$, then

1. if $i \in$*pre-set*$_R$(*gen*), according to Definition 5, *pre-set*$_R$(*gen*)$\subseteq$*pre-set*(*gen*), so $i \in$ *pre-set*(*gen*). In this case, *pre-set*$_R$(*gen*) retains the function of *pre-set*(*gen*).
2. if $i \in$*pre-set*(*gen*)$\setminus$ *pre-set*$_R$(*gen*), according to Definition 5, $\exists j \in$*pre-set*(*gen*), such that $j \in$*subsume*(i), i.e., g(*gen*)$\subseteq g(i) \subseteq g(j)$. If $j \in$*pre-set*$_R$(*gen*), then *pre-set*$_R$(*gen*) retains the function of *pre-set*(*gen*); else $j \in$*pre-set*(*gen*)$\setminus$ *pre-set*$_R$(*gen*), according to Definition 5, $\exists j_1 \in$*pre-set*(*gen*), such that $j_1 \in$*subsume*(j), i.e., g(*gen*)$\subseteq g(i) \subseteq g(j) \subseteq g(j_1)$. We can iterate the process until $j_k \in$*pre-set*$_R$(*gen*) is found. In this case, *pre-set*$_R$(*gen*) retains the function of *pre-set*(*gen*).

2) If there exists no $i \in$*pre-set*(*gen*), such that g(*gen*)$\subseteq g(i)$, then since *pre-set*$_R$(*gen*)$\subseteq$*pre-set*(*gen*), there also exists no such item i in *pre-set*$_R$(*gen*). In this case, *pre-set*$_R$(*gen*) retains the function of *pre-set*(*gen*).

Theorem 6. Under the framework of Index-CloseMiner, for agenerator *gen*, its reduced post-set (*post-set*$_R$(*gen*)) retains the function of its original post-set (*post-set*(*gen*)).

Proof. Under the framework of Index-CloseMiner, for any $i \in$*post-set*(*gen*), it is mainly used to check whether *gen*$\cup i$ is an order-preserving generator that can be used to calculate FCI.

1. For any $i \in$*post-set*(*gen*), such that *gen*$\cup i$ is an order-preserving generator. It means that we can at least get a closed itemset by expanding *gen*$\cup i$. According to Theorem 3, $\nexists\ j \in$*pre-set*$_R$(*gen*$\cup i$), such that g(*gen*$\cup i$) $\subseteq g(j)$. Based on Definition 5 and Definition 6, we know that $i \in$*post-set*$_R$(*gen*). That is order-preserving generator *gen*$\cup i$ can also be obtained by using *post-set*$_R$(*gen*). In this case, *post-set*$_R$(*gen*) retains the function of *post-set* (*gen*).

2. For any $i \in$*post-set*(*gen*), such that *gen*$\cup i$ is not an order-preserving generator. It means that we can not get any closed itemset by expanding *gen*$\cup i$. According to Theorem 3, there at least exits an item $j \in$*pre-set*$_R$(*gen*), such that g(*gen*$\cup i$) $\subseteq g(j)$. Based on Definition 5 and Definition 6, we know that such item $i \notin$*post-set*$_R$(*gen*). Thus, the redundant operation on expanding *gen* by i, as well as the check on whether g(*gen*$\cup i$) is included in tidsets of items in *pre-set*$_R$(*gen*$\cup i$), are avoided. In this case, *post-set*$_R$(*gen*) retains the function of *post-set* (*gen*).

From the above definitions and theorems, it can be found that the reduced pre-set and post-set not only retain the functions of original pre-set and post-set, but also have smaller sizes. Thus, the redundant subset checks of tidsets between candidate generators and items in pre-set and post-set are reduced to great extents, especially for datasets with lots of items.

4.3. The Index-CloseMiner algorithm

After the above processing, we give the following Index-CloseMiner algorithm.

In Algorithm 2, if order-preserving, closed itemsets, composed of representative item and its subsume index, are output one by one (Steps 2–4). According to Definition 6, reduced post-set is calculated in Step 5. Iteration call is only executed when the support of closed itemset is higher than the support threshold *min_supp* (Steps 6–7). This is because we have the following Theorem 7. According to Definition 5, reduced pre-set is calculated in Step 8. For procedure *Closed*, according to certain order, the "minimal" item in reduced post-set, as well as its subsume index, are merged with the current closed itemset to form new generator (Steps 12–14). If the new generator is frequent and order-preserving, it is used as candidate closed itemset (Steps 15–16). The new reduced post-set is initialized in Step 17. Then, items in reduced post-set are processed one by one (Steps 18–22). For certain item i in reduced post-set, there are two possible cases: 1) add i to new closed itemset (Steps 19–20); 2) determine whether add i to reduced post-set or not (Steps 21–22). In Step 23, new closed itemset as well as its support are output. Similar to Step 5 and Step 6, iteration call is only executed when the support of closed itemset is higher than the support threshold *min_supp* (Steps 24–25). According to Definition 5, reduced pre-set is calculated in Step 26. Function *is_dup* is used to check whether the candidate new generator is order-preserving or not (Steps 29–33).

Theorem 7. Let X be a closed itemset with *supp*(X)=*min_supp*, then there exists no item $i \notin X \wedge i \in I$, such that $X \cup i$ is frequent closed itemset.

Proof. We prove the theorem by contradiction. Since g is monotonous decreasing, and $X \subset X \cup i$, we have $g(X \cup i) \subseteq g(X)$, i.e., *supp*($X \cup i$) $\leqslant$*supp*(X). Assume *supp*($X \cup i$)= *supp*(X), this means there exits a superset of X with the same support. It is in contradiction with the hypothesis that X is a closed itemset. Thus, *supp*($X \cup i$)<*supp*(X)= *min_supp*. Then $X \cup i$ is not frequent.

Theorem 8 (**Soundness**)**.** Index-CloseMiner enumerates all frequent closed itemsets.

Proof. Index-CloseMiner correctly identifies all and only the closed itemsets, since it improves DCI-Closed algorithm in the following three aspects:

Firstly, by exploiting the heuristic information provided by subsume index, only closed itemsets are used as initial order-preserving generators, thus, the number of initial generators are reduced greatly, especially when the datasets contain a large number of items. The new generators, obtained by using index array, will not change the function of DCI-Closed, because of the proof of Theorem 1 and Theorem 4.

Secondly, only reduced pre-set and reduced post-set are used in Index-CloseMiner rather than original ones used in DCI-Closed. Not only the sizes of reduced pre-set and reduced post-set are smaller than

Algorithm 2 Index-CloseMiner algorithm

Input: Index array after the processing of Algorithm 1

Output: All the frequent closed itemsets

```
1: for all index[j] in index[] do
2:     new_closure = index[j].item∪index[j].subsume;
3:     if ¬ is_dup(new_closure, pre-set_R) then
4:       Write Out new_closure and its support;
5:     Calculate post-set_R
6:     if supp(new_closure)≥min_supp then
7:       Closed(new_closure, pre-set_R, post-set_R);
8:   Calculate pre-set_R;
9: end for

10: procedure Closed(closed_set, pre-set_R, post-set_R)
11: while post-set_R ≠∅ do
12:   i ← min_ ( post-set_R);
13:   post-set_R ← post-set_R \ i;
14:   new_gen←closed_set∪i∪subsume(i);
15:   if (supp(new_gen)≥ min_supp) and ¬ is_dup(new_gen, pre-set_R) then
16:       closed_set_New ← new_gen;
17:       post-set_RNew←∅;
18:       for each item j in post-set_R do
19:           if g(new_gen) ⊆ g(j) then
20:               closed_set_New←closed_set_New∪j;
21:           else
22:               determine whether add j to post-set_RNew according to Definition 6;
23:       Write Out closed_set_New and its support;
24:       if supp(closed_set_New) > min_supp then
25:           Closed (closed_set_New, pre-set_R, post-set_RNew);
26:       Calculate pre-set_R;
27:   end if
28: end while

29: function is_dup (new_gen, pre-set_R)          //Duplicate check
30:   for each item j in pre-set_R do
31:       if g(new_gen)⊆ g(j) then
32:           return TRUE;
33:   return FALSE
```

the counter ones, but also they retain the function of original pre-set and post-set used in DCI-Closed, which is confirmed by Theorem 5 and Theorem 6.

Thirdly, the iteration call is only executed when the support of closed itemset is higher than the support threshold *min_supp*. The correctness of this operation is confirmed by Theorem 7.

According to the above discussions, it can be found that Index-CloseMiner retains the function of

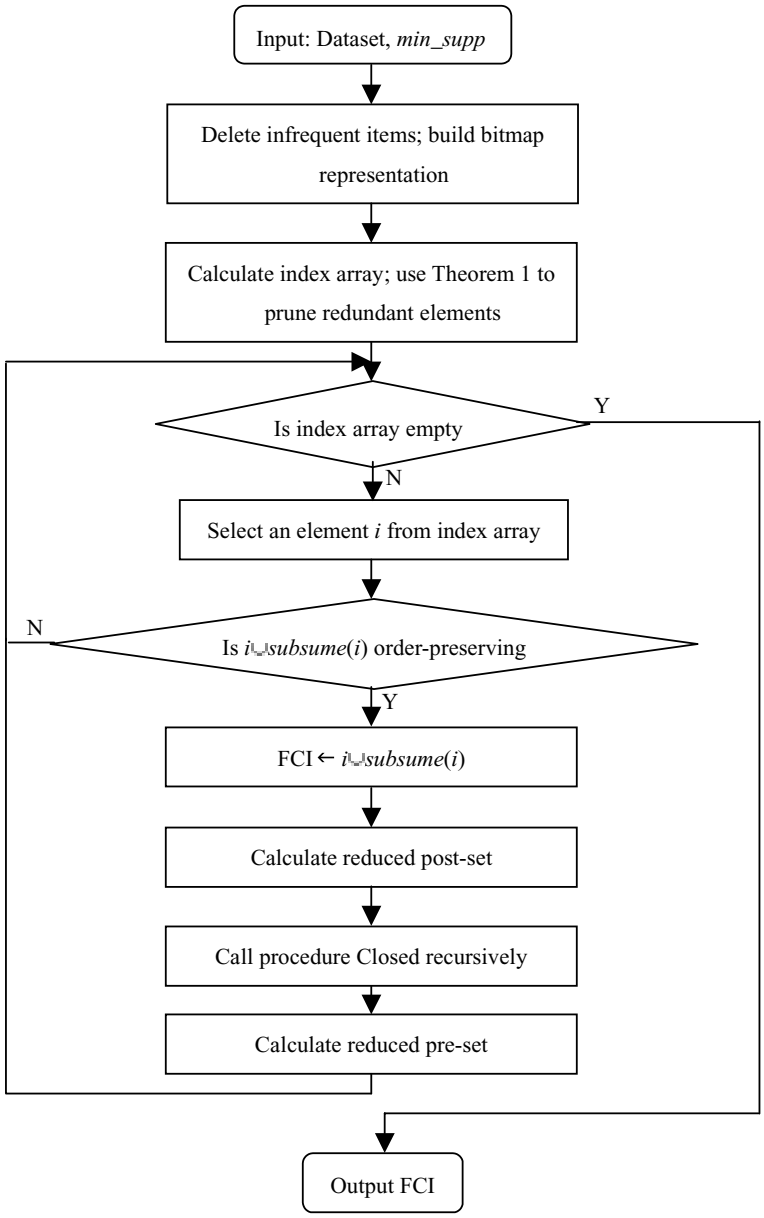


Fig. 3. The flow of information during the entire Index-CloseMiner process.

DCI-Closed. Since DCI-Closed is correct for mining FCI, Index-CloseMiner correctly identifies all and only the closed itemsets.

The flow of information during the entire process is shown in Fig. 3.

4.4. Comparison between Index-CloseMiner and DCI-Closed

Compared with DCI-Closed algorithm in [23,24], the advantages of Index-CloseMiner are listed as follows:

1. By using the heuristic information provided by subsume index, only closed itemsets are used as initial generators. Thus, the number of initial generators is reduced greatly especially for dense datasets. However, for DCI-Closed algorithm when $c(\emptyset) = \emptyset$, every item will be used as initial candidate generator. Thus, the search space is reduced greatly.
2. Only reduced pre-set and reduced post-set are used in Index-CloseMiner rather than original ones used in DCI-Closed. The sizes of reduced pre-set and post-set are smaller than original ones. It is also proved that the reduced pre-set and reduced post-set retain the function of original pre-set and original post-set. Thus, the redundant subset checks of tidsets between candidate generators and items in pre-set and post-set are reduced to great extent.
3. For Index-CloseMiner algorithm, the iteration call is only executed when the support of closed itemset is higher than the support threshold *min_supp*. So the times of iterations are reduced also.

Definition 7. Suppose there are totally n post-sets throughout the execution of Index-CloseMiner, we define *SizeSum*$= \sum_i |\mathrm{post} - \mathrm{set}_i|$.

SizeSum is the sum of cardinality of all the post-sets generated during the execution of Index-CloseMiner. To validate there are redundant elements in post-set, we will compare the *SizeSum* of post-set with *SizeSum* of reduced post-set in Section 5.3.

Example 2. We illustrate the basic idea of Index-CloseMiner via example database shown in Fig. 1.

FCI A is output at first. Call procedure *Closed*, pre-set$_R = \emptyset$, post-set$_R = \{B, D, E\}$, *SizeSum*=3. Then A is expanded by B and C, F in *subsume*(B). Since generator *ABCF* is order-preserving and frequent, it is examined that g(*ABCF*)$\subseteq g(D)$, and g(*ABCF*)$\subseteq g(E)$. So c(*ABCF*)=*ABCFD*, output *ABCFD* and its support. The iteration will not be called since *supp*(*ABCFD*)$= 2$=*min_supp*. Then post-set$_R$=$\{E\}$, *SizeSum*$= 3 + 1 = 4$. Generator *AE* is order-preserving, meanwhile the post-set$_R$ is empty, so output FCI *AE* and its support.

Return to the main procedure from procedure *Closed*. Merge initial generator B and C, F in *subsume*(B). pre-set$_R = \{A\}$, post-set$_R = \{E\}$, *SizeSum*=4+1=5. According to Definition 6, D will not be included in post-set$_R$. Since the tidset of *BCF* is not included in that of E in reduced post-set, output FCI *BCF* and its support. Call procedure *Closed*, Generator *BCFE* is order-preserving, meanwhile the reduced post-set is empty, so output FCI *BCFE* and its support.

Return to the main procedure from procedure *Closed*. Since A is in pre-ser$_R$ and $g(D) \subset g(A)$, D is ignored.

Return to the main procedure from procedure *Closed*. Output FCI E and its support. Call procedure *Closed*, pre-set$_R = \{A, B\}$, post-set$_R = \emptyset$. Since there is not any item in reduced post-set, this iteration ends the algorithm.

Figure 4 shows comparisons between Index-CloseMiner and DCI-Closed over the example database shown in Fig. 1. Note that the FCI of $\emptyset$ can be dealt separately by Index-CloseMiner.

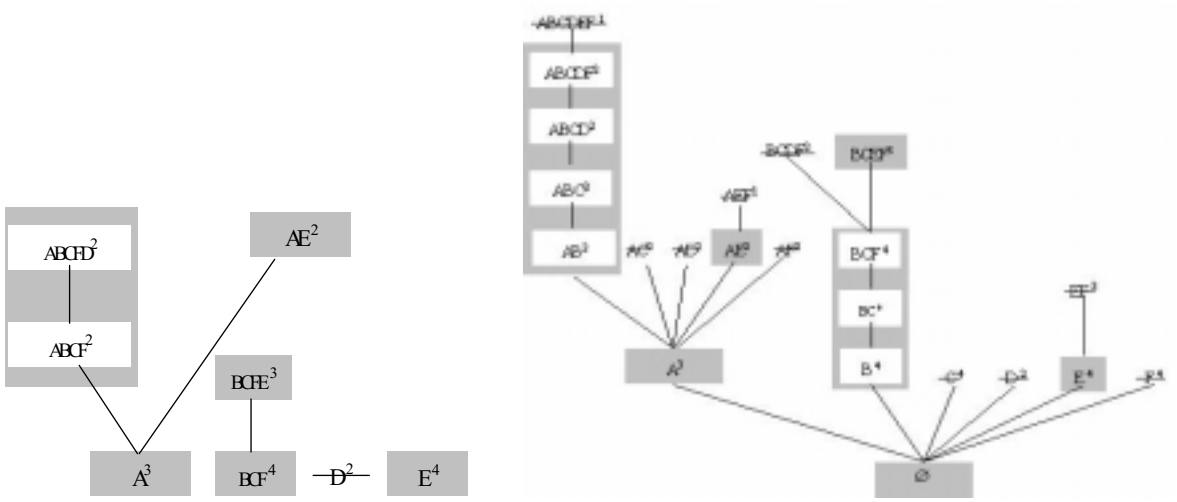
5. Performance evaluation

5.1. Test environment and datasets

We chose several real and synthetic datasets for testing the performance of Index-CloseMiner. All datasets are taken from the FIMI repository page http://fimi.cs.helsinki.fi. The Chess and Connect

Table 1
Characteristics of datasets used for experiment evaluations

Datasets	# Items	# Records	Avg. length
Chess	75	3,196	37
Connect	129	65,557	43
Mushroom	119	8,124	23
Pumsb	2113	49,046	74
Pumsb*	2088	49,046	50.5
T10I4D100K	870	100,000	11
T40I10D100K	942	100,000	40.5



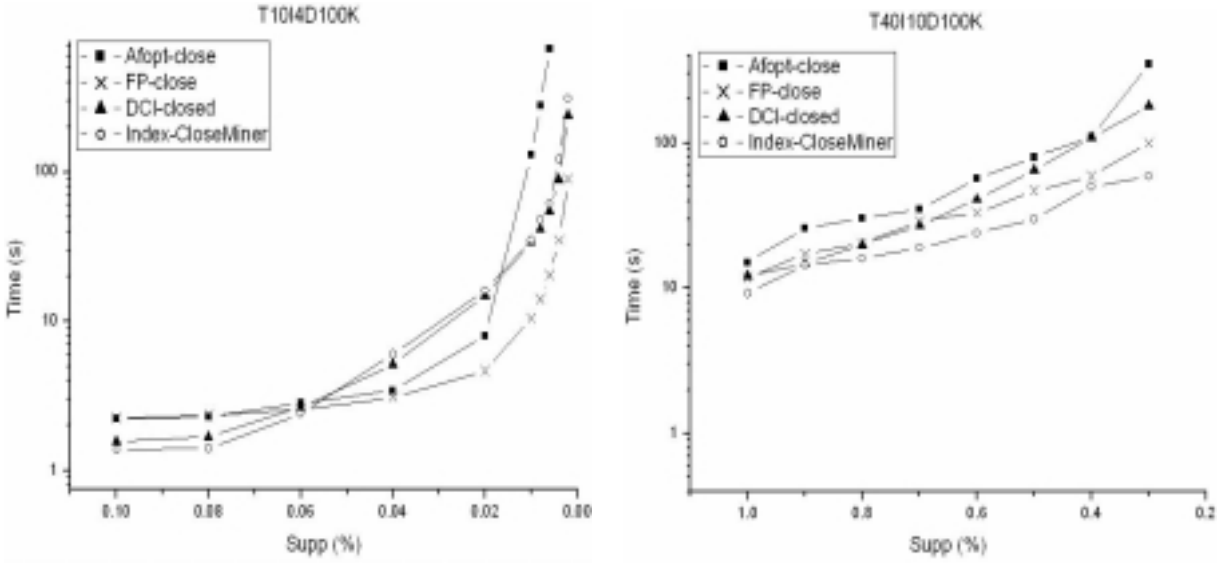
(a) Index-CloseMiner (b) DCI-Closed

Fig. 4. Index-CloseMiner verses DCI-Closed over the example database.

datasets are derived from their respective game steps. The Mushroom dataset contains characteristics of various species of mushrooms. While the Pumsb dataset contain census data. Pumsb* is the same as Pumsb without items with 80% or more support. Typically, these real datasets are very dense, i.e., they produce many long frequent itemsets even for very high values of support. We also chose a few synthetic datasets, which have been used as benchmarks for testing previous association mining algorithms. These datasets mimic the transactions in a retailing environment. Usually the synthetic datasets are sparse when compared to the real sets. Table 1 shows the characteristics of these datasets. The experiments were conducted on a Windows XP PC equipped with a 2.8 GHz Pentium IV and 512MB of RAM memory.

5.2. Performance comparisons

We compared the performances of Index-CloseMiner with DCI-Closed [23,24] and other two well-known state-of-the-art algorithms Afopt-Close [18] and FP-Close [19]. DCI-Closed, Afopt-Close and FP-Close are publicly available from the FIMI repository page http://fimi.cs.helsinki.fi. In this sets of experiments we confirmed the conclusion made at the FIMI 2003 workshop [25], that there are no clear winners with all databases. Indeed, algorithms that were shown to be winners with some databases were



(a) T10I4D100K (b) T40I10D100K

Fig. 5. Execution times on sparse datasets.

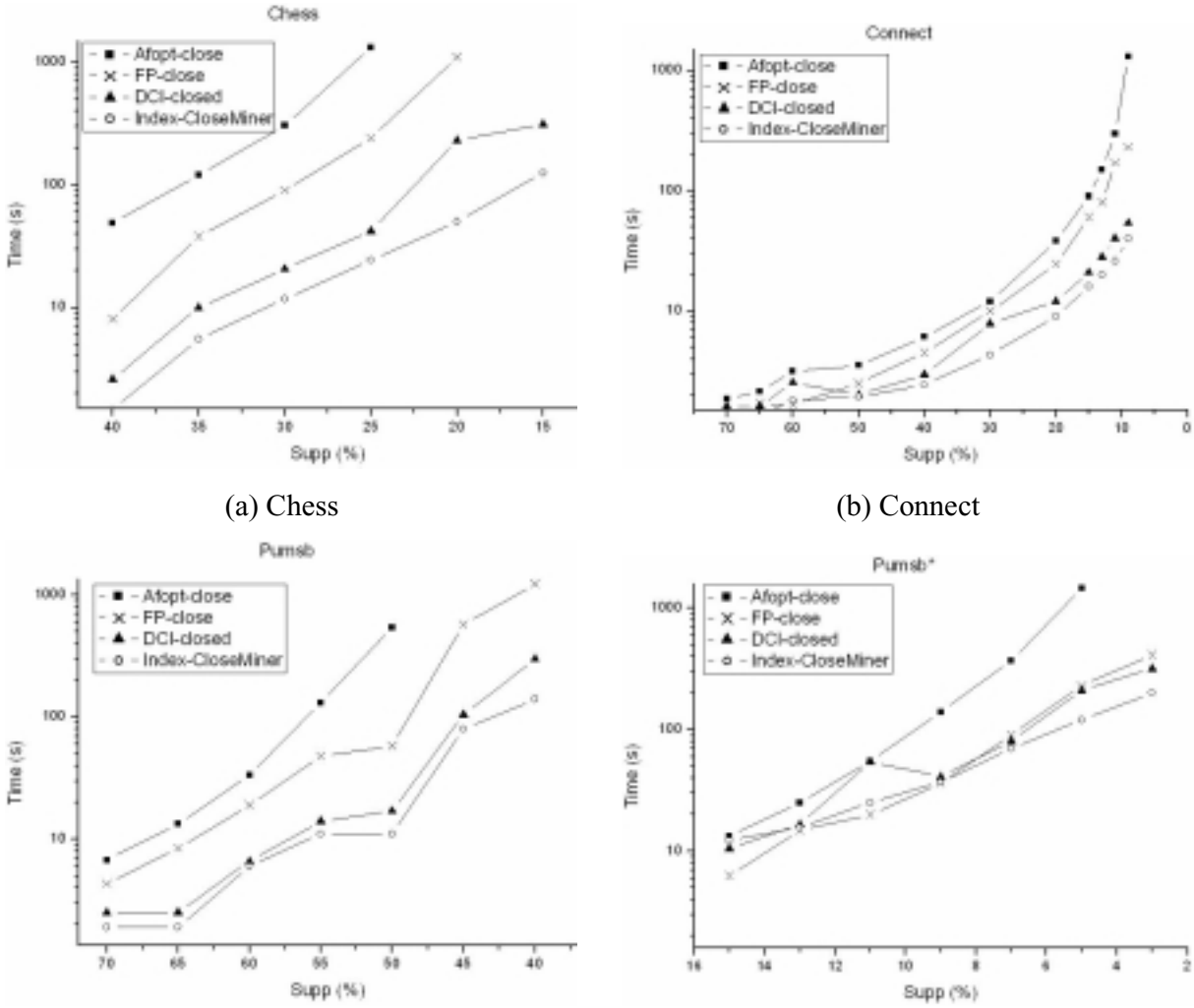
not the winners with others. Some algorithms quickly lose their lead once the support level becomes smaller. The recent survey on frequent closed itemsets algorithms also validates this claim [26].

All of the figures use total running time as the performance metric. Because all of the datasets are relatively small (the largest dataset is only 20MB), the time to load and prepare the data is negligible and, therefore, the total running time reflects the algorithmic performance only.

Figure 5 shows the results of comparing Index-CloseMiner with Afopt-Close, FP-Close and DCI-Closed on sparse data.

On T10I4D100K, Index-CloseMiner demonstrates the best performance of the four algorithms for higher supports. However, when support is lower than 0.06%, FP-Close passes Index-CloseMiner in performance to become the fastest algorithm. This is due to FP-Close profits from using an array-based technique to avoid traversing previously built FP-trees to construct the respective header tables of the new entries. And it is also interest to see that DCI-Closed always a bit faster than Index-CloseMiner. The main reason is that DCI-Closed uses a heuristic allowing to assess whether the dataset is dense or sparse. Using this information – the nature of the dataset – DCI-Closed launches a slightly modified version of the level-wise sweeping kDCI algorithm [27] in the case of a sparse dataset. Although it does not always work [26], this heuristic does always improve the performance of DCI-Closed on sparse datasets. Meanwhile, on this sparse dataset, of the highly increased number of frequent itemsets, few items appear together. Thus, the effect of using index array is not evident. The cost of generating index array leads to this lower speed of Index-CloseMiner. On T40I10D100K, Index-CloseMiner always demonstrates the best performance when support is lower than 1%.

The dense datasets in Fig. 6 support the idea that Index-CloseMiner runs the fastest on longer itemsets. For most supports on the dense datasets, Index-CloseMiner has the best performance. Index-CloseMiner runs around five to eight times faster than Afopt-Close on Connect, Pumsb, and Pumsb* and over five to 10 times faster on Chess. Index-CloseMiner outperforms FP-Close and DCI-Closed in most cases, but not all cases. In fact, as mentioned in [26], there is no outstanding algorithm to be qualified as the best



(a) Chess (b) Connect

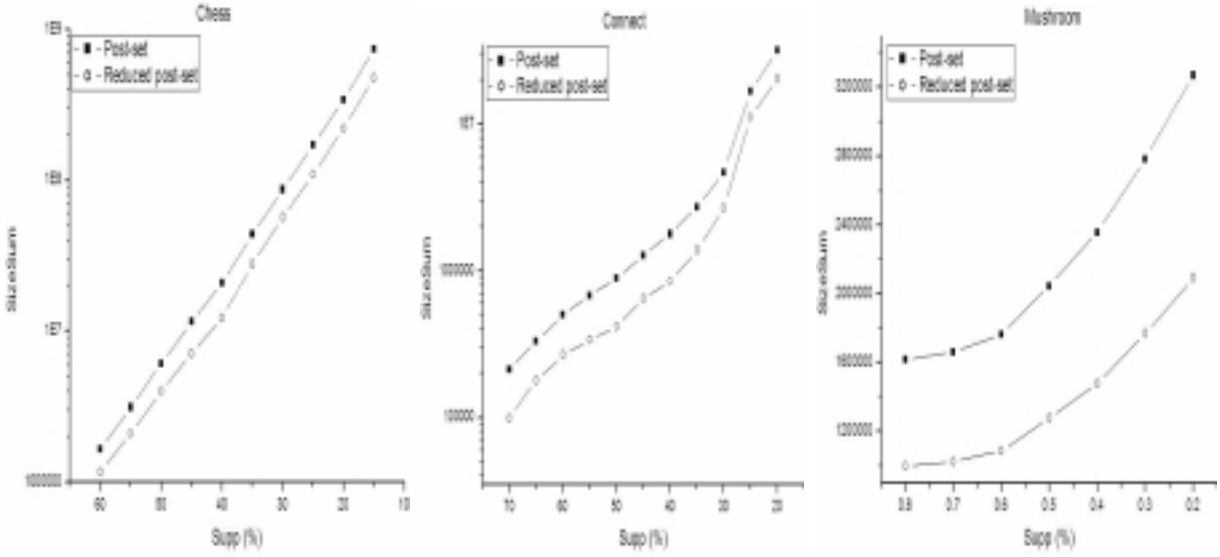
(c) Pumsb (d) Pumsb*

Fig. 6. Execution times on dense datasets.

for all datasets or at least for a given dataset type, i.e., dense or sparse. Moreover, in general, for a given dataset, there is no best algorithm for all *min_supp* values. Indeed, algorithm performances closely rely on *min_supp* values. A change in the *min_supp* value can lead to different information to be treated by the algorithm and so, an optimization or a heuristic that performs better for a given *min_supp* value can slow down performances due to this change.

5.3. The effect of reduced pre-set and reduced post-set

To validate there are redundant elements in both pre-set and post-set, we compare the number of elements in pre-set and post-set with the number of elements in counter reduced ones. From Algorithm 2, we can see that elements in pre-set are added gradually, and can be used throughout the whole execution process, so we can compare the cardinality of pre-set and that of reduced pre-set directly. While elements



(a) Chess (b) Connect (c) Mushroom

Fig. 7. Comparisons of the *SizeSum* of post-set with the *SizeSum* of reduced post-set.

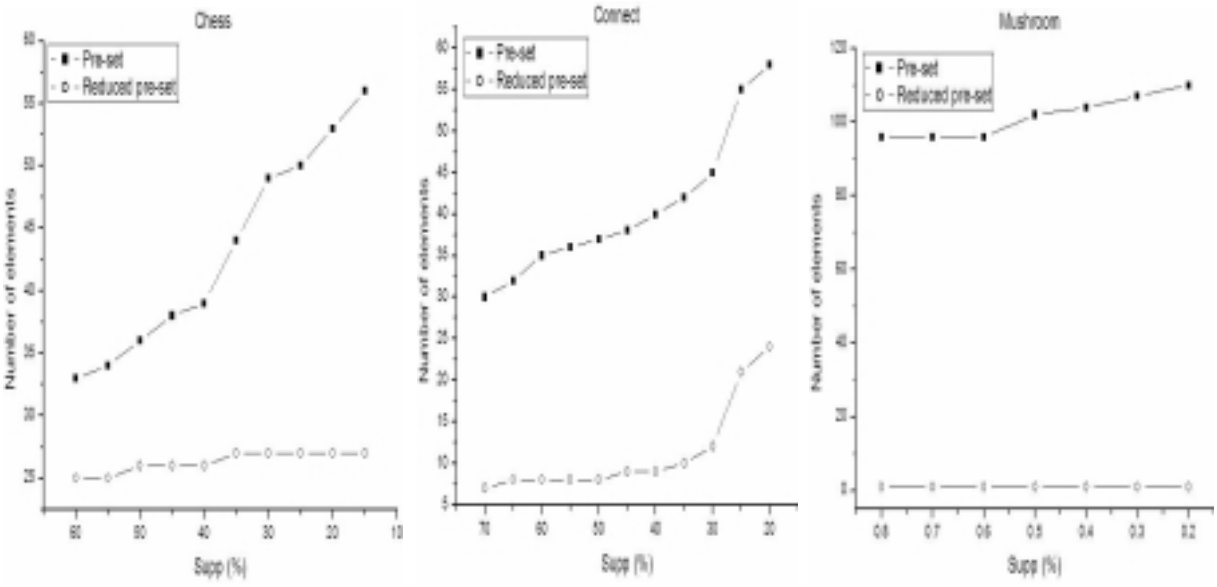
in post-set change dynamically with different closed itemsets, hence we add the size of post-set when a new post-set is constructed, that is *SizeSum* defined in Definition 7.

Figure 7 shows the results of comparing the *SizeSum* of post-set with the *SizeSum* of reduced post-set on Chess, Connect and Mushroom datasets. We can see from Fig. 7 that the *SizeSum* of reduced post-set can be much smaller than the *SizeSum* of post-set. For the support values we look at here, approximately we can get reductions in the cardinality with a factor of 2. That is nearly half of the elements in post-sets are redundant. And these redundant elements can be avoided in the proposed reduced post-set.

Figure 8 shows the results of comparing the size of pre-set with the size of reduced pre-set on Chess, Connect and Mushroom datasets. For Chess dataset, the reduced pre-set represents a reduction by a factor of 2 approximately of the items. On Connect dataset, reduced pre-set reduces the number of items by a factor varying from 2.8 to 4.5. On Mushroom dataset, we can get reductions ratio in the cardinality upto a factor of 110. It is interesting to see that there is only one element in the reduced pre-set. This is because the item occurs in all the transactions of Mushroom dataset. Thus, according to Definition 5, only this single item is enough for consisting the reduced pre-set.

6. Conclusion

In this paper, we have investigated the problem of efficiency in mining frequent closed itemsets from transactional datasets, and proposed an improved algorithm based on DCI-Closed. Index array, which is used for discovering those itemsets that always appear together, is introduced. Then, frequent items, which co-occur and share the same support, are merged together. Only closed itemsets are used as initial generators according to heuristic information provided by index array. Thus, the search space is reduced greatly. Based on index array, reduced pre-set and reduced post-set are proposed. It is proved that the reduced pre-set and reduced post-set retain the function of original pre-set and original post-set.



(a) Chess (b) Connect (c) Mushroom

Fig. 8. Comparisons of the size of pre-set with the size of reduced pre-set.

Thus, the redundant tidset-inclusion checks are avoided greatly. The experimental results show that the proposed algorithm is efficient especially on dense dataset.

Acknowledgements

The authors are indebted to the anonymous reviewers for their helpful comments and suggestions. This work is supported by the National Natural Science Foundation of China (60675030), by Funding Project for Academic Human Resources Development in Institutions of Higher Learning Under the Jurisdiction of Beijing Municipality, and by Key Youth Research Project of North China University of Technology.

References

[1] R. Agrawal, T. Imielinski and A. Swami, Mining associations between sets of items in massive databases, in *Proceedings of the 1993 ACM SIGMOD International Conference on Management of Data* (*SIGMOD'93*), 1993.
[2] P.A. Laur, J.E. Symphor, R. Nock and P. Poncelet, Statistical supports for mining sequential patterns and improving the incremental update process on data streams, *Intelligent Data Analysis* **11** (2007), 29–47.
[3] H. Mannila, H. Toivonen and A.I. Verkamo, Discovery of frequent episodes in event sequences, *Data Mining and Knowledge Discovery* **1** (1997), 259–289.
[4] A. Inokuchi, T. Washio and H. Motoda, Complete mining of frequent patterns from graphs: mining graph data, *Machine Learning* **50** (2003), 321–354.
[5] R. Agrawal and R. Srikant, Fast algorithms for mining association rules, in: *Proceedings of the 20th International Conference on Very Large Data Bases* (*VLDB'94*), 1994.
[6] F. Bodon, I.N. Kouris, C.H. Makris and A.K. Tsakalidis, Automatic discovery of locally frequent itemsets in the presence of highly frequent itemsets, *Intelligent Data Analysis* **9** (2005), 83–104.
[7] Y. Li and M. Kubat, Searching for high-support itemsets in itemset trees, *Intelligent Data Analysis* **10** (2006), 105–120.
[8] M.J. Zaki, Scalable algorithms for association mining, *IEEE Transactions on Knowledge and Data Engineering* **12** (2000), 372–390.

[9] J. Han, J. Pei, Y. Yin and R. Mao, Mining frequent patterns without candidate generation: a frequent-pattern tree approach, *Data Mining and Knowledge Discovery* **8** (2004), 53–87.
[10] K. Srikumar and B. Bhasker, Efficiently mining maximal frequent sets in dense databases for discovering association rules, *Intelligent Data Analysis* **8** (2004), 171–182.
[11] N. Pasquier, Y. Bastide, R. Taouil and L. Lakhal, Discovering frequent closed itemsets for association rules, in: *Proceedings of the 7th International Conference on Database Theory* (*ICDT'99*), 1999.
[12] M. J. Zaki, Mining non-redundant association rules, *Data Mining and Knowledge Discovery* **9** (2004), 223-248.
[13] N. Pasquier, R. Taouil, Y. Bastide, G. Stumme and Lotfi Lakhal, Generating a condensed representation for association rules, *Journal of Intelligent Information Systems* **24** (2005), 29–60.
[14] N. Pasquier, Y. Bastide, R. Taouil and L. Lakhal, Efficient mining of association rules using closed itemset lattices, *Information Systems* **24** (1999), 25–46.
[15] G. Stumme, R. Taouil, Y. Bastide, N. Pasquier and L. Lakhal, Computing iceberg concept lattices with TITANIC, *Data & Knowledge Engineering* **42** (2002), 189–222.
[16] J. Pei, J. Han and R. Mao, CLOSET: An efficient algorithm for mining frequent closed itemsets, in: *Proceedings of 2000 ACM SIGMOD Workshop on Research Issues in Data Mining and Knowledge Discovery* (*DMKD'00*), 2000.
[17] J. Y. Wang, J. Han and J. Pei, CLOSET+: searching for the best strategies for mining frequent closed itemsets, In *Proceedings of the Ninth ACM SIGKDD International Conference on Knowledge Discovery and Data Mining* (*KDD'03*), 2003.
[18] G.M. Liu, H.J. Lu, W.W. Lou, Y.B. Xu and J.X. Yu, Efficient mining of frequent patterns using ascending frequency ordered prefix-tree, *Data Mining and Knowledge Discovery* **9** (2004), 249–274.
[19] G. Grahne and J. Zhu, Efficiently using prefix-trees in mining frequent itemsets, in: *Proceedings of the ICDM 2003 Workshop on Frequent Itemset Mining Implementations* (*FIMI'03*), 2003.
[20] M.J. Zaki and C.J. Hsiao, CHARM: an efficient algorithm for closed itemset mining, in: *Proceedings of the Second SIAM International Conference on Data Mining* (*SDM'02*), 2002.
[21] N.G. Singh, S.R. Singh, A.K. Mahanta and B. Prasad, An algorithm for discovering the frequent closed itemsets in a large database, *Journal of Experimental and Theoretical Artificial Intelligence* **18** (2006), 481–499.
[22] T. Uno, T. Asai, Y. Uchida and H. Arimura, An efficient algorithm for enumerating closed patterns in transaction databases, in: *Proceedings of the 7th International Conference on Discovery Science* (*DS'2004*), 2004.
[23] C. Lucchese, S. Orlando and R. Perego, DCI_Closed: a fast and memory efficient algorithm to mine frequent closed itemsets, In *Proceedings of the ICDM 2004 Workshop on Frequent Itemset Mining Implementations* (*FIMI'04*), 2004.
[24] C. Lucchese, S. Orlando and R. Perego, Fast and memory efficient mining of frequent closed itemsets, *IEEE Transaction on Knowledge and Data Engineering* **18** (2006), 21-36.
[25] B. Goethals and M.J. Zaki, Advances in Frequent Itemset Mining Implementations Introduction to FIMI03, In *Proceedings of the ICDM 2003 Workshop on Frequent Itemset Mining Implementations* (*FIMI'03*), 2003.
[26] S.B. Yahia, T. Hamrouni and E.M. Nguifo, Frequent closed itemset based algorithms: a thorough structural and analytical survey, *SIGKDD Explorations* **8** (2006), 93–104.
[27] S. Orlando, C. Lucchese, P. Palmerini, R. Perego and F. Silvestri, kDCI: a multi-strategy algorithm for mining frequent sets, In *Proceedings of the ICDM 2003 Workshop on Frequent Itemset Mining Implementations* (*FIMI'03*), 2003.