A Sampling-Based Method for Mining Frequent Patterns from Databases

Yen-Liang Chen and Chin-Yuan Ho

Dept. of Information Management, National Central Univ, Chung-Li, Taiwan 320
{ylchen, chuckho}@mgt.ncu.edu.tw

Abstract. Mining frequent item sets (frequent patterns) in transaction databases is a well known problem in data mining research. This work proposes a sampling-based method to find frequent patterns. The proposed method contains three phases. In the first phase, we draw a small sample of data to estimate the set of frequent patterns, denoted as F^S. The second phase computes the actual supports of the patterns in F^S as well as identifies a subset of patterns in F^S that need to be further examined in the next phase. Finally, the third phase explores this set and finds all missing frequent patterns. The empirical results show that our algorithm is efficient, about two or three times faster than the well-known FP-growth algorithm.

1 Introduction

There have been many algorithms developed for fast mining of frequent patterns, which can be classified into two categories. The first category, candidate generation-and-test approach, such as Apriori [1] as well as many subsequent studies, is directly based on an anti-monotone property: if a pattern with k items is not frequent, any of its super-patterns with (k+1) or more items can never be frequent. The most famous algorithm in this category is the Apriori algorithm, which generates a set of candidate patterns of length (k+1) from the set of frequent patterns of length k and then checks their corresponding occurrence frequencies in the database. Since the algorithm needs multiple passes to generate frequent patterns and each pass requires one full scan of database, its efficiency is not satisfactory when the database contains long patterns or when the number of candidate patterns is huge. Therefore, a number of researches have been proposed to improve its performance by reducing the number of candidate patterns [2], reducing the number of transactions to be scanned [1, 2] or the number of database scans [3, 4].

Recently, another category, compress-and-projection approach, such as the FP-growth algorithm [5], has been proposed. The idea of this approach is, firstly, to build up a compressed data structure to hold the entire database in memory. Then, the database is recursively partitioned into multiple sub-databases according to the frequent patterns found so far and it would search for local frequent patterns to assemble longer global ones. The most famous algorithm in this category is the FP-growth algorithm, which use the FP-tree data structure to store the compressed database.

L. Wang and Y. Jin (Eds.): FSKD 2005, LNAI 3614, pp. 536 – 545, 2005.

Other algorithms in this category include the Pattern Repository algorithm [6], the Opportunistic Projection algorithm [7], the H-Mine algorithm [8], the DepthProject algorithm [9] and the MAFIA algorithm [10].

Interestingly but not surprisingly, the weakness of the one approach is the strength of the other approach. The major weakness of the candidate generation-and-test approach is its low efficiency. Two reasons result in this problem: (1) It usually generates a huge set of candidate patterns; (2) it may scan the database many times, especially when the database contains long patterns or dense patterns. In this regard, the compress-and-projection approach performs better because by compressing the entire database into a compressed structure in memory it eliminates the needs to access the database multiple times and by using recursive partition and projection to generate frequent patterns it eliminates the needs to generate the sets of candidate patterns. Therefore, the algorithms in the second category are usually much faster than those in the first category. However, the second approach has its own problems. The difficulty is that they may not fit in the memory when the database is huge. Besides, during the process of partition and projection, multiple copies of the database may be generated and kept in the main memory. This makes the algorithms not scalable in large databases. In this regard, the algorithms in the first category are exempted, because they don't store the compressed database in the main memory.

The comparisons above show a requirement to design an algorithm that has the advantages of both approaches but without their disadvantages. Thus, if the new algorithm is designed based on the first approach, it should meet the following requirements.

1. It needs only few database scans.
2. The set of candidate patterns should be small.
3. The performance should not be inferior to those in the second category.
4. Te entire database should not be kept in the main memory.

The goal of this paper is to propose an algorithm that meets all the above requirements. Basically, our algorithm adopts the framework proposed by Toivonen [11]. This framework consists of three phases. First, it mines a sample S of the database with a lower support threshold than the minimum support to find the frequent patterns local to S (denoted L^S). Then, the second phase scans the whole database once to compute the actual supports of each pattern in L^S. Here, we design a method to determine whether all frequent patterns appear in L^S. If they are, then one scan of database is sufficed. Otherwise, the third phase uses a second scan to find the frequent patterns that were missing in the second phase. In the original paper of Toivonen, he only gave a rough framework without specifying the implementation details.

Although our algorithm has the same framework as Toivonen's algorithm, we use a new advanced data structure to implement the framework. The data structure used in the algorithm is the all-subset tree, where each node corresponds to an itemset. This structure is similar to the lexicographical tree used in the TreeProjection algorithm [12] or the set-enumeration tree used in Max-Miner [13]. By combining sampling technique with the all-subset tree, a novel algorithm that satisfies all the above-mentioned requirements is developed. The major characteristics of this algorithm include: (1) Apart from drawing a sample, the algorithm needs at most two scans of

the database; (2) The number of nodes in the all-subset tree is the same as the size of L^S, and this size is much smaller than the size of the set of candidate patterns in the traditional Apriori-like algorithms; (3) Empirical results show that our algorithm can run about two or three times faster than the FP-growth algorithm, and; lastly, (4) The algorithm does not need to keep the database in the main memory.

The rest of the paper is organized as follows. In Section 2, we describe the problem definitions and propose the algorithm. Section 3 runs several simulations to evaluate its performance. Finally, Section 4 is the conclusion.

2 Problem Definitions and the Algorithm

Let $I=\{i_1, i_2, \ldots, i_m\}$ denote all items in database D. Each transaction $T=<e_1,e_2,\ldots,e_n>$ is a set of items, where $e_i \in I$ for all i and e_i is distinct from e_j for $i \neq j$. Let X be a pattern. Then T contains X ($T \supset X$) if every item p in X also appears in T. In database D, the percentage of transactions in database containing X is called the support of X, denoted by *support*(X). A pattern X is frequent if it satisfies *support*(X)$\geq$*minsup*, where *minsup* is specified by the user. Otherwise it is infrequent. We call the number of items in a pattern its size, and call a pattern of size k as a k-pattern. Items within a pattern are kept in lexicographic order. Besides, we use L_k to denote the set of all frequent k-patterns and C_k to denote the set of candidate k-patterns.

Since our algorithm is based on the all-subset tree, we will introduce the tree first. Basically, the tree stores a set of patterns, where each node corresponds to an item set. For example, if the actual frequent patterns for the database include {A}, {B}, {C}, {A, B}, {A, C}, {B, C}, {A, B, C}, {D}, {E}, {A, D}, {A, E}, {D, E}, {A, D, E}, then Fig. 1 shows the corresponding all-subset tree. The tree is named as the all-subset tree because all subsets of a longer pattern must exist in the tree. Note that a count field is associated with each node to record the support of the corresponding pattern. Besides, a hash structure similar to the one used in the hash tree of the Apriori algorithm [1] is attached with each node to accelerate the speed of searching.

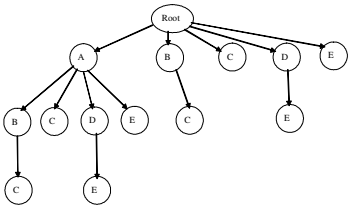

Fig. 1. The all-subset tree constructed from the actual large itemsets

The algorithm contains three phases. In the first phase, we randomly draw a small sample S of transactions from database, from which an approximation set L^S of frequent patterns is obtained. Then, we construct an all-subset tree $T1$ from L^S. Here,

two problems arise immediately about $T1$. First, the actual supports of the patterns in $T1$ are still unknown. Second, some frequent patterns in L may not appear in tree $T1$.

For ease of reference, let us call the children of the nodes in $T1$ but not in $T1$ as terminals. To solve the first problem, we need a full scan of the database. After the scan, the supports of all nodes in $T1$ are determined. To solve the second problem, we must check if there exist any terminals of the nodes in $T1$ that are frequent. If all the terminals are infrequent, then all frequent patterns must have already been in $T1$. On the contrary, if some terminals are frequent, then we must further explore these frequent terminals to find if there are any other frequent descendants spawning from them. Due to the consideration above, the second phase of our algorithm extends the tree with terminal nodes. That is, each node will have two kinds of children: those belonging to $T1$ and those not, called terminal nodes. Let us name this extended tree as $T2$. Then the second phase will scan the entire database one time. In processing each transaction, tree $T2$ is traversed and the support counts of the visited nodes, including terminal nodes and the nodes originally in $T1$, are computed. When the traversal is over, the supports of all the nodes are known, and we can find which terminal nodes are frequent. If all terminal nodes are not frequent, we are done. Otherwise, we must scan the database one more time to determine if there are any frequent patterns in the descendants of terminal nodes. That is what the third phase of our algorithm does. The following introduces these three phases in order.

2.1 The First Phase: Building $T1$ by Sampling

We draw x transactions from database in this phase, where x is a parameter specified by users. Furthermore, to ensure that the frequent patterns contained in F^S can be as complete as possible, we lower the minimum support threshold by dividing *minsup* with a parameter α, where $\alpha \geq 1$. After drawing the sample, we use the FP-growth algorithm to find F^S, because this algorithm is famous for its efficiency.

Example 1. Let us consider the database shown in Fig. 2. Suppose we have x=5, *minsup*=50% and α=1.5. Assume that we draw the first five transactions in the database. By running the FP-growth algorithm with the minimum support threshold 1.67, we find the set L^S as {{A}, {B}, {C}, {D}, {A, B}, {A, C}, {A, D}, {B, C}, {B, D}, {C, D}, {A, B, C}, {A, B, D}, {A, C, D}, {B, C, D}, {A, B, C, D}}. From L^S, we construct the all-subset tree $T1$ as shown in Fig. 3.

TID	Items	TID	Items
001	A, B, C, D	006	A, D, E, F
002	A, B, C	007	A, B, C, D, E
003	A, D, E, G	008	A, D, E, F, G
004	A, B, C, D	009	A, B, C, F
005	C, F	010	A, D, E, G

Fig. 2. The database

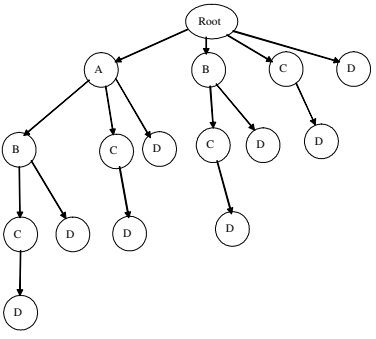

Fig. 3. $T1$ constructed from L^S

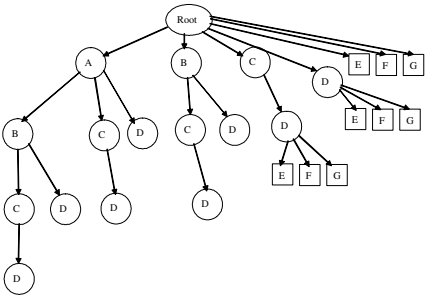

Fig. 4. Extend tree $T1$ with terminal nodes

2.2 The Second Phase: Building $T2$ and the First Scan of Database

As mentioned before, the most difficult problem is that some frequent patterns may be missing in $T1$. To remedy, we need to check if any children of the nodes in $T1$ are frequent. If there are some frequent children, then these children need to be further explored. On the other hand, if all children are infrequent, then all frequent patterns have already been included in $T1$.

Due to the reason above, we need to know not only the supports of the nodes in $T1$, but also the supports of terminal nodes, where the terminal nodes are the children of the nodes in $T1$ but are not in $T1$. Therefore, we extend the original tree $T1$ so that each node in $T1$ is expanded with a set of terminal nodes, which can be used to store the supports of these children. Let us use $T1$ in Fig.3 as an example. Since there are 7 items in the database, i.e., A, B, C, D, E, F and G, the extended tree will look like the one shown in Fig. 4, where we only show the terminal nodes for the root node, node {D} and node {C, D}.

Here, we use a method to speed up the performance. Let us observe the root node in Fig. 4, where we have three pointers pointing to the three terminal nodes and each terminal node needs a count field to store its support value. A smart reader may immediately find that these terminal nodes are not necessary, because we can use the space for storing pointers in the root to store their support values. Thus, we add a Boolean variable preceding each field to tell if this is a pointer pointing to an actual node in $T1$ or the count field of a virtual terminal node. By virtualizing tree $T1$ this way, we obtain a new tree $T2$.

The virtualization benefits the performance, because it makes the tree size very small. Suppose $|L^S|$ denote the number of frequent patterns in L^S. Then the number of the nodes in $T2$ will be also $|L^S|$, because we did not actually generate any terminal nodes. Later, when we need to traverse the tree $T2$ for every transaction, the small tree will make the traversal very efficient.

Having constructed tree $T2$, we need a full scan of the database to determine the supports of all the nodes as well as those of all virtual terminal nodes in $T2$. Once it is finished, we can output all the frequent patterns in $T2$ by depth first search. During the traversal, if we find some virtual terminal nodes that are frequent, then we must store

them into set *LT* for further processing in the third phase. Finally, if *LT* is empty, then the algorithm stops. Otherwise, we go to the next phase.

Example 2. Suppose that we use the database in Fig. 2 to traverse the tree shown in Fig. 4. Then, after scanned, the support counts of all nodes will be obtained. By using the depth first search to traverse the tree, we output the frequent patterns including {{A}, {B}, {C}, {A, B}, {A, C}, {B, C}, {A, B, C}, {D}, {E}, {A, D}, {A, E}, {D, E}, {A, D, E}}. Meanwhile, since there are some frequent terminal nodes, we keep them in *LT*={{E}, {D, E}, {A, E}, {A, D, E}} and go to the third phase.

2.3 The Third Phase: Building *T*3 and the Second Scan of Database

We perform this phase only when *LT* is not empty. In this situation, we need to further explore these frequent terminal nodes to see if they have any frequent descendants. To do so, we first construct an all-subset tree from the patterns in *LT*. After that, we insert every transaction into the tree. Note that, if a transaction does not contain any patterns in *LT*, then this transaction will insert nothing into the tree. But if there are some patterns in *LT* that are contained in the transaction, then the nodes corresponding to these patterns will grow descendant nodes.

For ease of presentation, all the nodes are classified into three different classes: non-leaf nodes, terminal nodes and newborn nodes. The terminal nodes correspond to the patterns in *LT*, the non-leaf nodes are the nodes in the paths from the root to the terminal nodes, and newborn nodes are the descendant nodes spawning from terminal nodes. The following steps show how to insert a transaction into the tree.

Subroutine *Traverse*(*u*, *T*, *i*)
Parameters: *u*: the node in the tree where we are currently located
T: the transaction that we are processing
i: the last position in the transaction that has been matched
if *u* is a newborn node then add 1 into the counter of node *u*;
if *u* is a terminal node or a newborn node then
for($j = i+1; j \le length(T); j$++)
if there is a child *v* of *u* satisfying *item*(*v*) = *item*(*T*(*j*))
then *Traverse* (*v*, *T*, *j*)
else create a newborn child *v* of *u* with item label *item*(*T*(*j*))
Traverse (*v*, *T*, *j*)
else
for($j = i+1; j \le length(T); j$++)
if there is a child *v* of *u* satisfying *item*(*v*) = *item*(*T*(*j*))
then *Traverse* (*v*, *T*, *j*);
return

Example 3. Since Example 2 has *LT*={{E}, {D, E}, {A, E}, {A, D, E}}, we perform the third phase. The first step uses the patterns in *LT* to construct the all-subset tree in Fig. 5. After that, we need to insert every transaction in Fig. 2 into the tree. During the insertion process, only transactions 003, 006 and 008 can insert the tree successfully. The other transactions will fail because they cannot match the patterns

in *LT*. Fig. 6 shows the resulting tree, where round, rectangle and round-cornered rectangle nodes denote non-leaf, terminal and newborn nodes, respectively. In addition, the number beside each newborn node is its support count.

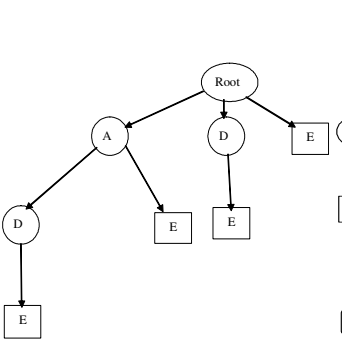

Fig. 5. Building tree from *LT*

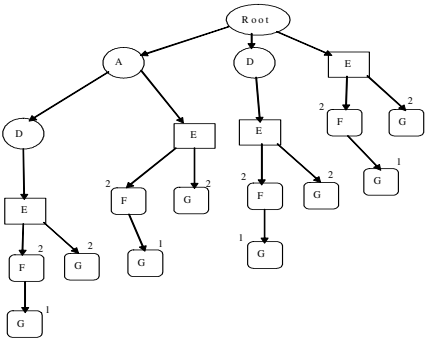

Fig. 6. After insertion

The constructed tree like Fig. 6 may produce a lot of descendents from a single terminal node. The following properties help us to prune the tree.

Property 1. Let u denote the item attached to the current node, and let w denote the item attached to an ancestor node. Then, the pattern corresponding to the current node can be frequent only if (1) $\{w, u\}$ is in L_2 and (2) u is in L_1.

Proof. The pattern corresponding to the current node is the set of items attached to the nodes along the path from the root to the current node. By the anti-monotone property, any subset of a frequent pattern must be also frequent. Therefore, both $\{w, u\}$ and u must be frequent.

Property 2. Let u denote the item attached to the current node and L' the set of frequent patterns obtained after phase 2. Then every frequent pattern in L_1 must be in L'.

Proof. Since $T2$ extends every node in $T1$ with terminal nodes, every candidate pattern in C_1 must appear in $T2$: either as an actual node or a virtual terminal node in the first level of the tree. Therefore, after computing the supports every frequent pattern in L_1 must exist in L'.

Property 3. Let L' denote the frequent patterns obtained after phase 2. If there are some frequent 2-patterns in L_2, say $\{w, u\}$, that are not in L', then w must be the item attached to a virtual node in the first level of $T2$.

Proof. If y is an actual node with item w in the first level of $T2$, then either y has a virtual child node corresponding to $\{w, u\}$ or has an actual child node corresponding to $\{w, u\}$. In both cases, $\{w, u\}$ must exist in L' after phase 2. This contradiction proves that y must be a virtual node with item w in the first level of $T2$.

Based on the properties above, before we insert a newborn node with item u, we will execute the following test, where w denotes the item attached to an ancestor node of the current node.

1 If u is not in L', then stop the insertion.
2 For every ancestor node with item w
  If (w, u) is not in L' and w is not the item attached to a virtual terminal node in the first level of $T2$, then stop the insertion.
3 Create this newborn node.

3 Experiment Results

To study the performance, our algorithm, named as the All-subset-tree algorithm, and the FP-growth algorithm are implemented by Visual C++ language and tested on a PC with Pentium-III 933 processor and 1.024G main memory under the Window 2000 operating system. We generate the synthetic datasets by applying the well-known synthetic data generation algorithm in [1]. Throughout the simulation, unless stated otherwise, we set the parameters as follows: $|L|$=2000, $|I|$=4, T=10, $|D|$=200000, *minsup* =1% and N=1000. When we need to draw a sample, we draw the data sequentially from the beginning until the number of the data records reaches the target.

At first, we determine the best combinations of the sample size x and the threshold divisor α. So, we execute the All-subset-tree algorithm for 40 different possible combinations. After an extensive study, we find that the better combinations are (4000, 1.6), (6000, 1.5) and (8000, 1.4). Therefore, the following use these three combinations as test cases.

Next, we explore how the performance varies for different minimum support thresholds. Therefore, we generate 30 data sets mentioned before, and vary the minimum support thresholds from 0.5% to 2%. Fig. 7 shows the execution times of the FP-growth algorithm as well as three different versions of the All-subset-tree algorithm. Meanwhile, Table 1 shows the average numbers of database scans required for these three versions of the All-subset-tree algorithm. The results show that all these three are faster than FP-growth, and the average number of scans is close to 1.

Table 1. Average numbers of database scans vs. minimum supports

	(4000, 1.6)	(6000, 1.5)	(8000, 1.4)
0.5%	2.00	1.73	1.77
0.75%	1.27	1.20	1.23
1%	1.03	1.10	1.13
1.5%	1.00	1.00	1.00
2%	1.00	1.00	1.00

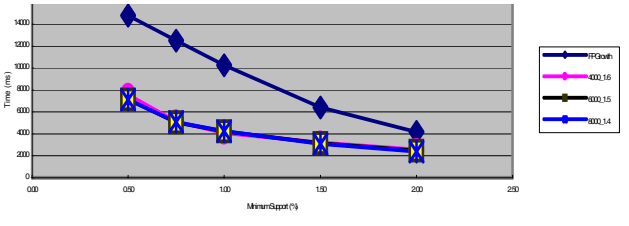

Fig. 7. Execution times vs. minimum supports

Finally, we study how the performance varies when the database size changes. Therefore, we run the comparison 30 times with different numbers of transactions, ranging from 200K to 1000K. Fig. 8 shows the execution times of the FP-growth algorithm as well as three different versions of the All-subset-tree algorithm. Meanwhile, Table 2 shows the average numbers of database scans required for these three versions. The results show that all three versions are faster than FP-growth algorithm. Moreover, it is clear that our algorithm has a better scalability than the FP-growth algorithm, because as the database size gets larger, the ratio of the needed run time of the FP-growth algorithm to that of our method gets larger as well.

Table 2. Average numbers of database scans vs. numbers of transactions

	(4000, 1.6)	(6000, 1.5)	(8000, 1.4)
1000K	1.17	1.07	1.13
800K	1.07	1.03	1.07
600K	1.10	1.03	1.07
400K	1.00	1.03	1.07
200K	1.03	1.10	1.13

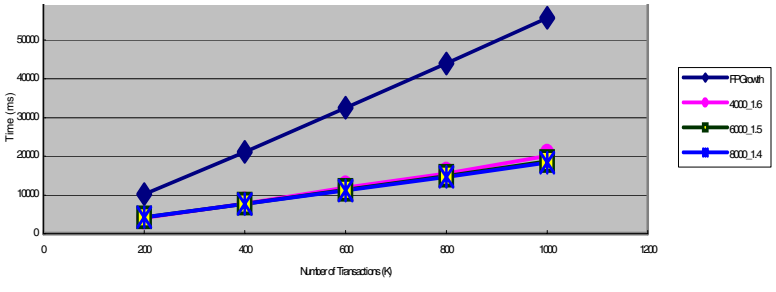

Fig. 8. Execution times vs. numbers of transactions

4 Conclusions

The goal of this paper is to develop a new efficient algorithm for mining frequent patterns. To this end, we use the all-subset tree data structure as the basis to develop a three-phase algorithm. The following are the major characteristics of the proposed algorithm.

1. The algorithm scans the database no more than twice. The experiments show that most of the time one database scan is enough.
2. Since we virtualized the all-subset tree structure, the number of the nodes in the all-subset tree equals to the size of L^S. This is much smaller than the size of the set of candidate patterns produced in the traditional algorithms such as the Apriori algorithm.
3. The algorithm is very efficient. The evaluation shows that it is about two or three times faster than the FP-growth algorithm.

4. The algorithm does not use the main memory to hold the entire database. Due to this reason, it has a better scalability in dealing with large databases.

Acknowledgment

The research was supported in part by the MOE Program for Promoting Academic Excellence of Universities under the Grant Number 91-H-FA07-1-4.

References

1. Agrawal, R., Srikant, R.: Fast algorithms for mining association rules. In: Proceedings of the 20th VLDB Conference. (1994) 478-499
2. Park, J.S., Chen, M.S., Yu, P.S.: Using a hash-based method with transaction trimming for mining association rules. IEEE Transactions on Knowledge and Data Engineering **9** (1997) 813-825
3. Brin, S., Motwani, R., Ullman, J., Tsur, S.: Dynamic itemset counting and implication rules for market basket data. In: Proceedings of the 1997 ACM-SIGMOD Conf. on Management of Data. (1997) 255-264
4. Savasere, A., Omiecinski, E., Navathe, S.: An efficient algorithm for mining association rules in large databases. In: Proceedings of Int'l Conf. Very Large Data Bases. (1995) 432-444
5. Han, J., Pei, J., Yin, Y.: Mining frequent patterns without candidate generation. In: Proceedings of SIGMOD. (2000) 1-12
6. Relue, R., Wu, X., Huang, H.: Efficient runtime generation of association rules. In: Proceedings of the Tenth International Conference on Information and Knowledge Management. (2001) 466-473
7. Liu, J., Pan, Y., Wang, K., Han, J.: Mining frequent item sets by opportunistic projection. In: Proceedings of 2002 Int. Conf. on Knowledge Discovery in Databases. (2002) 229-238
8. Pei, J., Han, J., Lu, H., Nishio, S., Tang, S., Yang, D.: H-mine: hyper-structure mining of frequent patterns in large databases. In: Proceedings of IEEE International Conference on Data Mining. (2001) 441-448
9. Agrawal, R.C., Aggarwal, C.C., Prasad, V.V.V.: Depth first generation of long patterns. In: Proceedings of SIGKDD Conference. (2000) 108-118
10. Burdick, D., Calimlim, M., Gehrke, J.: MAFIA: a maximal frequent itemset algorithm for transactional databases. In: Proceedings of 17th Int. Conf. Data Engineering. (2001) 443-452
11. Toivonen, H.: Sampling large databases for association rules. In: Proceedings of the 22th International Conference on Very Large Databases. (1996) 134-145
12. Agarwal, R., Aggarwal, C., Prasad, V. V. V.: A tree projection algorithm for generation of frequent itemsets. Journal of Parallel and Distributed Computing **61** (2001) 350-371
13. Bayardo Jr., R. J.: Efficiently mining long patterns from databases. In: Proceedings of the ACM-SIGMOD Int'l Conf. on Management of Data. (1998) 85-93