一种基于有效修剪的最大频繁项集挖掘算法

陈 鹏 吕卫锋

(北京航空航天大学 计算机学院, 北京 100083)

摘 要: 对关联挖掘中的最大频繁项集挖掘问题进行了研究,提出了一种基于项集格修剪机制的最大频繁项集挖掘算法. 采用项集格生成树的数据结构,将最大频繁项集挖掘过程转化为对项集格生成树进行深度优先搜索获取所有最大频繁节点的过程. 其中提高算法效率的一个重要措施是在遍历项集格生成树的过程中对生成树进行修剪. 给出了项集格生成树的三个性质,并在此基础上提出了直接超集修剪、间接超集修剪与事务集等价修剪三种修剪机制,尽可能忽略非频繁节点及其所生成的扩展节点以减少遍历的节点数目. 试验结果表明,三种修剪机制都能够有效地减少搜索空间,其中事务集等价修剪机制的效果最好,算法的性能与输入数据集的稠密程度相关.

关 键 词: 数据挖掘; 关联规则; 关联挖掘; 格

中图分类号: TP 311

文献标识码: A **文 章 编 号**: 1001-5965(2006)02-0218-06

Maximal frequent itemsets mining algorithm based on effective pruning mechanisms

Chen Peng Lü Weifeng

(School of Computer Science and Technology, Beijing University of Aeronautics and Astronautics, Beijing 100083, China)

Abstract: The maximal frequent itemsets mining problem was studied and an algorithm based on pruning itemset lattice effectively was proposed. The itemset lattice tree data structure was adopted to translate maximal frequent itemsets mining into the process of depth-first searching the itemset lattice tree. One of the key measures to promote performance of the algorithm is to prune the itemset lattice tree while traversing it. Three properties of itemset lattice tree were given and three pruning mechanisms, direct superset pruning, indirect superset pruning and transaction sets equivalence pruning, were proposed based on them respectively to prune the infrequent nodes and their extension nodes to reduce the number of nodes while traversing the itemset lattice tree. Test results indicate that all the three pruning mechanisms can reduce the search space effectively and the transaction sets equivalence pruning has the best effect on performance of the algorithm. Test results also indicate that performance of the algorithm is related to denseness of the datasets.

Key words: data mining; association rule; association mining; lattice

关联规则问题是数据挖掘研究领域中极其重要的一个问题,Agrawal 等最早给出了关联规则模型与支持度-置信度框架[1,2],并提出了挖掘关联规则的 AIS 算法[1],主要用于分析数据库中数据项的相互关系. AIS 算法被用于发现定量性的规则,但该算法只能生成有一个后继的关联规则. Houtsma 和 Swami 提出 SETM 关联性算法[3]. SETM 关联性算法中频繁序列的项为<TID, itemset>, 其中 TID 是每个事务的唯一标号,itemset 是每个事务中包括的项集. SETM 关联性算法可以

收稿日期: 2005-01-10

基金项目: 国家重点基础研究发展规划资助项目(G1999032709);国家自然科学基金资助项目(90104008)

作者简介: 陈 鹏(1974-),男,湖南郴州人,博士生, pchen@nlsde.buaa.edu.cn.

通过 SQL 语言的连接操作对前一次的频繁项集进行扩展,来产生新的候选项集. 但该算法的缺点是候选者集合的数目过大. AIS 算法和 SETM 算法都需要通过数据库中事务数据的项集对前一次的频繁项集进行扩展,来生成候选项集集合,很明显这样会产生并统计很多不是频繁的候选项集.

Agrawal 和 Srikant 提出了 Apriori 算法[2],该算法是关联规则挖掘算法中最为经典的算法. Apriori 算法通过前一次发现频繁集合,来生成新的候选项集,这样可以不必考虑数据库中的事务数据. 而且 Apriori 算法依据的基本思想是频繁项集的任何一个子集都是频繁项集,可以删除那些包含子集不是频繁项集的候选项集,这样可以大量地减少候选项集的数目. Zaki 和 Ogihara[4] 提出基于形式概念分析的格理论[5] 来构建关联规则的理论框架,通过该理论框架可以计算出所有关联规则的规则生成集.

关联规则的挖掘是一个两阶段的过程. 第一阶段必须找到所有频繁项集的集合 FI(the set of Frequent Itemtests);第二阶段由频繁项集产生强关联规则. 实际上挖掘过程中最耗时的是第一阶段. 因此,有人提出采用比 FI 更小同时对第二阶段来说具有足够的信息的替换,其中最为典型的是频繁项闭集的集合 FCI(the set of Frequent Closed Itemsets)与最大频繁项集的集合 MFI(the set of Maximal Frequent Itemsets). 显然有下列关系存在:MFI⊆FCI⊆FI. 这样,挖掘 MFI 就成为关联规则挖掘的核心问题.

现有的频繁项集挖掘算法基本上可以划分为两类:宽度优先搜索算法与深度优先搜索算法. 经典的宽度优先搜索算法有 Apriori[2,6],对项集格采用的是从下至上逐层搜索的方法. (k+1)-候选项集只能从 k-项集的频繁项集中产生. 通过扫描数据库来获取每一层的所有候选项集的频率. Agrawal 等提出压缩进一步迭代扫描的事务数的方法[6,7]. 因为不包含任何 k-项集的事务,不可能包含任何(k+1)-项集,可对这些事务加上删除标志,扫描数据库时不再考虑. Apriori 类方法要求多次扫描数据库,即如果频繁项集最多包含 10 个项,那么就需要扫描数据库 10 遍,这需要很大的 I/O 负载. 可能产生大量的候选项集,以及可能需要重复扫描数据库,是 Apriori 算法的两大不足.

MaxMiner[8] 是另一种典型的宽度优先算法,主要用于挖掘 MFI. 为了减少搜索空间,MaxMiner 采用了子集修剪,即忽略有非频繁子集的候选项集,其主要不足是修剪机制简单.

DepthProject 算法[9] 也是一种 MFI 挖掘算法,其主要特点是对项集格的搜索采用了一种深度优先/宽度优先相混合的策略. 算法同时采用了子集修剪与动态记录子节点的方法,因而不是完全的深度优先遍历. 通过动态的记录,可以修剪位于每个节点尾部以外的非频繁项,减小了搜索空间. DepthProject 算法主要存在的问题是修剪机制的效率尚待提高.

FP-stream 算法[10]、FP-max 算法[11] 与 FPMFI 算法[12] 都是基于频繁模式树 FP-tree(Frequent Pattern tree)的 MFI 挖掘算法,其共同特点是采用频繁模式树 FP-tree 的数据结构,并通过删除 FP 子树(conditional frequent pattern tree)的冗余信息,有效地压缩 FP 子树的规模,以减少遍历的开销.

本文设计了一种基于有效修剪机制的 MFI 挖掘算法,其核心思想是:采用项集格的数据结构,对项集格进行深度优先搜索,采用有效的修剪机制以裁减非频繁节点及其所生成的扩展节点.

1　项集格的相关预备知识

R. Wille 于 1982 年提出了格理论[5],Rymon 于 1992 年提出了简化项集格以构造按词典字母顺序子集树的方法[13],被称为项集格的概念框架. 此方法被 Agarwal 与 Bayardo 分别用于 DepthProject 算法[9] 与 MaxMiner 算法[8] 的设计. 项集格的定义如下.

定义 1　设数据库中存在项集 I 的按词典字母顺序的全序 $\leqslant_L$. 若项 i 位于项 j 之前,记为 $i \leqslant_L j$. 这种排序可以用来构造项集格或者项集 I 的幂集 S 之上的偏序. 定义偏序 $\leqslant_{on}$,使得对 $S_1, S_2 \in S$,若有 $S_1 \subseteq S_2$,则 $S_1 \leqslant_{on} S_2$. 若 $S_1 \not\subset S_2$,则 S_1 与 S_2 在偏序 $\leqslant_{on}$ 中无任何关系.

图 1 中给出了由 $\{a,b,c,d\}$ 4 个项所构造的完全项集格. 格的根节点为空集(该层记为 0 层),第 k 层的所有节点包括了所有的 k-项集. 第 k 层各节点所对应的 k-项集按照词典字母顺序进行排序,所有的子节点由前一层的父节点子集按照词典字母顺序生成. 这样就将项集格简化为按词典字母顺序子集树,称为项集格生成树.

定义 2　对于任意项集格生成树中的节点 N,定义其所对应的项集为节点 N 的 head.

定义 3　对于任意项集格生成树的节点 N,定义字母表中所有按词典字母顺序比其 head 中任意项都要大的所有项构成的集合为节点 N 的

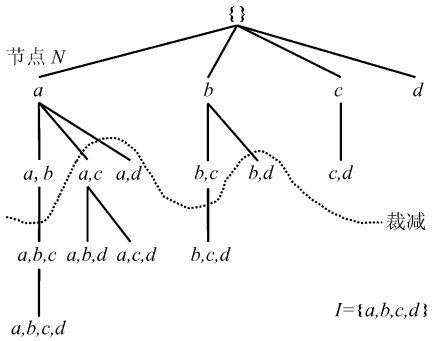


图 1 由 4 个项所构造的完全项集格

tail.

定义 4 对于任意项集格生成树中的节点 N,定义以节点 N 为根节点的子树中所对应项集为最大的节点是以节点 N 为根节点的最大扩展节点,而其所对应项集是以节点 N 为根节点的最大扩展项集.

从定义 2 可以看出,项集格生成树中节点 N 的 tail 中的所有元素实际上是节点 N 对应项集的 1-扩展项. 从定义 3 很容易可以得到,以节点 N 为根节点的最大扩展项集 = 节点 N 的 head ∪ 节点 N 的 tail.

项集格生成树构造完毕后,挖掘频繁项集的问题其实就可以被视为如何找到对该项集格生成树进行一次有效的裁减使得所有位于该裁减之上的项集为频繁项集,而位于该裁减之下的项集为非频繁项集. 上述裁减之上的频繁项集组成了一个'正边界',而之下的非频繁项集组成了一个'负边界'. 在没有使用修剪机制的情况下,对项集格生成树的遍历只需计算正边界及其下一层的所有项集的支持度 s. 例如在图 1 中,除了 $\{a,b,c,d\}$ 与 $\{a,c,d\}$ 之外的所有项集都需要被遍历并且计算其支持度.

2 项集格生成树的修剪机制

采用有效的修剪机制对挖掘算法非常重要,因为它可以减少搜索空间与计算量,缩短搜索时间. 本文针对项集格生成树的特点,提出项集格生成树的三个性质,分别基于这三个性质来形成三种不同的项集格生成树修剪机制.

定理 1 对 ∀ 节点 $N \in$ 项集格生成树,若以节点 N 为根节点的子树中的最左节点 N. leftmost _ extension 所对应的项集 N. leftmostc _ extension _ itemset 是频繁项集,则可判定以节点 N 为根节点的子树中除节点 N. leftmost _ extension 外的节点的对应项集都是非最大频繁项集.

证明 首先根据项集格生成树的深度优先搜索性质,可以推断节点 N. leftmost _ extension 是以节点 N 为根节点的子树中的最大扩展节点. 若项集 N. leftmost _ extension _ itemset 是频繁项集,则根据频繁项集的任意子集也是频繁项集推断出其任意子集不可能为最大频繁项集,因此不需要搜索 N. leftmost _ extension _ itemset 的任意子集,这些子集即是以节点 N 为根节点的子树中的所有节点,因此可以对该子树进行修剪. **证毕**

基于性质 1 所形成的项集格生成树修剪机制被称为直接超集修剪,其核心思想是判断当前节点对应项集的最大扩展项集是否频繁项集,如果是,则可以裁减当前节点的扩展节点中除最大扩展节点之外的所有节点.

定理 2 对 ∀ 节点 $N \in$ 项集格生成树,设以节点 N 为根节点的子树中的最左节点为节点 N. leftmost _ extension,其所对应的项集为 N. leftmost _ extension _ itemset. 若当前 MFI 中已有 N. leftmost _ extension _ itemset 的超集存在,则可判定以节点 N 及其所有扩展节点对应的项集都是非最大频繁项集.

证明 当前 MFI 中已有 N. leftmost _ extension _ itemset 的超集存在,因此可推出:①N. leftmost _ extension _ itemset 是频繁项集;②N. leftmost _ extension _ itemset 不是最大频繁项集. N. leftmost _ extension _ itemset 是频繁项集,则根据定理 1 的结论,可以推断出以节点 N 为根节点的子树中除节点 N. leftmost _ extension 外的节点的对应项集都是非最大频繁项集. 因此可以判定以节点 N 及其所有扩展节点对应的项集都是非最大频繁项集. **证毕**

基于性质 2 所形成的项集格生成树修剪机制被称为间接超集修剪,其核心思想是判断是否在当前 MFI 中已有当前节点对应项集的最大扩展项集的超集存在,如果是,则可以裁减当前节点的所有扩展节点.

定理 3 对 ∀ 节点 $N \in$ 项集格生成树,设 e 为节点 N 对应的项集,f 为节点 N 的一个 1-扩展项. 定义 $t(e)$ 为包含项集 e 的所有事务的集合. 若 $t(e) \subseteq t(f)$ 成立,则可从节点 N 的 tail 中将 f 移出至其 head.

证明 若 $t(e) \subseteq t(f)$ 成立,即任意包含 e 的事务也包含 f. 从而可以推出,任意包含 e 但不包含 f 的频繁项集 g 必有频繁超集 $g \cup \{f\}$. 由于只对最大频繁项集感兴趣,因此没有必要对包含 e 但不包含 f 的项集进行计算. 于是可以将项 f 从节

点 N 的 tail 中移至其 head,即 $e=e\cup\{f\}$ 且从节点 N 的 tail 中删除 f. **证毕**

称基于性质 3 的修剪机制为事务集等价修剪,其核心思想是对当前节点与所生成的 1-扩展子节点的所属事务集进行比较,如果相等,则可以削减当前节点的 tail,并将所对应的 1-扩展项 f 从当前节点的 tail 移至 head 中.

3　MFI _ Mining 算法

基于上述有关项集格生成树修剪机制的讨论,提出综合采用三种修剪机制来对项集格生成树搜索进行优化,在此基础上提出了一种深度优先搜索算法 MFI _ Mining. MFI _ Mining 算法的修剪机制与前述的 Max _ Miner 算法、DepthProject 算法的不同之处在于:后两者采用了子集修剪,MFI _ Mining 算法采用了两种超集修剪,DepthProject 算法采用了动态记录子节点机制,MFI+Mining 算法采用了事务集等价机制. 虽然动态记录子节点机制与事务集等价机制都是从节点的 tail 中精简 1-扩展项,但是事务集等价机制可以实现在正边界中对项集格生成树的非频繁子树进行裁剪,因此其效率比动态记录子节点机制要高.

MFI _ Mining 算法的要点如下:

1) 生成 1-扩展子节点的过程是按照词典字母顺序进行的深度优先搜索过程;

2) 通过计算 $N.\ \text{head}\cup\{k\}$ 的支持度来判断所生成的 1-扩展子节点是否频繁节点;

3) 若所生成的 1-扩展子节点不是频繁节点,则没有沿该路经继续生成子树的必要,因为 Apriori 算法中已经指出,若 $A\cup B$ 是频繁项集,则可以证明其任意子集也是频繁项集. 而对非频繁节点的任意扩展显然不可能是频繁节点;

4) 若对当前节点的所有 1-扩展都未成功产生频繁 1-扩展子节点,那么当前节点是一个叶节点;

5) 当判定当前节点为叶节点时,可以确定该节点对应项集为一个 MFI 候选项集,但还不能立即确定该节点项集是否为最大频繁项集,必须在当前的 MFI 中检查是否有该节点项集的超集存在,如果没有,则可以将该叶节点候选项集加入至当前 MFI;而另一方面,却不需要考虑是否当前 MFI 中有当前叶节点候选项集的子集,因为已经在当前 MFI 中的项集是按照词典字母顺序所产生的;

6) 沿当前节点的最左分支路径计算最左叶节点的 head,即为当前节点的最大扩展项集,在此基础上对项集格生成树进行超集修剪;

7) 因为深度优先搜索算法本身就是沿每个节点的最左路径进行搜索,因此不需要额外的操作,只需要加上 is _ leftmost _ child 这个布尔变量对最左子节点的位置进行标记即可.

基于上述考虑,综合采用三种项集格生成树修剪机制的深度优先搜索算法 MFI _ Mining 如下:

```
MFI _ Mining(Current node N, MFI, Boolean is _ leftmost _ child)
begin
if (N. max _ extension _ itemset 已经在当前 MFI 中) then 停止搜索并返回.
计算生成当前节点 N 的所有 1-扩展子节点.
基于事务集等价修剪机制对当前节点 N 的 tail 进行精简.
For each item k in 当前节点精简后的 tail
begin
if (k is 当前 tail 中最小的 1-扩展项)
then is _ leftmost _ child = TURE
else is _ leftmost _ child = FALSE.
NewNode=N. head∪{k}.
MFI _ Mining(NewNode,MFI, is _ leftmost _ child)
end.
if (is _ leftmost_child and 当前节点的所有 1-扩展子节点的对应项集是频繁项集) then
begin
停止搜索.
回退至上一次 is_leftmost_child 从 FALSE 变为 TRUE 时所记录的节点处.
end.
if(当前节点是叶节点且其对应项集不在当前 MFI 中) then 将当前节点的对应项集加入当前 MFI 中.
end.
```

4　试验分析

从两个方面对 MFI _ Ming 算法的性能进行了试验:①给定输入数据集,考察三种修剪机制对算法全局性能的影响;②给定输入数据集,与 Depthproject 算法进行性能比较.

试验所采用的计算环境为:2.8 GHz Pentium (R) 4 处理器,1 GHz 内存,Redhat9.0 Linux 操作系统. 应用程序采用 C++编译器进行编译. 试验选择三类数据集作为 MFI _ Mining 算法的测试输入数据集. 其中一类为天然稠密数据集,包括

Dataset1、Dataset2、Dataset32 与 Dataset4;第二类为由 IBM Almaden Synthetic Data Generator 所生成的标准合成稀疏数据集,包括 Dataset5 与 Dataset6;第三类是由 Blue Martini 所提供的有关电子商务网站访问情况的天然稀疏数据集[14],包括 Dataset7、Dataset8 与 Dataset9. 这 9 个数据集的具体指标如表 1 所示.

表 1 MFI 算法测试输入数据集的指标

数据集	事务数	项数	事务平均长度(项集数)
Dataset1	3 196	76	37
Dataset2	67 557	130	43
Dataset3	49 046	7 117	74
Dataset4	49 046	7 117	50
Dataset5	100 000	1 000	10
Dataset6	100 000	1 000	40
Dataset7	515 597	1 657	6.53
Dataset8	59 602	497	2.51
Dataset9	3 340	16	4.62

4.1 项集格修剪机制性能的相关试验

试验选择 *Dataset2* 为测试输入数据集,所采用的用户定义最小支持度阈值为 40%. 试验结果如表中 2 所示.

表 2 修剪机制对算法全局性能的影响

算法	搜索生成节点数	算法执行时间/s
基本算法	341 515 351	8 420.69
直接超集修剪	7 523 905	172.38
间接超集修剪	4 470 976	100.98
事务集等价修剪	847 393	19.89
直接+间接超集修剪	4 429 952	100.74
直接超集+事务集等价修剪	409 699	9.78
间接超集+事务集等价修剪	102 711	2.65
同时采用三种修剪	96 825	2.46

表 2 中的试验结果表明:

1) 三种修剪机制能够有效地减少项集格生成树的搜索空间;

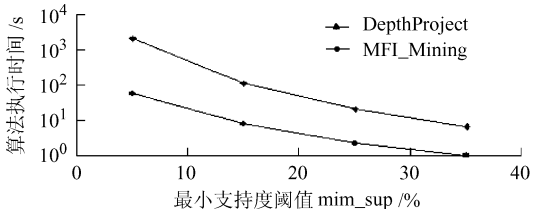
2) 直接超集修剪与间接超集修剪同属于超集修剪机制,只是实现的方法不同,因此对算法的影响比较类似,差距不大;

3) 事务集等价机制对于减少项集格生成树搜索空间最为有效. 其主要原因是,算法的执行时间主要耗费在对项集格生成树的正边界中频繁节点的搜索过程,而三种修剪机制中事务集等价机制在正边界中对项集格生成树的非频繁子树进行裁剪的可能性最大.

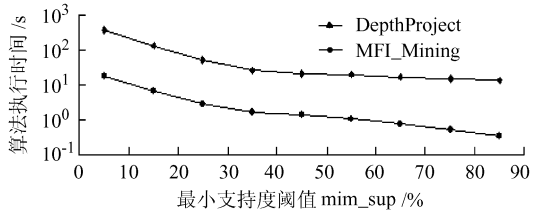
4.2 与 Depthproject 算法的性能比较试验

试验结果如图 2 ~ 图 4 所示.

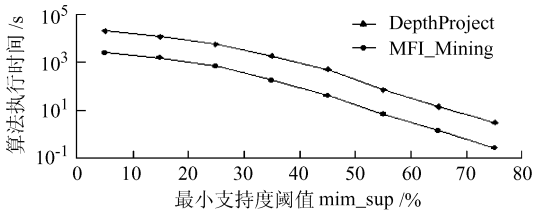
图 2 中的结果显示,MFI _ Mining 算法在对稠密数据集进行处理时执行速度最快. min _ sup 取(0,1)中的任意值,MFI _ Mining 算法都比 DepthProject 算法的性能要好. min _ sup 取较低值



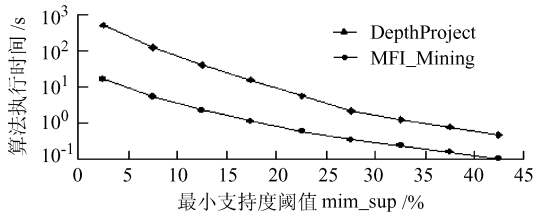
a 输入数据集 Dataset1



b 输入数据集 Dataset2

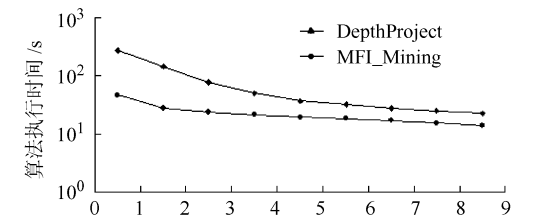


c 输入数据集 Dataset3

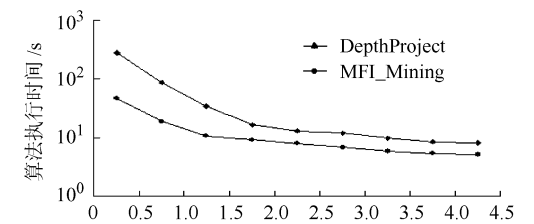


d 输入数据集 Dataset4

图 2 输入稠密数据集时的算法性能

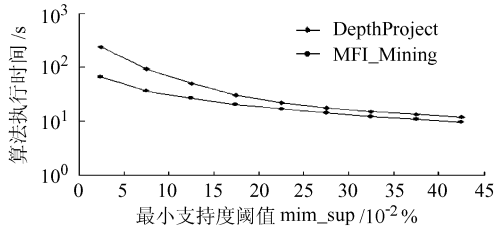


a 输入数据集 Dataset5

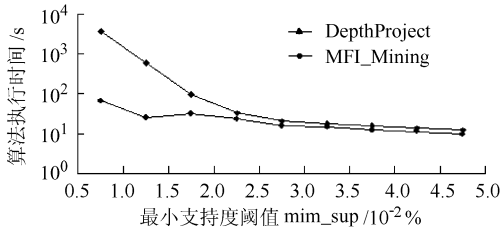


b 输入数据集 Dataset6

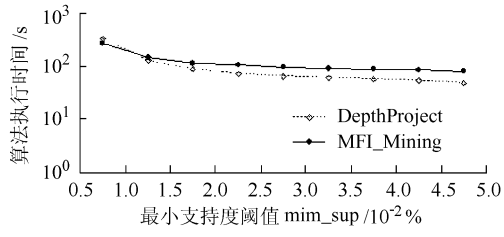
图 3 输入合成稀疏数据集时的算法性能



a　输入数据集 Dataset7



b　输入数据集 Dataset8



c　输入数据集 Dataset9

图 4　输入天然稀疏数据集时的算法性能

时,MFI _ Mining 算法比 DepthProject 算法要快 30 倍左右;min _ sup 取较高值时,MFI _ Mining 算法比 DepthProject 算法要快 10 倍左右.

图 3 中的试验结果显示,在对稀疏程度较高的标准合成稀疏数据集进行处理时,MFI _ Mining 算法始终比 DepthProject 算法要快. min _ sup 取较低值时,MFI _ Mining 算法比 DepthProject 算法要快 20 倍左右;min _ sup 取较高值时,MFI _ Mining 算法比 DepthProject 算法要快 2 倍左右.

图 4 中的试验结果显示,在对稀疏程度最高的天然稀疏数据集进行处理时,虽然 MFI _ Mining 算法比 DepthProject 算法要快,但是差距已经不很明显,当 min _ sup 取较大值时,两者的算法执行时间已经非常接近. 试验 9 的结果显示,对数据集 Dataset9 进行处理时,MFI _ Mining 算法的性能不太理想. 在 min _ sup 取<0.01 的较低值时,MFI _ Mining 算法比 DepthProject 算法要快,而 min _ sup 取>0.01 的较高值时,MFI _ Mining 算法甚至比 DepthProject 算法要慢.

综合以上试验结果分析,可以得出以下结论:对于事务平均长度较大的稠密数据集,MFI _ Mining 算法的性能优势更为明显一些;而当事务平均长度较小时,数据集的稀疏程度较高,此时 MFI _ Mining 算法的性能优势不明显.

5　结 束 语

本文提出了一种基于有效修剪机制的最大频繁项集挖掘算法 MFI _ Mining. 采用项集格的数据结构,对项集格生成树进行深度优先搜索,并基于有效的修剪机制裁减项集格生成树中的所有非频繁节点及其所生成的扩展节点. 试验结果表明:与已有的经典算法 Apriori、MaxMiner 和 DepthProject 等相比较,MFI _ Mining 既避免了生成大量的候选项集,又提高了项集格生成树修剪机制的效率.

致谢　感谢许可博士对本文的建议.

参考文献 (**References**)

[1] Agrawal R, Imielinski, Swami A, *et al*. Mining association rules between sets of items in large databases[A]. In: Peter Buneman, Sushil Jajodia, eds. Proceedings of the 1993 ACM SIGMOD International Conference on Management of Data[C]. Washington, 1993. 207~216

[2] Agrawal R, Srikant R. Fast algorithms for mining association rules in large database[R]. FJ9839, 1994

[3] Houtsma M, Swami A. Set-oriented mining of association rules [A]. In: Philip S Yu, Arbee L P Chen. eds. Proceedings of the 11th International Conference on Data Engineering[C]. Taipei, 1995. 25~33

[4] Zaki M, Ogihara M. Theoretical foundations of association rules [A]. In: 3rd ACM SIGMOD Workshop on Research Issues in Data Mining and Knowledge Discovery[C]. Washington, 1998. 7.1~7:8

[5] Wille R. Restructuring lattice theory: an approach based on hierarchies of concepts[A]. In: Ivan Rival, ed. Ordered Sets[C]. Reidel, Dordrecht-Boston, 1982. 445~470

[6] Agrawal R, Srikant R. Fast algorithms for mining association rules in large database[A]. In: Jorge B Bocca, Matthias Jarke Carlo Zaniolo, eds. Proceedings of the 20th International Conference on Very Large Data Bases[C]. Santiago, 1994. 487~499

[7] Han J, Fu Y. Discovery of multiple-level association rules from large databases[A]. In: Jorge B Bocca, Matthias Jarke, Carlo Zaniolo, eds. Proceedings of 21th International Conference on Very Large Data Bases[C]. Zurich, rland, 1995.39~46

[8] Bayardo R J. Efficiently mining long patterns from databases [A]. In: Proceedings of the 1998 ACM SIGMOD International Conference on Management of Data[C]. Seattle, Washington, 1998. 85~93

(下转第 243 页)