第 30 卷 第 5 期 自 动 化 学 报 Vol. 30, No. 5
2004 年 9 月 ACTA AUTOMATICA SINICA Sept., 2004

Discovering Maximal Frequent Itemsequences Based on Suboperators of Itemsequence Sets and Data Partitioning[1)]

MAO Guo-Jun LIU Chun-Nian
(*Beijing Municipal Key Laboratory of Multimedia and Intelligent Software Technology, School of Computer Science, Beijing University of Technology, Beijing* 100022)
(E-mail: maoguojun@bjut.edu.cn)

Abstract Discovering frequent itemsets or itemsequences is an important phase in mining association rules. This paper presents two new algorithms for discovering frequent itemsequences called Dfis and Dfisp, which are based on suboperators of itemsequence sets and data partitioning techniques. Dfis is an algorithm with one-pass over databases and Dfisp is with two-pass over databases. Experimental results show that using suitable number of data partitioning, Dfisp could keep memory usage space within acceptable ranges.

Key words Data mining, association rules, itemsequences, suboperators

1 Introduction

Association rule mining is an important problem in the data mining. It was first introduced in 1993[1]. Let $\boldsymbol{I}=\{i_1, i_2, \cdots, i_m\}$ be a set of items. Let database $\boldsymbol{D}$ be a set of transactions where each transaction t is a set of items (called *itemset*) such that $t\subseteq\boldsymbol{I}$. Now considering an arbitrary itemset $I_1\subseteq\boldsymbol{I}$, the support of I_1 is defined as the percentage of transactions containing I_1 in $\boldsymbol{D}$ (*i.e.*, $support(I_1)=\|\{t\in\boldsymbol{D}|I_1\subseteq t\}\|/\|\boldsymbol{D}\|$). An association rule is an expression of the form $I_1\Rightarrow I_2$, where I_1 and I_2 are itemsets, $I_1\cap I_2=\phi$. The rule $I_1\Rightarrow I_2$ holds with $confidence(I_1\Rightarrow I_2)$ in $\boldsymbol{D}$, which is the percentage of transactions containing both I_1 and I_2 among those transactions containing I_1 (*i.e.*, $confidence(I_1\Rightarrow I_2)=support(I_1\cup I_2)/support(I_1)$). The problem of mining association rules is to find all rules that satisfy *minimum support* and *minimum confidence* constraints.

In general, the problem of mining association rules can be divided into two sub-processes: 1) Find all *frequent itemsets* which satisfy at least the minimum support; 2) Generate all association rules from the found frequent itemsets which must satisfy the minimum confidence. The first sub-process is more complex and challenging. The most popular mining theory is that all nonempty sub-itemsets of a frequent itemset must be frequent[1]. Based on such a theory, Apriori[1] was given as the classical algorithm in mining association rules. There have been some methods on improving the efficiency of Apriori, including Partition[2], DHP[3], Sampling[4]. However, most of the earlier work still follows the Apriori process (repeatedly scanning the database), so mining efficiency had been limited. Recently, there are some excellent work in reducing both the number of database passing and the size of candidate itemsets. Close[5] is an algorithm based on the new theory that all nonempty closed sub-itemsets of a frequent closed itemset must be frequent, which could be more efficient by pruning the closed itemset lattice. FP-Tree[6] is the first algorithm that mines frequent itemsets without candidate generation.

In this paper, we propose a novel solution to discovering frequent itemsequences through creating the operating theory of itemsequence sets, which can generate frequent

1) Supported by National Natural Science Foundation of P. R. China(60173014), Natural Science Foundation of Beijing(4022003), and Educational Foundation of Beijing (KZ0703200356)
Received April 21, 2003; in revised form April 8, 2004
收稿日期 2003-04-21; 收修改稿日期 2004-04-08

itemsequences by 1 or 2 passes over databases and without candidate generation.

2 Set of Itemsequences and its operators

In this paper, we use term *itemsequence* rather than itemset. In short, an itemsequence is an ordered list of items. Certainly, a tuple of the transaction database can be characterized as an itemsequence as well as an itemset. However, the comparison between itemsequences can be easier than between itemsets. If $\mathbf{D}=\{d_1, d_1, \cdots, d_n\}$ be a transaction database that every tuple contains an itemsequence, then we may simply see such a database as a set of itemsequences. Therefore, the problem of discovering frequent itemsets may be transformed into processing between itemsequences. In this paper, we introduce the lattice framework of the itemsequence set and generate the set of frequent itemsequences through the operators on an itemsequence set. There have recently been more interests in mining maximal frequent patterns from databases[9,10]. A *maximal frequent itemsequence* is such a frequent itemsequence that cannot been contained by any other frequent itemsequences[10]. This paper focuses on discovering an efficient and novel method to mine the set of maximal frequent itemsequences.

Example 1. Let $SIS_1=\{AB, CD\}$ and $SIS_2=\{ABCD, AD\}$, then itemsequence $AB\in SIS_1$ and $AB\notin SIS_2$; $\{AB\}\subset SIS_1$; $\{AB\}\not\subset SIS_2$; $SIS_1\cup SIS_2=\{AB, CD, ABCD, AD\}$; $SIS_1\cap SIS_2=\varnothing$.

In fact, AB is a subsequence of $ABCD$ in SIS_2 though $AB\notin SIS_2$. This fact can be useful to discovering relations between itemsequences, so we should pay closer attention to these relations. For this sake, we first give the suboperators on an itemsequence set.

Definition 1 (Suboperators on sets of itemsequences). Let IS be an *itemsequence*. Let SIS_1 and SIS_2 be two *itemsequence sets* defined in $\mathbf{I}$. Then

1) IS *sub-belongs* to SIS_1 if $\exists IS_1\in SIS_1$: $IS\subseteq IS_1$, denoted by $IS\in_{sub}SIS_1$.
2) SIS_2 *sub-contains* SIS_1 if $\forall IS_1\in SIS_1$: $IS_1\in_{sub}SIS_2$, denoted by $SIS_1\subseteq_{sub}SIS_2$.
3) *Sub-intersection* of SIS_1 and SIS_2 define as $SIS_1\cap_{sub}SIS_2=\{IS \mid IS\in_{sub}SIS_1$ and $IS\in_{sub}SIS_2\}$.
4) *Sub-union* of SIS_1 and SIS_2 define as $SIS_1\cup_{sub}SIS_2=\{IS \mid IS\in_{sub}SIS_1$ or $IS\in_{sub}SIS_2\}$.

Example 2. Consider SIS_1 and SIS_2 as the same as example 1, then itemsequence $AB\notin SIS_2$ but $AB\in_{sub}SIS_2$.

These suboperators characterize such hidden relations within itemsequence sets that cannot be found by typical set operators. They may be used to find potential relations in itemsequences.

3 Discovering Frequent Itemsequences Based on Suboperators

In this section, we employ two sets in the memory called SIS and SIS^* to record related sets of itemsequences. The notation is given in Table 1.

Table 1 Notation

Names	Contains
SIS	The set of itemsequences obtained by scanning the database
SIS^*	The set of frequent itemsequences produced
Sup_count(IS)	The support count of itemsequence IS

3.1 Dfis algorithm

Fig. 1 gives the pseudo-codes of Dfis and its subprocedures. In Algorithm 1, each of iterations is related with a tuple of the database and consists of three phases. First, an itemsequence(called *IS*) is extracted from a tuple of the database. Next, *IS* is tried to enter *SIS* and its support count may be recalculated. Finally, SIS^* is updated through *IS* and new *SIS*. As *Produce_IS(d, IS)* is easy to be implemented, its discussion is omitted here.

3.1.1 Join(*IS*, *SIS*)

This process puts *IS* into *SIS* and initializes its support count if it has not been in *SIS*, or recalculates its support counts if it has been in *SIS*. Algorithm 2 gives its pseudo-code.

3.1.2 Make_fre(*IS*, *SIS*, SIS^*, *minsup_count*)

Join(*IS*, *SIS*) may change the supports of elements in *SIS*, so it is possible that new frequent itemsequences are produced. Make_fre(*IS*, *SIS*, SIS^*, *minsup_count*) tries to find frequent sub-itemsquences of *IS*. Algorithm 3 gives its pseudo-code.

In algorithm 3, *Prune*(IS^*, SIS^*) and *Prune*(IS^*, *SIS*) are called. As was stated above, we only put all maximal frequent itemsequences into SIS^*. If IS^* is frequent, it is reasonable to prune the sub-itemsequences of IS^* in SIS^* to get better performance. Also, when IS^* is frequent, its sub-itemsequences are pruned from *SIS* because they no longer need to be recorded. Algorithm 4 gives the process to delete an itemsequence and its sub-itemsequences from an itemsequence set.

```
Algorithm 1. (Dfis Algorithm)
1) Input(minsup_count);
2) SIS←∅; SIS*←∅;
3) FOR all d∈D DO BEGIN
4)    Produce_IS(d, IS);
4)    Join(IS, SIS);
5)    Make_fre(IS, SIS, SIS*, minsup_count);
6) END
7) Answer←SIS*.

Algorithm 2. Join(IS, SIS)
1) Sup_count (IS)=1; flag=0;
2) FOR all IS1∈SIS DO
3)    IF IS=IS1 BEGIN
4)       Sup_count (IS1)=Sup_count (IS1)+1;
5)       flag=1;
6)    END;
7) IF flag=0 SIS=SIS ∪ {IS};

Algorithm 3. Make_fre (IS, SIS, SIS*, minsup_count)
1) FOR all IS*∈sub{IS} DO BEGIN
2)    Sup_count (IS*)=0;
3)    FOR all IS**∈SIS DO
4)       IF IS*⊆IS** Sup_count(IS*)=
             Sup_count(IS*) +Sup_count(IS**);
5)    IF Sup_count(IS*)>minsup_count
6)       IF(IS*∉subSIS*)BEGIN
7)          Prune(IS*, SIS*);//see Algorithm 4
8)          SIS*=SIS*∪{IS*};
9)       END
10)   Prune(IS*, SIS);//see Algorithm 4
11) END;

Algorithm 4. Prune(IS1, SIS1)
1) FOR all IS2∈SIS1 DO
2) IF IS2∈sub{IS1} SIS1=SIS1-{IS2};
```

Fig. 1 Dfis Algorithm and its subprocedures

3.2 Example and experiments about Dfis algorithm

Table 2 gives a sample of transaction databases. Let us try to discover its maximal frequent itemsequences through Dfis Algorithm. Table 3 shows the execution of Dfis with minimum support count 2 on the sample database.

Table 2 Sample database

TID	Itemsequences
1	*A*, *B*, *C*, *D*
2	*B*, *C*, *E*
3	*A*, *B*, *C*, *E*
4	*B*, *D*, *E*
5	*A*, *B*, *C*, *D*

Table 3 Discovering frequent itemsequences with Dfis

	IS	*SIS*	SIS^*	Note
0		∅	∅	
1	*ABCD*	{(*ABCD*,1)}	∅	
2	*BCE*	{(*ABCD*,1),(*BCE*,1)}	{*BC*}	
3	*ABCE*	{(*ABCD*,1),(*BCE*,1),(*ABCE*,1)}	{*BCE*}	Deleted *BC* in SIS^*
4	*BDE*	{(*ABCD*,1),(*ABCE*,1),(*BDE*,1)}	{*BCE*, *BD*}	Deleted *BCE* in *SIS*
5	*ABCD*	{(*ABCD*,2),(*ABCE*,1),(*BDE*,1)}	{*BCE*, *ABCD*}	Deleted *BD* in SIS^*
Res		{(*ABCE*,1),(*BDE*,1)}	{*BCE*, *ABCD*}	Deleted *ABCD* in *SIS*

Obviously, Dfis is an one-pass mining algorithm to databases, so it has the same I/O costs as MAFIA[9], but less than Apriori[1], Close[5] and FP-Tree[6]. For one-pass mining algorithms, they would suffer from main memory problems if any effective measures are not taken. The memory usage of Dfis algorithm is dominated by SIS and SIS^* expenses. Theoretically speaking, SIS would take $O(\|\boldsymbol{D}\|)$ memory space in the worst case; and SIS^* is exponentially growing with $\|\boldsymbol{I}\|$, where $\boldsymbol{I}$ is the set of all items in $\boldsymbol{D}$. In fact, Dfis algorithm prunes in time sub-sequences of the maximal frequent itemsequences that have generated in SIS and SIS^*. With such pruning techniques, we may drastically reduce the number of elements in SIS^*, and control the size of SIS as possible. In order to assess relative performances of memory usage, we implemented Dfis algorithm and conducted some experiments. The first experiment about Dfis was on a series of databases with the sizes from 10K to 100K. If minsupport is 20%, the memory spaces of SIS and SIS^* with increasing sizes of the databases are shown on Fig. 2(a). The second experiment was done on a database with 100KB using different minsupports. Fig. 2(b) shows the experimental results.

These results tell us that the SIS^* is stable with increasing the sizes of databases and the numbers of minimum supports. However, we must pay more attention to the increasing sizes of SIS with growing the sizes of databases.

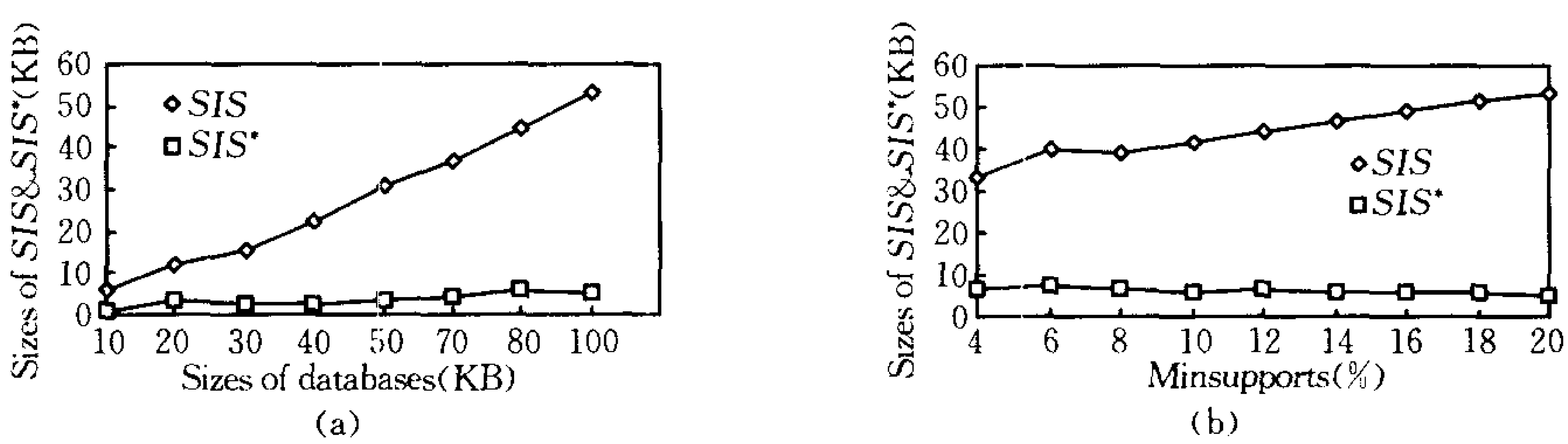

Fig. 2 Sizes of SIS and SIS^* on different databases and minsupports about Dfis

4 Improving Dfis algorithm by partitioning data

4.1 Dfisp algorithm

Dfisp algorithm partitions the database into data segments that are small enough to be effectively handled in memory. For each segment, locally frequent itemsequences are achieved by applying the above Dfis method to this segment. Dfisp algorithm consists of two passes over the database. The first pass generates all locally frequent itemsequences through iteratively calling Dfis to all segments. The collection of all locally frequent itemsequences becomes candidates for frequent itemsequences on the whole database. The second pass over the database produces global frequent itemsequences through testing supports of these candidates.

Theorem 1. Let a database $\boldsymbol{D}$ be partitioned into n nonoverlapping segments $\boldsymbol{D}_1, \boldsymbol{D}_2, \cdots, \boldsymbol{D}_n$, and global minimum support count be *minsup_count*. The local minimum support count for each segment $\boldsymbol{D}_i$, called $minsup\_count_i\ (i=1,2,\cdots,n)$, is formed by $minsup\_count_i = minsup\_count * \|\boldsymbol{D}_i\| / \|\boldsymbol{D}\|$. An itemsequence cannot be frequent in $\boldsymbol{D}$ with *minsup_count* if it is not frequent in any $\boldsymbol{D}_i$ with $minsup\_count_i\ (i=1,2,\cdots,n)$.

Proof. Let $sup\_count_i(IS)$ be the support count of itemsequence IS in $\boldsymbol{D}_i$. If IS is not frequent in any $\boldsymbol{D}_i$ with $minsup\_count_i\ (i=1,2,\cdots,n)$, *i.e.*,

$$\forall i=1,2,\cdots,n:\ sup\_count_i(IS) < minsup\_count_i$$

Then the support count of IS in $\boldsymbol{D}$, $sup\_coun\ (IS)$, should be the sum of all $sup\_count_i(IS)$ in $\boldsymbol{D}_i\ (i=1,2,\cdots,n)$, *i.e.*,

$$sup\_count\ (IS) = \sum sup\_count_i(IS) < \sum minsup\_count_i = \sum (minsup\_count^* \|\boldsymbol{D}_i\| / \|\boldsymbol{D}\|) = minsup\_count\ ^*(\sum \|\boldsymbol{D}_i\|) / \|\boldsymbol{D}\| = minsup\_count^* \|\boldsymbol{D}\| / \|\boldsymbol{D}\| = minsup\_count$$

Therefore *IS* is not frequent in **D**. □

In order to conveniently be called in Dfisp, algorithm Dfis is rewritten Algorithm 1*. In Algorithm 1*, *SIS* is first cleared, but *SIS** is not cleared to keep all local frequent itemsequences. Algorithm 5 gives the description of Dfisp which calls Dfis($\boldsymbol{D}_i$, *minsup_count*$_i$, *SIS**).

Algorithm 1* (Dfis($\boldsymbol{D}_i$, *minsup_count*$_i$, *SIS**))

1) *SIS* ← ∅;
2) FOR all $d \in \boldsymbol{D}_i$ DO BEGIN
3) Produce_IS(d, *IS*);
4) Join(*IS*, *SIS*);
5) Make_fre(*IS*, *SIS*, *SIS**, *minsup_count*$_i$);
6) END;

Algorithm 5 (Dfisp Algorithm)

1) Input(n, *minsup_count*);
2) *SIS** ← ∅;
3) Patition_DB($\boldsymbol{D}, \boldsymbol{D}_1, \boldsymbol{D}_2, \cdots, \boldsymbol{D}_n$);
4) FOR $i=1$ to n DO BEGIN
5) *minsup_count*$_i$ = *minsup_count** $\| \boldsymbol{D}_i \| / \| \boldsymbol{D} \|$;
5) Difs($\boldsymbol{D}_i$, *minsup_count*$_i$, *SIS**);
6) END
7) FOR all *IS** ∈ *SIS** support(*IS**) = 0;
8) FOR all $d \in D$ DO BEGIN
9) Produce_IS(d, *IS*);
10) FOR all *IS** ∈ *SIS**
11) IF *IS** $\in_{\text{sub}}$ {*IS*} support(*IS**) = support(*IS**) + 1;
12) END
13) FOR all *IS** ∈ *SIS**
14) IF support(*IS**) < *minsup_count* *SIS** = *SIS** − {*IS**};
15) Answer ← *SIS**.

Fig. 3 Modified Dfis procedure and Dfisp algorithm

4.2 Experiments about Dfisp algorithm

After using the partitioning technique, Dfisp makes the number of itemsequences that are recorded in memory to be reduced, so the size of *SIS* can be acceptable for large databases. We conducted an experiment on the different databases whose results are shown in Fig. 4(a). In this experiment, global minsupport is 20%, each of the analyzed databases is partitioned into the 5 segments with the same size. Comparing with the results that Fig. 2 (a) shows, *SIS* of Dfisp needs less memory space of than Dfis.

In order to further test the efficiency of Dfisp algorithm, we conducted some experiments on a database of 1MB. We first fixed the partitioning number to 5 and the database is divided into the segments of the same sizes. Fig. 4 (b) shows the changes of *SIS* and *SIS** in memory space with different minsupports. We can observe that Dfisp takes relative stable memory spaces with different minsupports.

Then, We fixed the minsupport to 5%, and tracked the execution time on the database with different the partitioning numbers. Our experimental computer is Pentium III with 256M RAM. The results are shown in Fig. 4(c). In fact, an optimized partitioning number exists for a specific database. By optimized partitioning technique, necessary memory space is cut down and the whole execution time can also be controlled within an acceptable ranges.

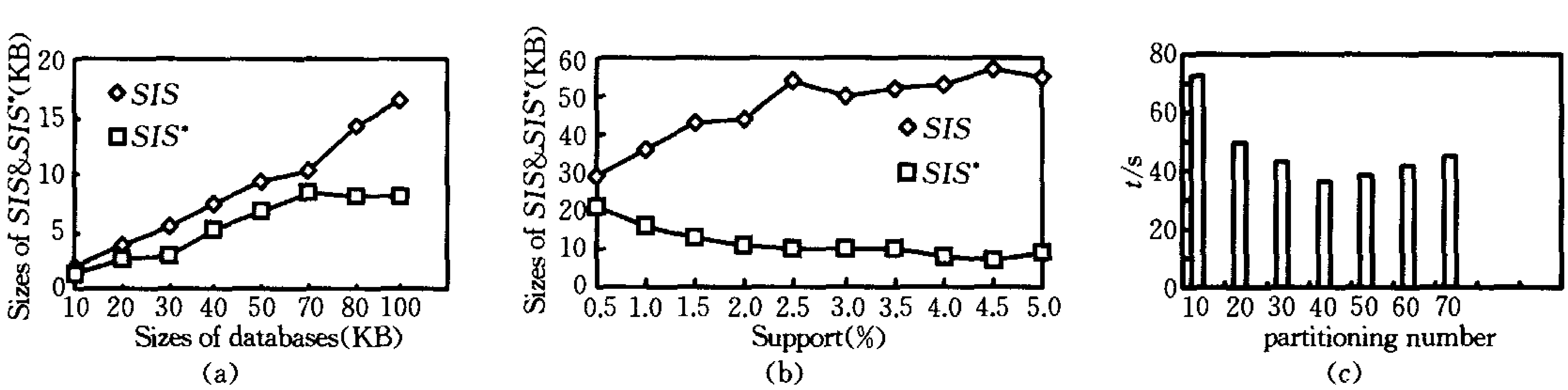

Fig. 4 Performance of Dfisp in memory and execution time

5 Conclusion

We presented the algorithms, called Dfis and Dfisp, for efficiently discovering maximal frequent itemsequences. Dfis is based on the operating theory on set of itemsequences. Unlike most existing algorithms, it does not need to repeatedly scan databases. It only employs one pass over the database. Dfisp is an improvement to Dfis by data partitioning which makes memory usage space to be controlled and CPU overhead to be lightened in large databases. We conducted a serial of experiments to evaluate Dfis and Dfisp algorithms. Experimental results showed that Dfisp is an efficient algorithm in execution time by using an optimized partitioning number.

References

1 Agrawal R, Imielinski T, Swami Arun N. Mining association rules between sets of items in large database. In: Proceedings of the ACM SIGMOD International Conference on Management of Data, Washington, DC: ACM Press, 1993, 207～216

2 Savasere A, Omiecinski E, Navathe S. An efficient algorithm for mining association rules in large databases. In: Proceedings of the 21st VLDB Conference, Zurich, Switzerland: Morgan Kaufman Publishers, 1995, 432～444

3 Park J S, Chen M S, Yu P S. An effective hash-based algorithm for mining association rules. In: Proceedings of the ACM SIGMOD, San Jose, USA: ACM Press, 1995, 175～186

4 Toivonen H. Sampling large databases for mining association rules. In: Proceedings of the 22nd VLDB Conference, Bombay, India: Morgan Kaufmann Publishers, 1996, 134～145

5 Pasquier N, Bastide Y, Taouil R, Lakhal L. Efficient mining for association rules using closed itermset lattices. *Information Systems*, 1999, **24**(1): 25～46

6 Han J W, Jian Pei J, Yin Y. Mining frequent patterns without candidate generation. In: Proceedings of the SIGMOD Conference, Dallas, Texas, USA: ACM Press, 2000, 1～12

7 Han J W, Pei J, Mortazavi A B, Chen Q, Dayal U, Hsu M. FreeSpan: Frequent pattern-projected sequential patterns mining. In: Proceedings of the KDD Conference, Boston, MA, USA: ACM Press, 2000, 355～359

8 Manmila H. Methods and problems in data mining. In: Proceedings of the 6th International Conference on Database Theory, Delphi, Greece: Springer-Verlag, 1997, 41～45

9 Doug B, Calimlim M, Gehrke J. MAFIA: A maximal frequent itemset algorithm for transactional databases. In: Proceedings of the 17th International Conference on Data Engineering, Heidelberg, Germany: ICDE Society, 2001, 443～462

10 Karam G, Zaki M J. Efficiently mining maximal frequent itemsets. In: Proceedings of the IEEE International Conference on Data Mining, San Jose: IEEE Computer Society, 2001, 163～170

MAO Guo-Jun Received his Ph. D. degree from Beijing University of Technology, P. R. China. His research interests include data mining and distributed system.

LIU Chun-Nian Professor of Beijing University of Technology. Received his Ph. D. degree from Norwegian University of Science and Technology. His research interests include AI, knowledge engineering, and data mining.

基于项目序列集亚操作和数据分割的最大频繁项目序列挖掘方法

毛国君　　刘椿年

(北京市多媒体与智能软件重点实验室,北京工业大学　北京　100022)

(E-mail: maoguojun@bjut.edu.cn; ai@bjut.edu.cn))

摘　要　发现频繁项目序列集是关联规则挖掘中的一个重要步骤.该文提出两个发现最大频繁项目序列的算法 Dfis 和 Dfisp. Dfis 算法基于项目序列集操作理论,只有一次数据库扫描. Dfisp 是 Dfis 的改进算法,它引入数据分割技术以提高内存使用率因而增强对大型数据库的处理能力,是一个两次数据库扫描算法.实验表明了它们的性能和优势.

关键词　数据挖掘,关联规则,项目序列,亚操作

中图分类号　TP311