Transactions on Information and Communications Technologies vol 19 © 1998 WIT Press, www.witpress.com, ISSN 1743-3517

Finding the $\mathcal{N}$ Largest Itemsets

Li Shen, Hong Shen, Paul Pritchard & Rodney Topor
School of Computing and Information Technology
Griffith University, Nathan, QLD4111, Australia
EMail: {L.Shen, H.Shen, P.Pritchard, R.Topor}@cit.gu.edu.au

Abstract

The largest itemset in a given collection of transactions $\mathcal{D}$ is the itemset that occurs most frequently in $\mathcal{D}$. This paper studies the problem of *finding the $\mathcal{N}$ largest itemsets*, whose solution can be used to generate an appropriate number of interesting itemsets for mining association rules. We present an efficient algorithm for finding the $\mathcal{N}$ largest itemsets. The algorithm is implemented and compared with the naive solution using the *Apriori* approach. We present experimental results as well as theoretical analysis showing that our algorithm has a much better performance than the naive solution. We also analyze the cost of our algorithm and observe that it has a polynomial time complexity in most cases of practical applications.

1 Introduction

Discovery of association rules is an important problem within the area of data mining. The problem is introduced by Agrawal et al. [2], and can be formalized as follows. Let $\mathcal{I} = \{i_1, i_2, \cdots, i_m\}$ be a set of literals, called *items*. Let $\mathcal{D}$ be a collection of transactions, where each transaction T has a unique identifier and contains a set of items such that $T \subseteq \mathcal{I}$. We call $\mathcal{I}$ *item-domain*, a set of items *an itemset*, and an itemset with k items *a k-itemset*. The *support* (i.e., *support count* in previous studies) of an itemset x in $\mathcal{D}$, denoted as $\sigma(x/\mathcal{D})$, is the number of transactions containing x in $\mathcal{D}$. An *association rule* is an expression $x \Rightarrow y$, where $x, y \subseteq \mathcal{I}$ and $x \cap y = \emptyset$. The *confidence* of $x \Rightarrow y$ is the ratio of $\sigma(x \cup y/\mathcal{D})$ to $\sigma(x/\mathcal{D})$. We use minsup and

Transactions on Information and Communications Technologies vol 19 © 1998 WIT Press, www.witpress.com, ISSN 1743-3517

minconf to denote the user-specified minimum support and confidence respectively. An itemset x is *large* (*frequent*) if $\sigma(x/\mathcal{D}) \geq$ minsup. An association rule $x \Rightarrow y$ is *strong* if $x \cup y$ is large and $\frac{\sigma(x \cup y/\mathcal{D})}{\sigma(x/\mathcal{D})} \geq$ minconf. The problem of *mining association rules* is to *find all strong association rules*, which can be divided into two subproblems: (1) *find all large itemsets* (called MS-Problem below); (2) *find all strong rules from all large itemsets*. Because the second subproblem is relatively straightforward, all previous studies [1, 2, 7, 8, 9] have emphasized on developing efficient algorithms for the first subproblem.

In essence, the first subproblem aims to obtain a set of interesting itemsets. However, by giving only a minsup, users have no idea at all about how many itemsets will be generated in the result. On one hand, the result may contain too few itemsets so that users fail to get enough useful ones. On the other hand, the result may contain too many itemsets so that most of them are actually uninteresting. This observation motivates us to study how to generate an appropriate number of the most interesting itemsets. Given $x, y \subseteq I$, we say that x is *larger* than y or y is *smaller* than x if $\sigma(x/\mathcal{D}) > \sigma(y/\mathcal{D})$. Since users are usually more interested in those itemsets with larger supports, we propose a new problem (called NL-Problem) as follows: *find the $\mathcal{N}$ largest itemsets from a given collection of transactions, where $\mathcal{N}$ is a user-specified number of interesting itemsets.*

A naive solution for NL-Problem is to directly extend the existing algorithms for MS-Problem as follows. Choose a minsup and execute one of the existing algorithms. If the result contains more than $\mathcal{N}$ itemsets, choose the $\mathcal{N}$ largest ones from the result and we are done. Otherwise, execute one of the existing algorithms by using a smaller minsup repeatedly until the result contains at least $\mathcal{N}$ itemsets; then choose the $\mathcal{N}$ largest ones from the result.

However, the above naive solution is inefficient. First, the solution might repeatedly execute one of the existing algorithms with multiple different minsups, which results in many redundant or repeated operations. Second, the speeds of all the existing algorithms are related to minsup. The smaller the minsup is, the slower the algorithms are. For very small minsup, these algorithms break down because they have an exponential worst-case time complexity (in $|\mathcal{I}|$). To get enough itemsets in the result, the last minsup chosen by the naive solution tends to be relatively small. Therefore, the naive solution not only often runs slowly, but also is easy to run into the exponential bottleneck.

In this paper, we develop an efficient algorithm for NL-Problem, which is implemented and compared with the naive solution. Both experimental results and theoretical analysis show that our algorithm has a much better performance than the naive solution. We present some preliminary concepts in Section 2; describe our algorithm in Section 3; present our experimental results as well as algorithm analysis in Section 4; and conclude the paper in Section 5.

2 Preliminary Concepts

We always use $\mathcal{I}$ to denote *the item-domain*, $\mathcal{D}$ to denote *the relevant collection of transactions*, $\mathcal{N}$ to denote *the user-specified number of desired largest itemsets*, and $\sigma(x/\mathcal{D})$ to denote *the support of an itemset x in $\mathcal{D}$*. By definition, $\emptyset$ is always the largest itemset, but we won't include it in our study because it can't provide users with any helpful information. Thus, all itemsets mentioned hereafter refer to *nonempty* itemsets.

Let x be an itemset. We introduce the *rank* of x, denoted by $\theta(x)$, as follows: $\theta(x) = |\{y \mid \sigma(y/\mathcal{D}) > \sigma(x/\mathcal{D}),\ \emptyset \subset y \subseteq \mathcal{I}\}| + 1$. We call x a *winner* if $\theta(x) \leq \mathcal{N}$ and $\sigma(x/\mathcal{D}) \geq 1$, which means that x is one of the $\mathcal{N}$ largest itemsets and x occurs in $\mathcal{D}$ at least once. We don't regard any itemset with support 0 as a winner even if it is ranked below $\mathcal{N}$, because there is no need to provide users with an itemset that doesn't occur in $\mathcal{D}$ at all.

We use $\mathcal{W}$ to denote *the set of all winners*; and call the support of the smallest winner *the critical support*, denoted by crisup. Clearly, $\mathcal{W}$ exactly contains all itemsets with support exceeding crisup; and also we have crisup ≥ 1. It is easy to see that $|\mathcal{W}|$ may be different from $\mathcal{N}$: (1) if the number of all itemsets occurring in $\mathcal{D}$ is less than $\mathcal{N}$, $|\mathcal{W}|$ will be less than $\mathcal{N}$; (2) $|\mathcal{W}|$ may also be greater than $\mathcal{N}$, as different itemsets may have the same support. The problem of *finding the $\mathcal{N}$ largest itemsets* is to *generate $\mathcal{W}$*.

Let x be an itemset. We use $\mathcal{P}_k(x)$ to denote *the set of all k-subsets (subsets with size k) of x*. We use $\mathcal{U}_k$ to denote $\mathcal{P}_1(\mathcal{I}) \cup \cdots \cup \mathcal{P}_k(\mathcal{I})$, which means *the set of all itemsets with size not greater than k*. Thus, we introduce the *k-rank* of x, denoted by $\theta_k(x)$, as follows: $\theta_k(x) = |\{y \mid \sigma(y/\mathcal{D}) > \sigma(x/\mathcal{D}),\ y \in \mathcal{U}_k\}| + 1$. We call x a *k-winner* if $\theta_k(x) \leq \mathcal{N}$ and $\sigma(x/\mathcal{D}) \geq 1$, which means that, among all itemsets

Transactions on Information and Communications Technologies vol 19 © 1998 WIT Press, www.witpress.com, ISSN 1743-3517

with size not greater than k, x is one of the $\mathcal{N}$ largest ones, and also x occurs in $\mathcal{D}$ at least once. We use $\mathcal{W}_k$ to denote *the set of all k-winners*. We define *k-critical-support*, denoted by k-crisup, as follows: if $|\mathcal{W}_k| < \mathcal{N}$, k-crisup is 1; otherwise, k-crisup is the support of the smallest k-winner. Clearly, $\mathcal{W}_k$ exactly contains all itemsets with size not greater than k and support not less than k-crisup. We present some useful properties of the above concepts as follows.

Property 2.1 *Let k and i be integers such that $1 \le k \le k+i \le |\mathcal{I}|$.*

(1) Given $x \in \mathcal{U}_k$, we have $x \in \mathcal{W}_k$ iff $\sigma(x/\mathcal{D}) \ge k$-crisup.

(2) If $\mathcal{W}_{k-1} = \mathcal{W}_k$, then $\mathcal{W} = \mathcal{W}_k$.

(3) $\mathcal{W}_{k+i} \cap \mathcal{U}_k \subseteq \mathcal{W}_k$.

(4) $1 \le k$-crisup $\le (k+i)$-crisup.

Proof (1) Straightforward by relevant definitions.

(2) By $\mathcal{W}_{k-1} = \mathcal{W}_k$, $\mathcal{W}_k$ contains no k-itemsets. Thus, none of k-itemsets has support exceeding k-crisup. Since any $(k+1)$-itemset is not larger than any of its k-subsets, we know that none of $(k+1)$-itemsets has a support exceeding k-crisup and can become a $(k+1)$-winner. So we have $(k+1)$-crisup $=$ k-crisup and $\mathcal{W}_k = \mathcal{W}_{k+1}$. Likewise, we have $\mathcal{W}_{k+1} = \mathcal{W}_{k+2} = \cdots = \mathcal{W}_{|\mathcal{I}|}$. Since $\mathcal{W} = \mathcal{W}_{|\mathcal{I}|}$ (by definition), we have $\mathcal{W} = \mathcal{W}_k$.

(3) Let $x \in \mathcal{W}_{k+i} \cap \mathcal{U}_k$. By $x \in \mathcal{U}_k$ and $\mathcal{U}_k \subseteq \mathcal{U}_{k+i}$, we have $\theta_k(x) \le \theta_{k+i}(x)$. From $x \in \mathcal{W}_{k+i}$, we have $\theta_k(x) \le \theta_{k+i}(x) \le \mathcal{N}$, and so $x \in \mathcal{W}_k$. Hence, $\mathcal{W}_{k+i} \cap \mathcal{U}_k \subseteq \mathcal{W}_k$.

(4) By k-crisup ≥ 1 and $(k+i)$-crisup ≥ 1, the proof is obvious for k-crisup $= 1$ or $i = 0$. The following is the proof for k-crisup > 1 and $i > 1$. We assume that $(k+i)$-crisup $<$ k-crisup. Thus $\mathcal{W}_{k+i}$ contains not only all k-winners but also at least one itemset with support $(k+i)$-crisup, say x. Since all k-winners are larger than x, we have $\theta_{k+i}(x) > |\mathcal{W}_k|$. From k-crisup > 1, we have $|\mathcal{W}_k| \ge \mathcal{N}$, and so $\theta_{k+i}(x) > \mathcal{N}$, which contradicts that x is a $(k+i)$-winner. Hence, we have $(k+i)$-crisup $\ge$ k-crisup. □

3 Algorithm

To find all winners, our algorithm makes multiple passes over the data. In the first pass, we count the supports of all 1-itemsets, select the $\mathcal{N}$ largest ones from them to form $\mathcal{W}_1$, and then use $\mathcal{W}_1$ to

Transactions on Information and Communications Technologies vol 19 © 1998 WIT Press, www.witpress.com, ISSN 1743-3517

generate potential 2-winners with size 2. In each subsequent pass k: first we count the support for potential k-winners with size k (called *candidates*) during the pass over $\mathcal{D}$; then select the $\mathcal{N}$ largest ones from a pool precisely containing supports of all these candidates and all $(k-1)$-winners to form $\mathcal{W}_k$; finally use $\mathcal{W}_k$ to generate potential $(k+1)$-winners with size $k+1$, which will be used in the next pass. This process continues until we can't get any potential $(k+1)$-winners with size $k+1$, which implies $\mathcal{W}_{k+1} = \mathcal{W}_k$. By Property 2.1(2), we know that the last $\mathcal{W}_k$ exactly contains all winners. We present our algorithm as follows.

Algorithm 3.1 *Find the $\mathcal{N}$ largest itemsets.*
Input: *(1) $\mathcal{I}$, item-domain; (2) $\mathcal{D}$, a collection of transactions over $\mathcal{I}$; (3) $\mathcal{N}$, a user-specified number of desired largest itemsets.*
Output: *$\mathcal{W}$, the set of all winners.*

```
(1)   Let $Q$ be a priority-queue with $\mathcal{N}$ elements that are all equal to 1;
(2)   $C_1$ = the set of all 1-itemsets;
(3)   for ($k = 1; C_k \neq \emptyset; k++$) do begin
(4)     count supports for all $c$'s in $C_k$;  //Count supports by a pass over $\mathcal{D}$
(5)     $k$-crisup = Crisup-gen($Q$, $C_k$);  //Calculate $k$-crisup
(6)     $L_k = \{c \in C_k \mid \sigma(c/\mathcal{D}) \geq k\text{-crisup}\}$;  //Obtain k-winners with size k
(7)     $\mathcal{W}_k = L_k \cup \{c \in \mathcal{W}_{k-1} \mid \sigma(c/\mathcal{D}) \geq k\text{-crisup}\}$;  //Get all k-winners
(8)     $C_{k+1}$ = Candidate-gen($L_k$);  //Generate potential $(k+1)$-winners
(9)   end
(10)  $\mathcal{W} = \mathcal{W}_{k-1}$;

(11)  function Candidate-gen($L_k$: a set of $k$-itemsets)
(12)    $C_{k+1} = \{x \cup y \mid x, y \in L_k\} \cap \mathcal{P}_{k+1}(\mathcal{I})$;  //Join
(13)    for every $c \in C_{k+1}$ do
(14)      if $\mathcal{P}_k(c) \not\subseteq L_k$ then delete $c$ from $C_{k+1}$;  //Prune
(15)    return $C_{k+1}$;

(16)  function Crisup-gen($Q$: a priority-queue, $C_k$: candidate set)
(17)    for every $c \in C_k$ do begin
(18)      Let $q$ be one of the smallest elements in $Q$;
(19)      if $\sigma(c/\mathcal{D}) > q$ then delete $q$ from $Q$ and insert $\sigma(c/\mathcal{D})$ into $Q$;
        //Maintain the property that $Q$ stores the $\mathcal{N}$ largest supports found so far
(20)    end
(21)    return the smallest support in $Q$;
```

Now we explain the main procedure of Algorithm 3.1. First, the algorithm initializes a priority-queue Q at line 1, and Q will be used

Transactions on Information and Communications Technologies vol 19 © 1998 WIT Press, www.witpress.com, ISSN 1743-3517

later for calculating k-crisup. We use C_k to store *a superset of all k-winners with size k*, called *candidate set*, and so the algorithm lets C_1 contain all 1-itemsets at line 2. Then the algorithm enters the *for loop* at lines 3-9, where each iteration k consists of five phases. First, the supports of candidates in C_k are counted by a pass over $\mathcal{D}$ at line 4. A hash tree (refer to [1]) is used to store C_k so that this procedure of support-counting can be accelerated. Second, the algorithm calls *Crisup-gen* function (described below) at line 5 to calculate k-crisup. Third, at line 6, it selects all elements in C_k whose supports are not less than k-crisup to get L_k, the set of all k-winners with size k. The completeness of this phase is guaranteed by the inductive assumption that C_k is a superset of all k-winners with size k. Fourth, at line 7, the algorithm generates $\mathcal{W}_k$ based on Property 2.1(3). Finally, it uses L_k and *Candidate-gen* function (describe below) to generate C_{k+1}, which is a superset of all $(k+1)$-winners with size $k+1$ and will be used in the next iteration. The above loop continues until $C_k = \emptyset$, which implies $\mathcal{W}_{k-1} = \mathcal{W}_k$. Thus, by Property 2.1(2), the algorithm generates $\mathcal{W} = \mathcal{W}_{k-1}$ at line 10.

The *Candidate-gen* function uses L_k, the set of all k-winners with size k, to generate C_{k+1}, a superset of all $(k+1)$-winners with size $k+1$. The procedure is the same as *Apriori Candidate Generation* in [1]. First, in the *join* step at line 12, we join L_k with L_k to get C_{k+1}. Next, in the *prune* step at lines 13-14, we delete all itemsets $c \in C_{k+1}$ such that some k-subset of c is not in L_k. Now we need to show that C_{k+1} is a superset of all $(k+1)$-winners with size $k+1$. Let x be a $(k+1)$-winners with size $k+1$, i.e., $x \in \mathcal{W}_{k+1} \cap \mathcal{P}_{k+1}(\mathcal{I})$. Thus, it will be enough to prove that $\mathcal{P}_k(x) \subseteq L_k$. By Property 2.1(1)(4), we have $\sigma(x/\mathcal{D}) \geq (k+1)$-crisup $\geq k$-crisup. Since any itemset is not larger than any of its subsets, we have $\sigma(y/\mathcal{D}) \geq \sigma(x/\mathcal{D}) \geq k$-crisup and so $y \in L_k$, for all $y \in \mathcal{P}_k(x)$. Thus, we have $\mathcal{P}_k(x) \subseteq L_k$.

We use a priority-queue Q to help the calculation of k-crisup. Q is always maintained to store the $\mathcal{N}$ largest supports found currently. Thus, the initial value of Q should contain the $\mathcal{N}$ minimum supports that are all equal to 1, and this initialization of Q has been done at line 1 of the main procedure. Then, in each iteration k of the main procedure, Q is updated once (line 5), after the new candidates in C_k have obtained their supports. The *Crisup-gen* function is used to update Q. Before calling Crisup-gen(Q,C_k), Q stores the $\mathcal{N}$ largest supports among all those of itemsets in $\mathcal{U}_{k-1}$. It is obvious that the

Transactions on Information and Communications Technologies vol 19 © 1998 WIT Press, www.witpress.com, ISSN 1743-3517

work done by lines 17-20 is to update Q such that Q stores the $\mathcal{N}$ largest supports among all those of itemsets in $\mathcal{U}_{k-1} \cup C_k$. Since C_k is a superset of all k-winners with size k, $\mathcal{U}_{k-1} \cup C_k$ is a superset of all k-winners. Thus, currently, the smallest support in Q equals k-crisup and also Q stores the $\mathcal{N}$ largest supports among all those of itemsets in $\mathcal{U}_k$. Finally, the *Crisup-gen* function returns the value of k-crisup at line 21.

The priority-queue Q can be implemented by several ways. In our implementation of Algorithm 3.1, we use a heap structure (refer to [4] for details) to implement Q such that line 18 can be completed in time $O(1)$ and line 19 in time $O(\log \mathcal{N})$. Thus the execute time of Crisup-gen(Q,C_k) is in the order of $O(|C_k| * \log \mathcal{N})$.

4 Performance Study

We study the performance of Algorithm 3.1. For comparison, we use *Apriori* introduced in [1] to implement the naive solution (*Naive* for short) mentioned in Section 1, as *Apriori* is one of the best known algorithms for finding all itemsets with support exceeding minsup.

4.1 Experimental Results

To assess the performances of Algorithm 3.1 and *Naive*, we performed several experiments on a SUN ULTRA-1 workstation with 64 MB main memory running Sun OS 5.5. The synthetic datasets used in our experiments were generated by a tool described in [1]. We use the same notation $Tx.Iy.Dz$ to denote a dataset in which x is the average transaction size, y is the average size of a maximal potentially large itemset and z is the number of transactions. We refer the reader to [1] for more details on the dataset generation.

To keep the comparison fair, we implemented all the algorithms using the same basic data structures. Moreover, in our implementation of *Naive*, we call *Apriori* repeatedly with a sequence of $<$ minsup_1, $\text{minsup}_1 - \delta$, $\text{minsup}_1 - 2\delta$, $\cdots >$ until the result contains at least $\mathcal{N}$ itemsets. To keep the execution time of *Naive* as minimum as possible, we implemented an incremental version of *Apriori*: in the k-th call of *Apriori*, our implementation did not count those itemsets whose supports had already been obtained.

Figure 1 shows the experimental results for four synthetic datasets T5.I2.D100K, T10.I4.D100K, T20.I4.D100K and T20.I6.D100K to

Transactions on Information and Communications Technologies vol 19 © 1998 WIT Press, www.witpress.com, ISSN 1743-3517

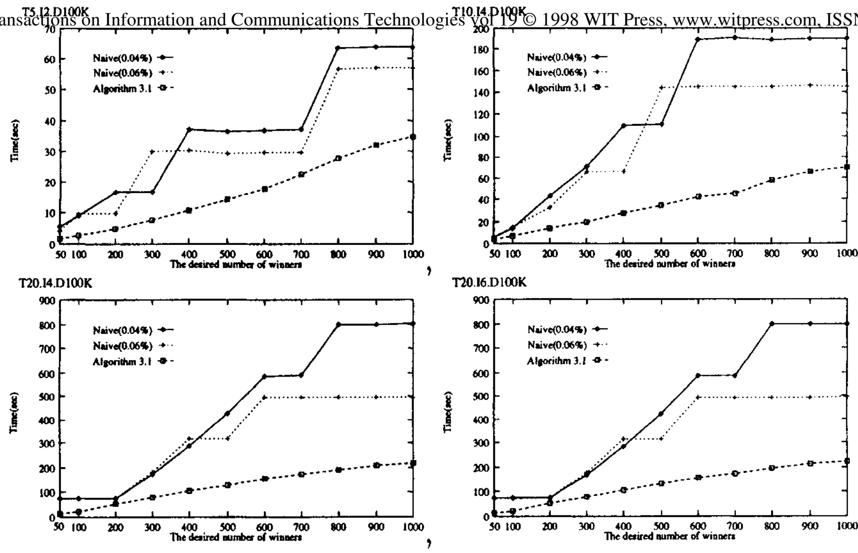

Figure 1: Performance Comparison of Algorithm 3.1 and *Naive*

compare the performances of Algorithm 3.1 and *Naive*. In the figure, *Naive(0.04%)* refers to the execution of *Naive* using $\mathsf{minsup}_1 = 2.5\% * |\mathcal{D}|$ and $\delta = 0.04\% * |\mathcal{D}|$; *Naive(0.06%)* refers to the execution of *Naive* using $\mathsf{minsup}_1 = 2.5\% * |\mathcal{D}|$ and $\delta = 0.06\% * |\mathcal{D}|$. We observe that Algorithm 3.1 outperforms both executions of *Naive* for all $\mathcal{N}$'s, by factors mainly ranging from 2 to 5.

Actually, the speed of *Naive* is largely dependent on two required parameters: minsup_1 and δ. For a small minsup_1 or a large δ, *Naive* could complete by calling *Apriori* very few times, but the execution of the last *Apriori* might take quite long time because the last minsup tends to be very small. In contrast, for a large minsup_1 and a small δ, *Naive* could complete every *Apriori* within reasonable time, but it may contain considerable tradeoff for calling *Apriori* many times because the sequence of $< \mathsf{minsup}_1, \mathsf{minsup}_1 - \delta, \mathsf{minsup}_1 - 2\delta, \cdots >$ might be relatively long so that crisup can be reached in the end. So, *how to set a reasonable sequence of decreasing* minsup*s to accelerate Naive* seems to be an interesting problem for future study. With this observation, we also did many other experiments using a wide range of values of minsup_1 and δ on *Naive*, and found that some executions are similar to the above two but most of them are far slower. In those cases, Algorithm 3.1 outperforms *Naive* further more.

Transactions on Information and Communications Technologies vol 19 © 1998 WIT Press, www.witpress.com, ISSN 1743-3517

4.2 Theoretical Analysis

We now analyze theoretically why Algorithm 3.1 outperforms *Naive*. In theory, the best case of finding $\mathcal{W}$ is to call *Apriori* with minsup=crisup because of no redundant computation in this case. Actually, the execution of Algorithm 3.1 is very close to this best case, though it has the following two extra costs: (1) the cost of finding k-crisup in the k-th iteration: as we have mentioned in Section 3, it takes $O(|C_k| * \log \mathcal{N})$ time to obtain k-crisup by calling Crisup-gen(Q,C_k) in our implementation; (2) the cost of counting supports for extra candidates: a k-itemset x is an extra candidate iff some $(k-1)$-subset of x is not a winner but all $(k-1)$-subsets of x are $(k-1)$-winners. Compared with the cost of counting supports for all candidates in C_k by a pass over $\mathcal{D}$ in the k-th iteration, the first cost of $O(|C_k| * \log \mathcal{N})$ is trivial due to $\mathcal{N} \ll |\mathcal{D}|$. Moreover, we observe that most of itemsets with large supports have relatively small sizes because any itemset is not larger than any of its subsets. Thus, usually, no big difference exists between crisup and any of k-crisups, which implies that the number of extra candidates is often bounded and the second cost is not expensive. Thus, the execution of Algorithm 3.1 is almost the same as the above best case. For *Naive*, however, it is very hard to meet the best case, as *Naive* has no effective method but guessing and testing on finding crisup. Therefore, Algorithm 3.1 has a much better performance than *Naive*.

Another drawback of *Naive* is that it not only has an exponential worst-case time complexity, but also is quite easy to run into this bottleneck because the last minsup chosen by *Naive* tends to be very small so that its result can contains enough itemsets. However, from the following analysis, we can see that Algorithm 3.1 has a polynomial time complexity at most cases, especially in practical applications.

Lemma 4.1 *Let X_k be a set of k-itemsets and X_{k+1} be a set of $(k+1)$-itemsets, where $1 \leq k < |\mathcal{I}|$. If $\bigcup_{a \in X_{k+1}} \mathcal{P}_k(a) \subseteq X_k$, then $|X_{k+1}| \leq min\{|X_k|, |\mathcal{I}|\} * |X_k|$.*

Proof Let $Y = \{a \cup b \mid a, b \in X_k\}$, $Y_{k+1} = Y \cap \mathcal{P}_{k+1}(\mathcal{I})$, $Z = \{a \cup \{b\} \mid a \in X_k, b \in \mathcal{I}\}$, and $Z_{k+1} = Z \cap \mathcal{P}_{k+1}(\mathcal{I})$. First, we have $|Y_{k+1}| \leq |Y| \leq |X_k| * |X_k|$ and $|Z_{k+1}| \leq |Z| \leq |X_k| * |\mathcal{I}|$. Next, by $Y_{k+1} \subseteq Z$, we get $Y_{k+1} \subseteq Z_{k+1}$. Thus we have $|Y_{k+1}| \leq |Z_{k+1}| \leq |X_k| * |\mathcal{I}|$. Then, by $\bigcup_{a \in X_{k+1}} \mathcal{P}_k(a) \subseteq X_k$, we have $X_{k+1} \subseteq Y_{k+1}$. Hence, $|X_{k+1}| \leq |Y_{k+1}| \leq \min\{|X_k|, |\mathcal{I}|\} * |X_k|$. □

Transactions on Information and Communications Technologies vol 19 © 1998 WIT Press, www.witpress.com, ISSN 1743-3517

Since the main cost of Algorithm 3.1 is the cost of counting supports for all candidates and $|\mathcal{D}|$ is fixed, it will be enough to just estimate the size of C, where C denotes the set of all candidates generated in Algorithm 3.1.

We assume that M_k is the number of itemsets with support equal to k-crisup and size not greater than k, where $1 \leq k \leq |\mathcal{I}|$, and M is the maximum of all $M_1 \sim M_{|\mathcal{I}|}$. Thus, we have $|\mathcal{W}_k| = \mathcal{N} + M_k - 1 < \mathcal{N} + M$. In Algorithm 3.1, C_k denotes the candidate set generated in the k-th iteration, and L_k denotes the set of all k-winners with size k by selecting from C_k using k-crisup. Thus we have $|L_k| \leq |\mathcal{W}_k| < \mathcal{N} + M$. Since C_{k+1} is generated by using L_k such that $\bigcup_{a \in C_{k+1}} \mathcal{P}_k(a) \subseteq L_k$, by Lemma 4.1, we have $|C_{k+1}| \leq \min\{|L_k|, |\mathcal{I}|\} * |L_k| < \min\{(\mathcal{N} + M), |\mathcal{I}|\} * (\mathcal{N} + M)$.

Assume that $C_1, C_2, \cdots, C_n$ are all candidate sets generated in Algorithm 3.1. Thus there is at least one winner x whose size is not less than $n-1$, i.e., $|x| \geq n-1$, because otherwise C_n couldn't be generated. Furthermore, all subsets of x are winners, and $2^{|x|}$ is the number of all subsets of x, hence we have $2^{|x|} \leq |\mathcal{W}_{|\mathcal{I}|}| < \mathcal{N} + M$, and so $n - 1 \leq |x| < \log(\mathcal{N} + M)$.

Now we can estimate the size of C: $|C| = |C_1| + |\bigcup_{k \in [2,n]} C_k| = |\mathcal{I}| + \sum_{k=1}^{n-1} |C_{k+1}| < |\mathcal{I}| + (n-1) * \min\{(\mathcal{N} + M), |\mathcal{I}|\} * (\mathcal{N} + M) < |\mathcal{I}| + \log(\mathcal{N} + M) * \min\{(\mathcal{N} + M), |\mathcal{I}|\} * (\mathcal{N} + M)$. Thus, the number of all candidates generated in Algorithm 3.1 is in the order of $O(\log(\mathcal{N} + M) * \min\{(\mathcal{N} + M), |\mathcal{I}|\} * (\mathcal{N} + M))$. Hence, the time complexity of Algorithm 3.1 is polynomial for bounded $\mathcal{N}$ and M.

In practical applications, users are usually interested only in a limited number of largest itemsets so that $\mathcal{N}$ is limited, which implies that 1-crisup $\sim$ $|\mathcal{I}|$-crisup are relatively high. Thus, $M_1 \sim M_{|\mathcal{I}|}$ are often very limited because it is easy to see that the number of itemsets sharing the same support equal to *a fixed and relatively high value* can not be large in most cases. As a result, M is limited. Therefore, in practice, Algorithm 3.1 usually runs in polynomial time.

5 Conclusions

We have proposed a new problem: *finding the $\mathcal{N}$ largest itemsets*, whose solution can be used to efficiently generate a set of interesting itemsets for mining association rules. We have presented an efficient algorithm, Algorithm 3.1, for solving this problem. Algorithm 3.1

Transactions on Information and Communications Technologies vol 19 © 1998 WIT Press, www.witpress.com, ISSN 1743-3517

has been implemented and compared with the naive solution using *Apriori* approach. We have presented experimental results as well as theoretical analysis showing that our algorithm has a much better performance than the naive solution. We have also analyzed the cost of Algorithm 3.1 and found that it usually has a polynomial time complexity in practice.

Finding a set of interesting itemsets is the main task of mining association rules. Previous study suggested obtaining them by using minsup. However, an unsuitable choice of minsup might make all the existing algorithms either obtain too many or too few interesting itemsets or run into the exponential-time bottleneck. So we have proposed an alternative method of obtaining interesting itemsets by generating the top $\mathcal{N}$ ones, which can overcome the above drawback.

Furthermore, we can also use both $\mathcal{N}$ and minsup to generate interesting itemsets, which induces a closely related problem: *finding the $\mathcal{N}$ largest itemsets with support exceeding* minsup. A slightly modification of Algorithm 3.1 can solve this problem efficiently, by initializing Q (used for calculating k-crisup) to be a priority-queue with $\mathcal{N}$ elements that are all equal to minsup instead of 1, please refer to Section 3 for details.

We observe that finding interesting patterns by using a user-specified size of the result $\mathcal{N}$ or a combination of $\mathcal{N}$ and some threshold is also applicable to the following problems: mining generalized association rules [5, 10, 11], mining path traversal patterns [3], mining sequential patterns [12], and even data mining theoretic issue [6]. We are currently working on efficient solutions for these problems.

Acknowledgment

Special thanks to Ms Ling Cheng for her helpful implementation and experiment on Algorithm 3.1 and the naive solution using *Apriori*.

References

[1] Agrawal R., Mannila H., Srikant R., Toivonen H. & Verkamo A.I., Fast Discovery of Association Rules, *Advances in Knowledge Discovery and Data Mining*, pp. 307-328, AAAI/MIT Press, 1996.

Transactions on Information and Communications Technologies vol 19 © 1998 WIT Press, www.witpress.com, ISSN 1743-3517

[2] Agrawal R., Imielinski T. & Swami A., Mining Associations between Sets of Items in Massive Databases, *Proc. of ACM SIGMOD Int. Conf.*, Washington D.C., pp. 207-216, May 1993.

[3] Chen M.S., Park J.S. & Yu P.S., Data Mining for Path Traversal Patterns in a Web Environment, *1996 IEEE Proc. of 16th ICDCS.*

[4] Cormen T.H., Leiserson C.E. & Rivest R.L., Introduction to Algorithms, Cambridge, Mass.:MIT Press; NY:McGraw-Hill, 1989.

[5] Han J. & Fu Y., Discovery of Multiple-level Association Rules from Large Databases, *Proc. of the 21st VLDB Int. Conf.*, Zurich, Switzerland, 1995.

[6] Mannila H. & Toivonen H., Levelwise Search and Borders of Theories in Knowledge Discovery, Report C-1997-8, University of Helsinki, Department of Computer Science, 1997.

[7] Mannila H., Toivonen H. & Verkamo A.I., Efficient algorithms for discovering association rules, *AAAI Workshop on Knowledge Discovery in Databases*, pp. 181-192, July 1994.

[8] Park J.S., Chen M.S. & Yu P.S., An Effective Hash-Based Algorithm for Mining Association Rules, *Proc. of ACM-SIGMOD Int. Conf.*, San Jose, CA, pp. 175-186, May 1995.

[9] Savasere A., Omiecinski E. & Navathe S., An Efficient Algorithm for Mining Association Rules in Large Databases, *Proc. of the 21st VLDB Conference*, pp. 432-444, Zurich, Switzerland, 1995.

[10] Srikant R. & Agrawal R., Mining Quantitative Association Rules in Large Relational Tables, *Proc. of ACM SIGMOD Int. Conf.*, Montreal, Canada, June 1996.

[11] Srikant R. & Agrawal R., Mining Generalized Association Rules, *Proc. of the 21st VLDB Conf.*, Zurich, Switzerland, pp. 407-419, 1995.

[12] Srikant R. & Agrawal R., Mining Sequential Patterns: Generalizations and Performance Improvements, *Proc. of the 5th EDBT Int. Conf.*, Avignon, France, March 1996.