维普资讯 http://www.cqvip.com

一种直接在Trans-树中挖掘频繁模式的新算法*)

范明 王秉政
(郑州大学计算机科学系 郑州450052)

A Novel Algorithm for Mining Frequent Patterns Directly in Trans-Tree

FAN Ming WANG Bing-Zheng
(Department of Computer Science, Zhengzhou University, Zhongzhou 450052)

Abstract Frequent pattern mining plays an essential role in many important data mining tasks. FP-growth is a very efficient algorithm for frequent pattern mining. However, it still suffers from creating conditional FP-tree separately and recursively during the mining process. In this paper, we propose a new algorithm, called Least-Item-First Pattern Growth (LIFPG), for mining frequent patterns. LIFPG mines frequent patterns directly in Trans-tree without using any additional data structures. The key idea is that least items are always considered first when the current pattern growth. By this way, conditional sub-tree can be created directly in Trans-tree by adjusting node-links and recounting counts of some nodes. Experiments show that, in comparison with FP-Growth, our algorithm is about four times faster and saves half of memory; it also has good time and space scalability with the number of transactions, and has an excellent performance in dense dataset mining as well.

Keywords Data mining, Frequent pattern, Association rule

1. 引言

频繁模式的挖掘是关联规则[1,2]、相关分析[3]、序列模式[4]、显露模式[5]等许多重要数据挖掘任务的基本步骤.长期以来,挖掘频繁模式主要采用 Apriori 算法[1]及其改进形式(参见文[6]).此类方法采用这样的启发式思想:若一个项集是非频繁的,则它的任何超集也是非频繁的.算法迭代地从长度为k的频繁项集产生长度为k+1的候选项集,并扫描数据库考察候选项集是否是频繁的,从而挖掘出全部频繁项集.这类算法的主要缺点是:需要产生大量候选项集,并反复扫描数据库,以确定新的候选项集是否是频繁的.这样,不仅大大增加了I/O开销,而且需要大量的CPU时间来处理数目巨大的候选项集.特别是当存在较长频繁模式或存在较大数量的候选项集时,这些开销将呈指数增加.

FP-Growth 算法[7]是一种基于模式增长的频繁模式挖掘算法,本质上不同于 Apriori 的候选产生-测试方法.FP-Growth 算法只需要两次扫描数据库:第一次扫描数据库,得到频繁1项集;第二次扫描数据库,利用频繁1项集过滤数据库中的非频繁项,同时生成 FP-树.由于 FP-树蕴涵了所有的频繁项集,其后的频繁项集的挖掘只需要在 FP-树上进行.文[7]的性能研究表明,FP-Growth 算法比 Apriori 算法快一个数量级.

然而,由于在挖掘频繁模式时需要递归地创建条件 FP-树,并且每当频繁模式增长时就要创建一棵条件 FP-树,FP-Growth 算法的时空效率仍然不够高.特别是在支持度阈值较小或存在长模式时,即使对于不太大的数据库,也将产生数以十万计的频繁模式。动态地创建数以十万计的条件 FP-树,将耗费大量时间和空间。

为了避免分离地创建条件 FP-树,本文提出了一种直接在 Trans-树上挖掘频繁模式的新算法 LIFPG(Least-Item-First Pattern Growth)算法。Trans-树(定义在第3节给出)是一种严格有序的线索树,并且与 FP-树相比,需要较少的空间.通过引进被约束子树,采用最小项优先的模式增长方式和有效的优化策略,LIFPG 算法通过调整 Trans-树中节点信息和节点链直接在 Trans-树中挖掘频繁模式,而不需要任何附加的数据结构.性能研究表明:LIFPG 算法速度大约是 FP-Growth 算法的3~4倍,而所需要的存储空间不到 FP-Growth 算法的一半.对于稠密数据集,LIFPG 算法的优势更加显著.

2. 问题描述

设项的集合 $I=\{x_1, \cdots, x_M\}$,一个事务数据库表示为 $TDB=\langle T_1, T_2, \cdots, T_N\rangle$,其中 $T_i\subseteq I$ 是一个事务.项的集合被称为一个项集或模式.事务 T 包含项集 X 当且仅当 $X\subseteq T$.在事务数据库 TDB 中,包含项集 X 的事务的个数称为 X 在 TDB 中的支持计数,记作 $count(X)$,X 的支持度表示为 $support(X)=count(X)/N$,其中 N 是 TDB 中事务的个数.给定的事务数据库 TDB 和最小支持度阈值 min_sup,项集 X 是频繁的当且仅当 $support(X)\geqslant min_sup$。

通常,最小支持度阈值 min_sup 由用户或领域专家指定,而最小支持度计数 $\xi=\lceil min_sup\times N\rceil$。

在事务数据库中挖掘频繁模式的问题可以描述为:给定事务数据库 TDB 和最小支持度阈值 min_sup,挖掘所有的频繁模式。

由于任何项集都可能是频繁的,为挖掘所有的频繁模式,我们可能需要考察所有可能的模式。但是,如果这样做,我们将面临组合爆炸问题.Apriori 性质告诉我们:如果一个长度为 k 的模式不是频繁的,则它的长度为$(k+1)$的超模式不可

*)本文的工作得到河南省自然科学基金(项目号:0111060700,0211050100)的资助。

维普资讯 http://www.cqvip.com

能是频繁的。利用 Apriori 性质，我们可以有效地对搜索空间进行剪裁。记 I 中的频繁项为 *item* [1]，*item* [2]，…，*item* [*max*]，则挖掘所有的频繁模式可以通过 *MineFPs*(∅,*max*)调用如下过程完成：

```
Procedure MineFPs(FP, last){
  if (FP+{item [1]} is frequent)then
     Output FP+{item [1]} and its support;
  for i=2 to last do {
    if (FP+{ item [i]} is frequent){
       Add item [i] into FP;
       Output FP and its support;
       MineFPs(FP, i-1);
       Remove item [i] from FP;
}}}
```

不难看出该算法是正确的。然而，当数据库很大时，判定一个模式是否频繁并求它的支持度并非易事。我们需要建立适当的数据结构，以便有效地实现上面的算法。

3. 事务树和被约束的子树

现在，我们来建立有效实现 MineFPs 所需要的数据结构。不失一般性，我们假定 I 上存在一个偏序，记作□。有许多方法定义□。例如，我们可以将所有项按支持度计数升序/降序排列来决定该偏序，或者直接使用字典序。本文假定□用项的支持度计数降序定义，不同的偏序对算法的影响将在算法的性能分析中讨论。

由于事务中的项可以按 I 上的偏序排序，并过滤掉非频繁项而不会影响频繁模式的挖掘，我们可以将每个事务看作项的有序集$\{i_1,i_2,\cdots,i_m\}$，其中 i_1□i_2□…□i_m；并假定事务中的所有项都是频繁的。

3.1 事务树

设 $T=\{i_1,i_2,\cdots,i_m\}$是一个事务，$\{i_1,i_2,\cdots,i_k\}$ $(k\leqslant m)$称事务 T 的前缀。

定义1 事务树(Trans-树)(事务树的定义类似于 FP-树，但要求严格有序。FP-树不是一个好的术语，它暗示有简单的搜索方法从 FP-树中挖掘频繁模式。事实上，每个事务(而不是每个频繁模式)对应 FP-树的一条路径，从而没有平凡的方法从 FP-树中挖掘频繁模式。因此，我们使用事务树，而不用 FP-树)是一棵线索树，定义如下：

1)Trans-树包含一个根节点，用"*null*"标记，和一组子树作为根的子女；每棵子树或者是树叶节点，或者有若干子女，每个子女都是一棵子树。

2)子树的每个节点包含三个字段：*item*，*count*，*next*。其中，*item* 记录该节点代表的项；*count* 是计数，如果项集$\{i_1,\cdots,i_k\}$(i_k= *item*)依次出现在从根到该节点的路径上，则 *count* 等于以$\{i_1,\cdots,i_k\}$为前缀的事务的个数；*next* 是一个指针，或者指向树中具有相同 *item* 值的下一个节点，或者为空(如果没有下一个节点)。

3)Trans-树是有序的：(1)如果节点 M 是节点 N 父母节点，则 $M.item$□$N.item$；(2)如果节点 N_1和 N_2都是节点 M 的子女，并且 N_1在 N_2的左边，则 $N_1.item$□$N_2.item$。

4)Trans-树是一棵线索树：具有相同 *item* 值的节点被 *next* 指针链接在一起，形成节点链。有两个数组 *node-link* 和 *totalcnt*，其中 *node-link* [*item*]指向树中具有 *item* 值的第一个节点，而 *totalcnt* [*item*]记录对应节点链中节点的计数和。

给定事务数据库 TDB，Trans-树可用如下步骤创建：

1. 从空树出发，扫描事务数据库，将每个事务插入 Trans-tree。

2. 创建节点链，并累计节点链中节点的计数和。

步骤1类似于文[7]创建 FP-树，本文从略。步骤2可以用如下过程实现：

```
procedure create_node_link(T){
  for (each child N of node T){
  totalcnt [N.item]= totalcnt [N.item]+ N.count; // accumulate counts
  Insert node N to the node link pointed by node-link [N.item];
  create_node_link(N); // create node link recursively
}}
```

例1. 考虑表1给出的事务数据库，其中 TID 表示事务的标识。假定最小支持度计数为2。a,b,c,d,e 和 f 的支持度计数分别为4，6，6，2，4和1。从而，b□c□a□e□d□f。在创建 Trans-树时，f 将被忽略，因为它不是频繁的。所创建的 Trans-树如图1所示。

表1 一个事务数据库

TID	itemset
100	*c, e*
200	*a, b, c, e*
300	*b,c, d*
400	*b, c*
500	*a, b*
600	*b, f*
700	*b, e*
800	*a, c, e*
900	*a, c, d*

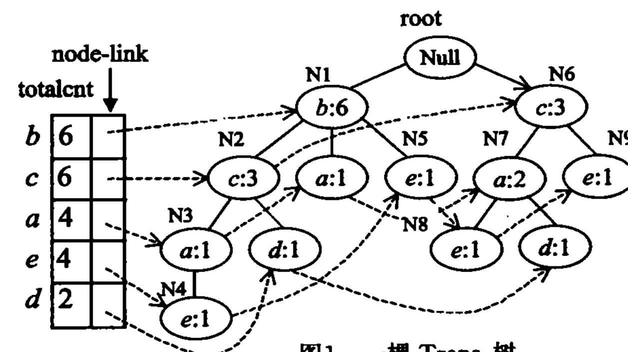

图1 一棵 Trans-树

3.2 被约束的子树

定义2 设 i_k□…□i_1 $(k\geqslant 1)$是项，P 是一条从根到节点 N 的路径。P 被项集$\{i_k,\cdots,i_1\}$约束，如果存在 N 的子孙节点 N' 使得 $i_k,\cdots,i_1$依次出现在从 N 到 N' 的子路径上，并且 i_k 出现在节点 N 上，i_1出现在节点 N' 上。节点 N' 的计数称为路径 P 在约束$\{i_k,\cdots,i_1\}$下的基计数。

定义3 被相同项集$\{i_k,\cdots,i_1\}$约束的所有路径都包含根节点，从而形成 Trans-树的一棵子树，称为被$\{i_k,\cdots,i_1\}$约束的子树，记作 $ST(i_1,\cdots,i_k)$。设 N 是 $ST(i_1,\cdots,i_k)$中的节点，所有经过 N 被$\{i_k,\cdots,i_1\}$约束的路径的基计数之和称为节点 N 的频度计数。

如果将节点的计数看作频度计数，Trans-树是一棵被空集约束的子树 ST(∅)。

引理1 设 i_k□i_{k-1}□…□i_1 $(k\geqslant 1)$是项，则

(1)$ST(i_1,\cdots,i_{k-1},i_k)$是 $ST(i_1,\cdots,i_{k-1})$的子树。

(2)设 N 是 $ST(i_1,\cdots,i_{k-1},i_k)$的节点，而 $ST(i_1,\cdots,i_{k-1})$中 *item* 等于 i_{k-1}并且是 N 的后代的节点为 $N_1,\cdots,N_j$，则 N 的频度计数等于 $N_1,\cdots,N_j$ 的频度计数和。

维普资讯 http://www.cqvip.com

该引理可以根据定义2和3,对 k 归纳加以证明。

该引理表明 $ST(i_1,\cdots,i_{k-1},i_k)$ 可以直接由 $ST(i_1,\cdots,i_{k-1})$ 创建,从而被约束的子树可以递归地创建。对于任意的 i_j,如果我们直接用节点中的 *count* 记录 $ST(i_j)$ 中节点的频度计数,直接在 Trans-树创建 $ST(i_j)$ 是简单的。我们只需要对 $ST(i_j)$ 中的节点重新设置计数,建立节点链,并累计节点链中节点的计数和。下面的例子解释了这一过程,详细的算法将在4.2给出。

例2. 考虑在图1所示的 Trans-树中建立 $ST(e)$。e 的节点链上有4个节点 $N4$,$N5$,$N8$和 $N9$。首先,考虑 $N4$,出现在从 $N4$到根的路径上的节点依次为 $N3$,$N2$和 $N1$。这些节点都是第一次遇到,我们用 $N4$的 *count* 值1设置 $N3$,$N2$和 $N1$的 *count* 值,把它们作为相应的节点链的第一个节点插入,并同时累计相应节点链的计数和。现在,我们有 $N3.count = N2.count = N1.count = 1$,$totalcnt[a] = totalcnt[c] = totalcnt[b] = 1$。然后,考虑 $N5$,出现在从 $N5$到根的路径上的节点只有 $N1$,并且不是第一次遇到。我们将我们用 $N5$的 *count* 值1累加到 $N1.count$ 和 $totalcnt[b]$,得到 $N1.count = 2$,$totalcnt[b] = 2$。用类似的方法处理 $N8$和 $N9$,最后得到 $ST(e)$如图2(a)所示。其中,$ST(e)$中的节点用浅灰色标记,以便于观察。

有了 $ST(e)$,用类似的方法可以创建 $ST(e,a)$,如图2(b)所示。

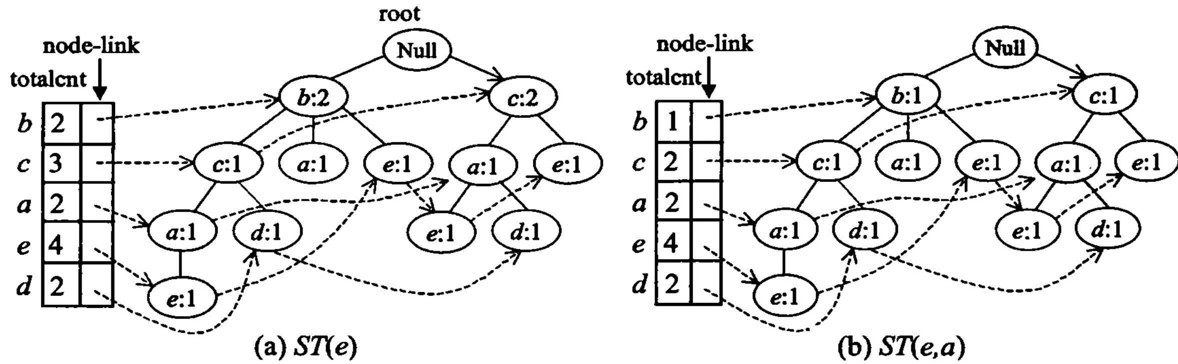

图2 在 Trans-树中递归地创建被约束的子树

从上面的例子可以看出:如果我们先创建 $ST(e)$,尽管可以创建 $ST(e,a)$,但我们无法正确地创建 $ST(a)$,因为在创建 $ST(e)$时已经改变了节点的计数和节点链。解决该问题的基本策略是最小项优先(Least Item First,简记为 LIF):当我们从 Trans-树出发创建被单个项约束的被约束子树时,我们优先考虑(在偏序□意义下的)最小项。

例如,对于图1的 Trans-树,这个次序为 $ST(b)$,$ST(c)$,$ST(a)$,$ST(e)$,$ST(d)$,因为我们有 b□c□a□e□d。在递归地创建被约束的子树时,同样应当遵循 LIF 策略。值得注意的是 LIF 策略与 *MineFPs* 过程是相容的。事实上,过程 *MineFPs* 采用的正是 LIF 策略。

为了将以上数据结构用来有效地实现 *MineFPs*,我们还需要一个引理,建立该结构与项集的联系。

引理2 如果项 i 出现在 $ST(i_1,\cdots,i_k)$的某节点中,则项集$\{i_1,\cdots,i_k,i\}$的支持度计数等于 $totalcnt[i]$。

该引理可以根据 Trans-树和被约束的子树的定义加以证明。为节省篇幅,本文从略。

4 LIFPG 算法

利用 Trans-树,我们可以有效地实现 MineFPs 过程。其基本思想是:通过考察 $totalcnt[i]$是否大于最小支持度阈值 ξ 来判定 $FP+\{i\}$是否频繁,并且一旦 i 添加到 FP,就立即创建被 FP 约束的子树。由于在创建被约束子树时我们采取了最小项优先策略,我们称该算法为最小项优先模式增长(Least-Item-First Pattern Growth, LIFPG)算法。

4.1 进一步优化策略

LIFPG 算法的主要开销是递归地建立被约束的子树。从前面的讨论(例2)我们看到,建立一棵被约束的子树,如 $ST(i_j)$的主要开销是对 $ST(i_j)$中的节点重新计算计数,建立节点链,并累计节点链中节点的计数和。通常,被约束的子树中的节点链可能很多。例如,如果事务数据库中有1000个不同的项 i_1□i_2,□…□i_{1000},为建立 $ST(i_{900})$,需要建立的节点链可能多达900个。然而,只有少量的节点链,其节点的计数和大于或等于最小支持度阈值 ξ。假定存在 k,当 $j<k$ 时,$totalcnt[i_j]<\xi$,而 $totalcnt[i_k]\geq\xi$。根据 Apriori 性质和引理2,当 $j<k$ 时,我们不必创建 $ST(i_{900},i_j)$。进一步说,如果 i 是不小于 i_k 的项,在创建 $ST(i_{900},i)$之后,当 $j<k$ 时,也必然有 $totalcnt[i_j]<\xi$。也就是说,在创建 $ST(i_{900},i)$时,考虑这些包含 i_j 的节点是不必要的。一般地,我们有如下优化策略:

如果在 $ST(i_1,\cdots,i_l)$中,当 $j<k$ 时,$totalcnt[i_j]<\xi$,而 $totalcnt[i_k]\geq\xi$,则在创建 $ST(i_1,\cdots,i_l,i)$时,只需要对其 *item* 值大于或等于 i_k 的节点重新计算计数,建立节点链,并累计节点链中节点的计数和。

该优化策略具有很好的效果。我们的实验表明,它可以节省大约1/3的挖掘时间。

4.2 LIFPG 算法的伪代码

现在,我们给出 LIFPG 算法的伪代码。为简化符号,假定频繁项的集合已经同构地映射到整数集$\{1,2,\cdots,max\}$,并且对于给定的事务数据库 TDB 和支持度阈值 ξ,Trans-树已经建立。挖掘所有频繁模式可以通过调用 *MineFPs*(∅,1,*max*)实现。而 *MineFPs* 的伪代码如下:

```
Procedure MineFPs (FP, first, last){
  while (totalcnt [first]<ξ)do
      first=first + 1;
  Output FP+{first} and its support totalcnt [first]/N;
  for i = first + 1 to last do {
    if (totalcnt [i]≥ξ){
        Add i into FP;
        Output FP and its support totalcnt [i]/N;
        Build_ST(FP, first);
        MineFPs (FP, first, i-1);
        Remove i from FP;
```

维普资讯 http://www.cqvip.com

```
}}}
```

其中,建立被约束的子树可以用如下过程实现:

```
Procedure Build_ST(FP, first){
  Let k be the minimum item in FP;
  for i=first to k-1 do
    Clear node-link [i] and totalcnt [i];
  for (each node N in node-link [k])do {
    basecount = N.count;
    Let M be the parent of node N;
    while (M.item≥first)do {
      if (M is not in node-link [M.item])then {
        Insert M into node-link [M.item];
        M.count = basecount;
      }else M.count = M.count + basecount;
      totalcnt [M.item] = totalcnt [M.item]+basecount;
      Let M be its parent;
}}}
```

利用引理2,可以证明:

定理1 给出事务数据库和支持度阈值,本文的算法能正确地挖掘出所有频繁模式。

5 性能研究

FP-Growth 是已知的挖掘频繁模式的高效算法之一。在基本相同的条件下,我们从不同的方面对 LIFPG 算法进行了测试,并与 FP-Growth 算法进行了比较。

实验环境:CPU Intel 450MHz;内存128Mbyte

数据集:T10K:含1K 个项,10k 个事务,事务的平均长度为25。

T100K:含10K 个项,100k 个事务,事务的平均长度为60。

Mushroom:含117个不同项,8124个事务,事务的平均长度为50。

其中,数据集 T100K 和 T10K 取自文[7],而 Mushroom 取自 Irvine Machine Learning Database Repository[8]。

图3展示4.1节的优化策略对本文算法性能的影响。可以看出,通过避免处理无用的链,LIFPG 算法的运行时间大约可以节省1/3。图4给出在支持度递减序、支持度递增序和字典序三种不同偏序下,LIFPG 算法在 T100K 数据集上的运行时间随支持度变化的曲线。在三种不同的偏序下,LIFPG 算法的性能差别不显著,而在支持度递减序下的性能略好一些。采用支持度递减序,最小项优先相当于最大支持度优先(Maximum-Support-First, MSF)。因此,此时 LIFPG 算法也称 MSFPG 算法。

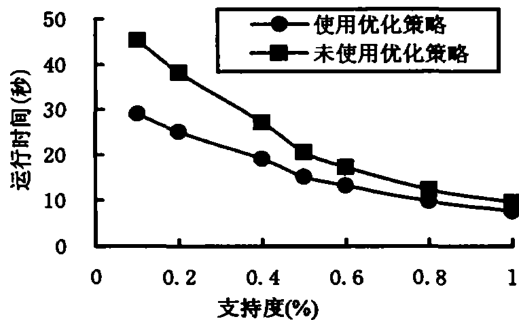

图3 避免处理无用的链可以优化 LIFPG 算法的性能

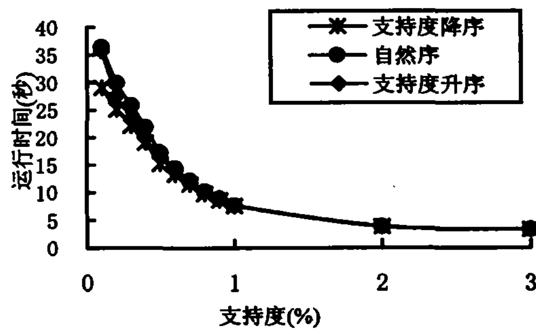

图4 项的不同偏序对 LIFPG 算法的影响

图5给出 LIFPG 算法和 FP-Growth 算法的运行时间比较,其中 FP-Growth 算法的运行时间取自文[7]。可以看出,本文算法的时间效率明显优于 FP-Growth。当支持度为0.1%时,在 T10K 的数据集上,本文算法运行时间大约为 FP-Growth 算法的1/3;而在 T100K 的数据集上,本文算法运行时间大约 FP-Growth 的1/4。此外,与 FP-Growth 算法相比,本文算法运行时间随支持度减小(频繁模式数目增多)而增长的趋势也平缓得多。

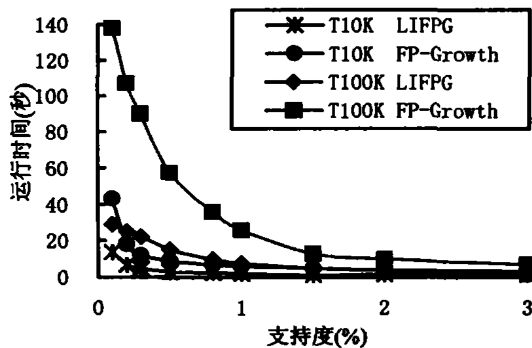

图5 LIFPG 与 FP-Growth 比较
(稀疏数据集 T10K 和 T100K)

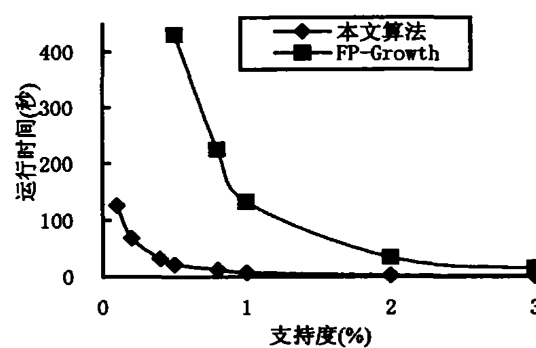

图6 LIFPG 与 FP-Growth 比较
(稠密数据集 Mushroom)

在稠密数据集上,LIFPG 算法的优势更加显著。Mushroom 数据库有8124个事务,事务的平均长度为50,包含117个不同项(属性)。这虽然是一个很小的数据库,但它非常稠密,项之间的相关性很强,因此即使在最小支持度阈值较大时,也存在大量频繁模式,并且随着支持度的减小存在严重的组合爆炸问题[9]。图6给出了 FP-Growth 算法和本文算法在 Mushroom 数据库上的运行时间(不含频繁模式的输出时间)。可以看出,对于稠密数据集,本文的算法比 FP-Growth 算法快20多倍。

固定最小支持度阈值为1%,从数据集 T100K 中随机抽取事务组成数据集。随事务数量的增长,FP-Growth 算法与

(下转第123页)

维普资讯 http://www.cqvip.com

结语 信息不完备是复杂决策环境中不可避免的问题[5]。由于不完备决策表中的不确定成分太多，因此从不完备决策表中挖掘出潜在的有用知识，增加了数据挖掘的难度。传统的粗糙集理论是处理不确定信息的有效方法，但它仅实用于完全决策表的情况，因此在应用中受到限制。本文利用基于优势-等价关系的扩展粗糙集模型，结合模糊集理论知识，给出了有效的区间值属性决策表的数据挖掘方法。算例证明了该方法的有效性。对于区间值属性与数量型属性复合的决策表，以及区间值属性与数量型属性、质量型属性三种情况复合的决策表，可用类似的方法来处理。

参考文献

1 李永敏，朱善君，陈湘晖，等. 基于粗糙集理论的数据挖掘模型. 清华大学学报(自然科学版)，1999，39(1)

2 Greco S, Matarazzo B, Slowinski R. Rough Approximation of a Preference relation by dominance relations. Eropean Journal of Operational Research ，1997，117：63～83

3 Greco S, Mataarazzo B, Slowinski R. Rough Approximation by Dominance Relations. International Journal of Intelligent Systems，2002，17：153～171

4 张全，樊治平，潘德惠. 不确定多属性决策中区间数的一种排序方法. 系统工程理论与实践，1999(5)：129～133

5 赵卫东，李旗号，盛昭翰. 区间值属性不完全信息下的数据挖掘. 系统工程理论与方法，2001，10(2)

(上接第120页)

本文的算法运行时间和存储空间的增长分别如图7和图8所示(由于文[7]未提供对应数据，我们实现了 FP-Growth 算法。我们实现的 FP-Growth 算法运行效率与文[7]报告的结果大致相当，我们的实现稍快一点。例如，对于数据集 T100K，当最小支持度阈值为1%时，文[7]报告的运行时间大约为26秒，而我们的实现为22.18秒。图6～图8的比较涉及的 FP-Growth 算法是基于我们的实现)。可以看出，随着数据集的增大，两个算法的运行时间和存储空间都线性地增长。然而，LIFPG 算法增长的速度比较缓慢，并且时空性能始终优于 FP-Growth 算法。这表明与 FP-Growth 算法相比，本文的算法具有更好的可伸缩性。

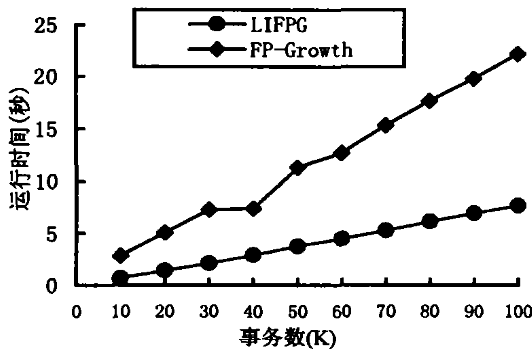

图7 算法的时间可伸缩性(最小支持度1%)

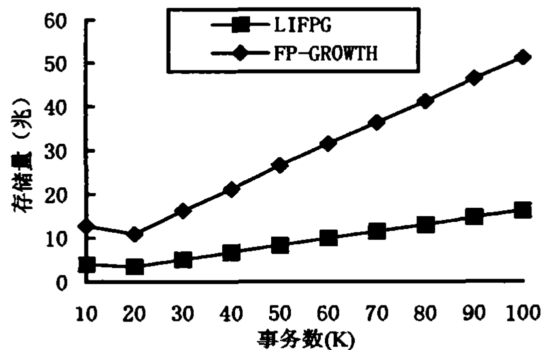

图8 算法的空间可伸缩性(最小支持度1%)

结束语 我们提出了一种直接在 Trans-树中挖掘频繁模式的有效算法 LIFPG 算法，实现了该算法，并研究了它的性能。我们的实验表明，与 FP-Growth 算法相比，LIFPG 算法的挖掘速度大约提高4倍，而存储开销节省一半。对于稠密数据，速度的提高更为显著。随着数据库容量的增长，本文的算法具有更好的可扩展性。

参考文献

1 Agrawal R, Srikant R. Fast algorithms for Mining association rules. In：Proc. 1994 Int'l Conf. on Very Large Data Bases，Sept. 1994. 487～499

2 Park J S，Chen M S. Yu P S. An effective hash-based algorithm for mining association rules. In：Proc. 1995 ACM-SIGMOD Int'l Conf. on Management of Data, May 1995. 175～186

3 Brin S，Motwani R，Silverstein C. Beyond market basket：Generalizing association rules to correlations. In：Proc. 1997 ACM-SIGMOD Int'l Conf. on Management of Data，May 1997. 265～276

4 Agrawal R，Srikant R. Mining sequential patterns, In ICDE'95, pages 3～14

5 Dong G，Li J. Efficient mining of emerging patterns：Discovering trends and differences. In：Proc. of the fifth ACM SIGKDD Intl. Conf. on Knowledge Discovery and Data Mining, Aug. 1999. 43～52

6 Han Jiawei，Kamber M 著，范明，孟小峰等译. 数据挖掘：概念与技术. 机械工业出版社，2001. 149～184

7 Han J，Pei J，Yin Y. Mining frequent patterns without candidate generation. In：Proc. 2000 ACM-SIGMOD Intl. Conf. on Management of Data，May 2000. 1～12

8 http：//www. ics. uci. edu/～mlearn/MLRepository. html

9 Bykowski A，Rigotti C. A Condensed Representation to Find Frequent Patterns. In：Proc. of the 20th ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems (PODS 2001)，Santa Barbara，CA，USA，ACM Press，2001. 267～273