Fast Discovery of Frequent Itemsets: a Cubic Structure-Based Approach

Renata Ivancsy and Istvan Vajk
Department of Automation and Applied Informatics
Budapest University of Technology and Economics
and HAS-BUTE Control Research Group
H-1111, Goldmann Gy. ter 3, Budapest, Hungary
{renata.ivancsy,vajk}@aut.bme.hu

Keywords: frequent itemset mining, Apriori algorithm, FP-growth algorithm

Received: November 20, 2004

Mining frequent patterns in large transactional databases is a highly researched area in the field of data mining. The different existing frequent pattern discovering algorithms suffer from various problems regarding the computational and I/O cost, and memory requirements when mining large amount of data. In this paper a novel approach is introduced for solving the aforementioned issues. The contribution of the new method is to count the short patterns in a very fast way, using a specific index structure. The suggested algorithm is partially based on the apriori hypothesis and exploits the benefit of a new index table-based cubic structure to count the occurrences of the candidates. Experimental results show the advantageous execution time behavior of the proposed algorithm, especially when mining datasets having huge number of short patterns. Its memory requirement, which is independent from the number of processed transactions, is another benefit of the new method.

Povzetek:

1 Introduction

The task of association rule mining is to find hidden, previously unknown and potentially useful information in large amount of data. Since it was first introduced by Agrawal et al [1] the problem of discovering frequent patterns has received a great deal of attention. The problem is widely known as market basket analysis, however, several other applications exist which are searching for frequent recurring itemsets.

In general the process of association rule mining consists of two main steps. The first one is to discover the frequent itemsets in the dataset. The second one is to create rules from the itemsets found during the first step. Most of the existing algorithm's aim is to find the frequent itemsets, i.e. the frequent patterns in the transactions because of two reasons. The first reason is the much higher computational complexity of the frequent pattern discovery task than that of the rule generation. The frequent itemsets are discovered from the original database, which can be terabytes in size; meanwhile the rules are generated from the relatively small number of itemsets found by the first step. The second reason is that the approach of discovering frequent patterns is utilized in wide range of applications, for example for mining sequential patterns, episodes, partial periodicity and many other important data mining tasks.

The different types of frequent itemset mining algorithms suit to datasets having different characteristics. However all of them has problems either with the computational cost or the I/O activity or the memory requirement. The "candidate generate and test" algorithms, such as the Apriori algorithm [2], suffer from the problem spending much of their time to discard the infrequent candidates on each level. Another problem can be the high I/O cost which is inseparable from the level-wise approach. In case of the Apriori algorithm the database is accessed as many times as the size of the maximal frequent itemset is. Several algorithms were developed based on the Apriori method in order to enhance its performance. One of them is the DHP (Dynamic Hash and Prune) [3] algorithm which uses hash tables to collect support information about the potentially $(i+1)$-itemsets when discovering the i-itemsets. In this way the cost of generating and testing the candidates on the $(i+1)^{th}$ level is reduced. Another enhancement of the Apriori algorithm is the DIC (Dynamic Itemset Counting) [4] algorithm. It defines checkpoints in the database and scans it continuously. When a checkpoint is reached, new candidates are generated from those itemsets which are proved frequent and those are discarded which are proved infrequent since the last pass of the same checkpoint. In this way the number of the database scans can be reduced. There are several algorithms contributed [5, 6, 7, 8] to improve the performance of the Apriori algorithm that use different type of approaches. An analysis of the best known algorithms can be found in [9].

The FP-growth (Frequent Pattern-growth) [10] algorithm differs basically from the level-wise algorithms, that use a "candidate generate and test" approach. It does not

use candidates at all, but it compresses the database into the memory in a form of a so-called FP-tree using a pruning technique. The patterns are discovered using a recursive pattern growth method by creating and processing conditional FP-trees. The drawback of the algorithm is its huge memory requirement which is dependent on the minimum support threshold and on the number and length of the transactions. [11] suggest a variant of the FP-growth algorithm, such that the memory cost of building conditional FP-trees are minimized due to building a so-called COFI-tree for each frequent item. Another memory resident algorithm is the H-mine algorithm [12] which represents the transactions as a list of elements in the memory. The traversal of the lists is helped with some header tables. A further memory-based frequent itemset counting algorithm was introduced in [13]. One advantage of the memory resident algorithms is that the number of the database accesses is independent from the size of the maximal frequent itemset. Unfortunately, the size of the memory is a function of the number of transactions.

The method proposed in this paper belongs to the Apriori-like algorithms, thus it uses candidates, but it has the advantage counting and testing them quickly using an index structure. Its other advantage is the relatively small memory requirement that is dependent on the minimum support threshold and on the item number.

The organization of the paper is as follows. [1] Section 2 defines the association rule mining problem. In Section 3 two of the most common association rule mining algorithms are described in detail, a level-wise "candidate generate and test" method, and a memory-based algorithm. The execution behavior of the presented algorithms is analyzed in Section 4. After drawing the conclusion of the experiments a new method is suggested and described in detail in Section 5. Some experimental results are shown in this section as well. Conclusion can be found in Section 6.

2 Problem statement

Frequent pattern mining is one of the most fundamental data mining tasks. It is used besides several applications mainly in association rule mining algorithms. This section formally introduces the problem of association rule mining and defines the most important terms in this field.

The association rule mining problem is defined as follows. Let $I = \{i_1, i_2, \ldots, i_n\}$ be the complete set of items appearing in the transactions, where n denotes the maximum number of items. An itemset is a non-empty subset of I, and if the length of the itemset is k, then it is called k-itemset. A transaction T is a set of items such that $T \subseteq I$. Each transaction in the database has an identifier, called TID. A transaction T contains an itemset X if and only if $X \subseteq T$. The support of the itemset X, denoted as $\sigma(X)$, is defined as the percentage of the transactions in the database which contain X.

An association rule is an implication of the form $X \rightarrow Y$ where both X and Y are itemsets, and there exists no item which appears both in X and in Y, formally, $X \subset I$, $Y \subset I$ and $X \cap Y = \emptyset$. An association rule has two properties: the support and the confidence. The support of the rule $X \rightarrow Y$ equals to the support of the itemset XY. The confidence, denoted with c, is the percentage of the transactions in the database containing X that also contain Y. This is taken as a conditional probability, $P(Y|X)$.

In order to reduce the search space and to discover only those rules which can be interesting for the user, two thresholds are introduced, the minimum support and the minimum confidence thresholds. An itemset is frequent if its support exceeds a user-defined minimum support threshold, σ_{min}. However the support and the minimum support threshold are defined as percentage, the algorithms convert them to an integer value (sup) using the number of transactions N. In this way calculating the support of the itemset is only counting its occurrences in the transactions, and it can be easily compared to the integer minimum occurrence threshold ($minsup$). The rules are created only from frequent itemsets. The rule is only a valid rule if its confidence exceeds the minimum confidenc threshold ($minconf$).

3 Basic Algorithms

The frequent itemset mining algorithms can be classified regarding several aspects. One of the most distinctive features of the methods is whether they use candidates. Another aspect of the classification can be the number of the database scans because of the high cost of the I/O activity. Regarding these aspects two basic algorithms are explained in this paper whose approaches are fundamentally different. The first algorithm, introduced in Subsection 3.1, is the Apriori algorithm that is a basic level-wise method and uses candidates to discover the frequent itemsets. The other algorithm, presented in Subsection 3.2, is the FP-growth algorithm which is a two-phase method and does not use any candidates to generate the patterns. The two algorithms are selected for presentation because their importance in the data mining field. Before introducing the algorithms in detail some assumptions are needed to explain. These assumptions without loss of generality make easier to handle the problem. Firstly it is assumed that the items are presented with continuous integers. The other supposition is that the items are in lexicographic order both in the transactions and in the candidates. If it is not the case the conversions can be done during a preprocessing step.

3.1 Apriori algorithm

The most commonly known, and the first presented association rule mining algorithm is the Apriori algorithm introduced by Agrawal et al in [2]. Since its introduction several other algorithms were presented which are based on it.

[1] Short version of this paper is presented in [14].

The main idea of the algorithm is based on the a priori hypothesis, namely, an itemset can only be frequent if all its subsets are also frequent. In other words, if an itemset is not frequent, no superset of it can be frequent. Exploiting this knowledge makes possible to reduce the search space efficiently when discovering the frequent itemsets, because using this knowledge the number of the candidates can be reduced. The Apriori algorithm is a level-wise method, which means that it discovers the k-itemsets during the kth database scan.

The algorithm works as follows. During the first database scan the items in the transactions are counted and the infrequent ones are discarded. In this way the frequent 1-itemsets are found. From these frequent items two candidates are generated by creating all the combination of them by keeping the lexicographic order. Formally, the items x and y form a candidate (x, y) when $x \leq y$. During the second database scan the support of the 2-candidates are counted. After a database reading the counters of the candidates are checked whether they are over the minimum support threshold. If a value of a counter exceeds the threshold, the candidate belonging to it becomes frequent, otherwise it is filtered out. The 3-candidates are generated from the frequent 2-itemsets regarding the following rule. Let be given two itemsets (i_1, i_2) and (i_3, i_4) where $i_1 < i_2$ and $i_3 < i_4$ as mentioned earlier. The two itemsets can form a 3-candidate if $i_1 = i_3$ and $(i_2, i4)$ is also frequent. Fulfilling the second condition means that the a priori hypothesis is fulfilled. The resulting 3-candidate is the following: (i_1, i_2, i_4). In general two k-itemsets are joined by keeping the lexicographic order to form a $(k+1)$-itemset if the first $k-1$ items of them are in common and all the $(k-1)$-subsets of the resulting candidate are frequent as well. The algorithm terminates if no candidates can be generated or no frequent itemsets are found. The pseudo code of the algorithm is depicted in Table 1 and Table 2.

```
procedure Apriori(minsup)
L_1 = find frequent 1-itemsets
for (k = 2; L_{k-1} != null; k++)
   C_k = AprioriGen(L_{k-1})
   for each transaction t do
      C_t = subset(C_k, t)
      for each candidate c in C_t do
         c.counter++
      for each c in C_k do
         if c.counter >= minsup then
            L_k.Add(c)
return C_k
```

Table 1: Pseudo code of the Apriori algorithm

3.2 FP-growth algorithm

One of the algorithms which do not use any candidates to discover the frequent patterns is the FP-growth (Frequent Pattern Growth) algorithm proposed in [10]. The other main difference to the Apriori algorithm is the number of the database readings. While the Apriori is a level-wise algorithm the FP-growth is a two-phase method. It reads the database only twice and stores the database in a form of a tree in the main memory.

```
procedure Apriori(minsup)
for each itemset l_1 in L_{k-1} do
   for each itemset l_2 in L_{k-1} do
      if l_1[1] = l_2[1]
         and l_1[2] = l_2[2]
         and ... and l_1[k-2] = l_2[k-2]
         and l_1[k-1] < l_2[k-1]
      then
         c = l_1 join l_2
         if c has infrequent subset
         then DELETE c
         else C_k.Add(c)
return C_k
```

Table 2: Pseudo code of the AprioriGen procedure

The algorithm works as follows. During the first database scan the number of occurrences of each item is determined and the infrequent ones are discarded. Then the frequent items are ordered descending their support. During the second database scan the transactions are read and the frequent items of them are inserted into a so-called FP-tree structure. In this way the database is pruned and is compressed into the memory. The aim of using the FP-tree is to store the transactions in such a way that discovering the patterns can be achieved efficiently.

Each node in the tree contains an item, a counter to count the support, and links to the child nodes, to the parent nodes and to the siblings of the node. The rule for constructing the FP-tree is as follows. When reading a transaction its infrequent items are omitted and the frequent ones are ordered regarding their support. The transaction is then inserted into the tree. If the tree is empty the transaction is inserted as the only branch in the tree. If it is not empty, while the first k items of the transaction fit the prefix of one of the branches of the tree, a counter is incremented in each referred node in the tree. From the $(k+1)^{th}$ item, a new branch is created as a child of the node, which corresponds to the kth item in the transaction, and the further items in the transactions are inserted as this new branch with a support counter set to one. A header belongs to the FP-tree which contains the sorted 1-frequent items, their supports and a pointer to the first occurrence of the given item in the tree. The other occurrences of the given item in the tree are linked together sequentially as a list.

The FP-tree is processed recursively by creating several so-called conditional FP-trees. This is the recursive pattern growth method of the algorithm. When a conditional FP-tree contains exactly one branch the frequent itemsets are generated from it by creating all the combinations of each

items. When traversing the whole FP-tree, all the frequent itemsets are discovered. The pseudo code of the FP-growth algorithm is depicted in Table 3.

```
procedure FPGrowth(Tree, α)
if Tree contains a sigle path P then
   for each β = comb. of nodes in P do
      pattern = β ∪ α
      sup = min(sup of the nodes in β)
else
   for each a_i in the header of Tree do
      generatepattern = β ∪ α
      sup = a_i.support
      construct β's conditional pattern base
      FPTree = construct β's
                  conditional FP-tree
      if FPTree != 0 then
         FPGrowth(FPTree,β)
```

Table 3: Pseudo code of the FP-growth algorithm

4 Comparison of the Algorithms

The experimental results presented in this paper are performed on semantic datasets generated by the dataset generator downloaded from the IBM website. The datasets generated with this program accomplish the conditions introduced in [2]. The algorithms were implemented in $C\#$. The simulations were executed on a Pentium 4 CPU, 2.40 GHz, and 1GB of RAM computer on .NET Framework v1.1. The naming conventions of the datasets are shown in Table 4. The number of the items that can occur in the transactions is 1000.

Parameter	Meaning
T	Average length of the transactions
I	Average size of maximal frequent itemsets
D	Number of transactions
K	Thousand

Table 4: Meaning of the parameters in the names of the datasets

In order to find a more effective algorithm to solve the frequent itemset mining problem in a given range of the parameters the behavior of the most representatives algorithms should be investigated. After detecting their drawbacks a novel method can be developed which aim is to avoid the disadvantages found by the algorithms examined. The objectives of the investigation are the execution time behavior and the memory requirements of each methods.

A major aspect of the examination is which parameters of the dataset affect the behavior of the algorithms significantly. The two main parameters of the datasets are the number of items that can appear in the transactions, denoted with n, and the number of transactions, denoted with T.

Fig. 1 shows the execution times of the two algorithms as a function of the number of transactions. It can be easily concluded that the execution time dependency of the Apriori algorithm on the number of transactions is linear, and that of the FP-growth algorithm is rather a polynomial of two degree. The memory requirement of the two algorithms is depicted in Fig. 2 as a function of the number of transactions. It is obvious, that the memory requirement of the Apriori algorithm does not depend on the number of transactions. The reason for that can be found in the "candidate generate and test" approach. The number of the candidates does not depend on the number of transactions; it depends only on the item number and on the minimum support threshold.

The memory requirement of the FP-growth algorithm increases significantly with the growth of the number of transactions. The reason for this can be found when examining the sizes of the trees which are generated by the algorithm. If the algorithm mines two datasets with the same statistical properties but the one contains an order of magnitude more transactions than the other, the first FP-tree built by the FP-growth algorithm contains an order of magnitude more nodes in the former case than in the latter. However the rules that have been found are nearly the same. From this fact we can draw the conclusion that several redundant nodes are in the FP-tree when increasing the number of the transactions. The claim is laid to modify the algorithm so that the created tree does not contain as redundant nodes as in the original case. The function between the number of transactions and the size of the first generated tree is linear, which is shown in Fig. 3 by different minimum support thresholds.

The advantage of the FP-growth algorithm is the quick mining process which does not use candidates. Its drawback is, however, that the memory requirement of the algorithm is huge, especially by lower minimum support threshold. The main problem of the "candidate generate and test" methods is the computational cost when filtering out the infrequent itemsets. Fig. 4 shows the execution time of the Apriori algorithm by itemset levels when using T20I7D200K dataset. When investigating the execution times by itemset levels the fact is proved that the algorithm uses most of its time to discover the small frequent itemsets. In general it uses more than 70% of its execution time to discover the 4-frequent itemsets, and more than 50% of this time is used to find the 2-frequent itemsets. Its reason is the huge number of candidates in the first four levels. The candidate numbers in each single level are depicted in Fig. 5. It can be seen well that the number of the candidates in the second level is two orders of magnitude higher than in the further levels, however the number of the frequent itemsets, depicted in Fig. 6, are about the same.

The Apriori algorithm stores the candidates in a hash tree

in order to quick find those candidates, which are to be checked whether they are contained by a certain transaction. The benefit of using a hash-tree is to reduce the number of candidates to be checked when processing a transaction. During a database scan each transaction is processed and its subsets are checked whether a counter belongs to it in the hash tree or not. This method is faster than finding the candidates by linear search, but in case of huge candidate number, using a hash tree is inefficient. The number of the database accesses of the Apriori algorithm equals to the size of the maximal frequent itemset. It accesses the database k times even than when only one k-frequent itemset exists. If the dataset is huge, the multiple database scans can be one of the drawbacks of the Apriori algorithm.

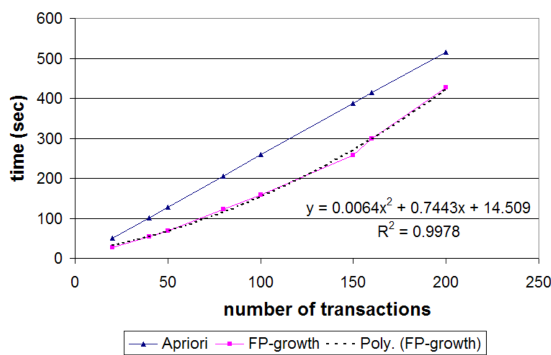

Figure 1: Execution time of the two algorithms as a function of the number of transactions by 0.9% minimum support threshold

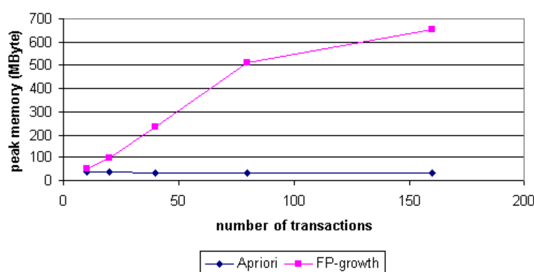

Figure 2: Peak memory of the two algorithms as a function of the number of transactions by 0.9% minimum support threshold

5 Cubic algorithm

The previous section describes the advantages and the disadvantages of the Apriori and the FP-growth algorithms. The main motivation of the novel method, called Cubic, is to enhance the aforementioned algorithms both regarding the execution time behavior and the memory requirement.

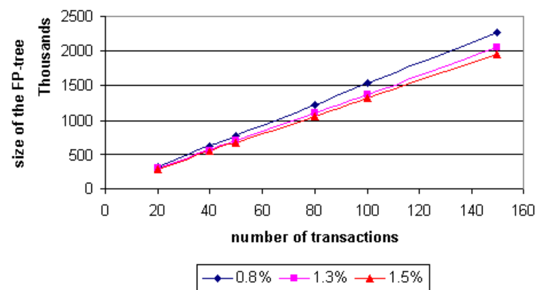

Figure 3: Sizes of the first generated FP-tree as a function of the number of transactions by 0.8%, 1.3% and 1.5% minimum support thresholds

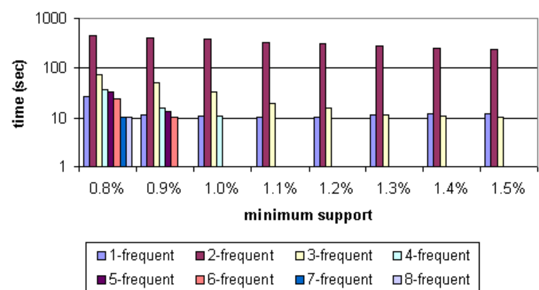

Figure 4: 7 Execution time on each level of the Apriori algorithm when using T20I7D200K dataset

The aim was to develop an algorithm whose memory usage is significantly lower than that of the FP-growth algorithm, and its execution time is smaller than the execution times of both of the algorithms described earlier. The new method is based on the Apriori algorithm, its aim is to enhance the discovering of the small patterns. Thus the novel method is faster than the introduced algorithms especially in those cases when the characteristics of the dataset shows much more small patterns than long ones.

5.1 Description of the algorithm

The Cubic algorithm is a novel method to find the frequent 4-itemsets quickly. It discovers the 4-itemsets in only two database scans. During the first disk access the support of the one and two itemsets are counted using an upper triangular matrix M. If n stands for the cardinality of the items in the database, then the size of M equals to $\frac{n(n+1)}{2}$. The diagonal elements of the matrix contains counters for the items, and the other cells are counters for the item pairs. The support counting can be achieved by a direct indexing method using the matrix and in this manner it is the fastest way.

The three and four frequent itemsets are counted during a further database scan. For efficient counting the support of the candidates their counters are stored in an index table-

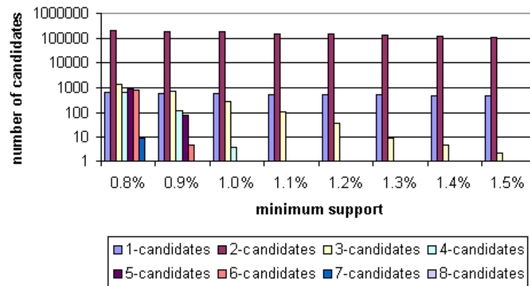

Figure 5: Sizes of the candidates in each level when using T20I7D200K dataset

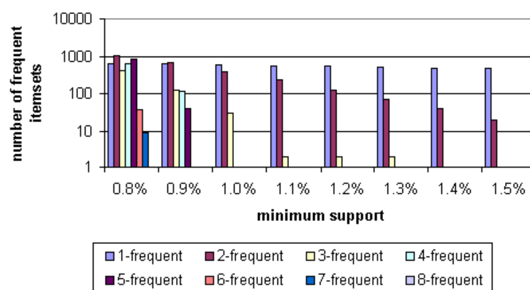

Figure 6: Sizes of the frequent itemsets in each level when using T20I7D200K dataset

based cubic structure. This is built when traversing the matrix M. A cube is created for those rows of the matrix whose value of the diagonal element is over the minimum support threshold. It means, that one cube is created in order to store the 3 and 4-candidates which belong to the frequent 2-itemsets beginning with the same item. In this manner the first item of a candidate selects the appropriate cube and the further items addresses the cells in the cube.

The matrix M is processed by rows. The ith row is only processed, if the value of the ith diagonal element in M is greater than the minimum support threshold. In this case a new index table is created with size of n, and the values in the ith row are checked whether they are over the threshold or not. If $M[i, j] > minsup$, $(i < j)$, the jth element in the index structure is set to the number which will later index the cube. If all the elements of the ith row are checked, a cube is created. The size of the cube is the number of the frequent item pairs in the ith row. A reverse index is created as well, in order to easy converting the index value, which addresses the cells in the cube, to the original item when traversing the cubes. This is used by the counter checking process. During the second database scan every 3 and 4-subsets of the transactions are created, which has at least one 2-frequent subset, and the appropriate element in the cube is incremented. The cube is selected by the first item of the subset. The other items address the counters in the cube using the index structure belonging to the selected cube.

The Apriori hypothesis is used only partially because of the following reason. The Apriori assumption is exploited when the algorithm creates different cubes for the itemsets having different first item. However it is not used when the edges of the cube are created. If the value of $M[i, j]$ is greater than the minimum support threshold, the item j is added to the index table of the cube independently whether the elements $M[j, s]$, $(i < s < n)$ are greater than the minimum support or not, where s denotes those items which satisfy the $M[i, s] > minsup$ condition. The reason for this is that the storage space for the cube is rather compact, and there would not be any benefit discarding these items. In addition it would take more time to discard the item than to count its support. The main parts of the algorithm are depicted in Table 5 and Table 6.

The Cubic algorithm discovers the 4-frequent itemsets. The further itemsets can be found in different ways. One of the possibilities is a level-wise approach, which simply invokes the Apriori algorithm. This is the easiest way and often a very quick solution because the Apriori algorithm finds the itemsets with cardinality greater than five relatively quick. This algorithm is called Cubic-Apriori.

Another way is to call the FP-growth algorithm after discovering the frequent 4-itemsets. The FP-tree should be created by leaving out those transactions, which do not contain frequent 4-itemsets. So the basic idea of the suggested Cubic FP-growth algorithm is that there is no need to build a much larger tree, if the rules are contained also in a smaller. In this case the FP-tree must be generated only from those transactions, which contains at least one 4-frequent itemset. In this way the profit is the smaller tree generated by the FP-growth algorithm, thus, in general, the execution time is enhanced as well. In addition only one additional database scan is needed in this case than in case of using the original FP-growth algorithm.

5.2 Simulation results

In Fig. 7 the execution time of the four algorithms is analyzed when using T20I7D200K dataset. It is clear, that the Cubic method continued by the Apriori algorithm, called Cubic Apriori algorithm, is the fastest of all the four methods. The execution time of the Cubic FP-growth method is always smaller than that of the Apriori algorithm but it is not always smaller, than the execution time of the FP-growth algorithm. The reason for that is illustrated in Fig. 8. The sizes of the first generated FP-trees are depicted in it in cases of the FP-growth and of the Cubic FP-growth algorithms when using T20I7D200K dataset as a function of the minimum support threshold. Apparently the sizes of the tree in case of small minimum support thresholds are near to each other, moreover by minimum support threshold of 0.5% they are about the same. It means that the Cubic FP-growth algorithm has to accomplish about the same recursive pattern growth process as the FP-growth algorithm does, but before this, the Cubic FP-growth al-

```
procedure FillCubes()
for each transaction t do
  for (i=0; i < t.count; i++)
    if ixStruct[t[i]] = null then
      continue
    for (j=i + 1; j < t.count; j++)
      if M[t[i], t[j]] < minsup then
        continue
      ix1 = IxStruct[t[i]][t[j]]
      for (k = j + 1; k < t.count; k++)
        ix2 = IxStruct[t[i]][t[k]]
        if ix2! = −1 then
          CubeL[t[i]][ix1, ix2, 0]++
          for (l = k + 1; l < t.count; l++)
            ix3 = IxStruct[t[i]][t[l]] + 1
            if ix3! = 0 then
              CubeL[t[i]][ix1, ix2, ix3]++
```

Table 5: Pseudo code of the candidate counting procedure of the Cubic algorithm

gorithm has also to mine the 4-frequent itemsets using the Cubic method. In this case filtering the transactions by using the results of the Cubic algorithm causes no significant profit regarding the number of nodes in the tree. The saving in the node number is rather by minimum support threshold higher than 0.7%. In Fig. 9 the peak memory sizes are illustrated as a function of the number of transactions when the average size of the maximal frequent items is 7 and the average size of the transactions is 20. The minimum support threshold is set to 0.9%. It is shown, that the memory requirement of the Cubic Apriori algorithm does not depend on the number of transactions.

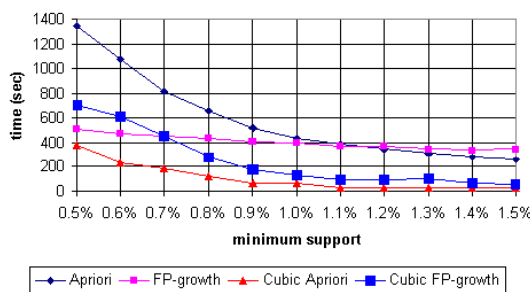

Figure 7: Execution time of the four algorithms when using T20I7D200K

6 Conclusion

This paper is concerned with the problem of efficiently discovering frequent itemsets in transactional databases. The algorithms dealing with this type of data mining problem can be divided into several classes regarding their behavior.

```
procedure CheckCubes()
rI=reverseIndexTable.Clone()
for (i=0;i < cubeL.count;i++)
  if cubeL[i]!=null then
    for (j=0;j < rI[i].count;j++)
      for (k=j + 1;k < rI[i].count;k++)
        if cubeL[i][j; k; 0] >= minSup
        then
          item2 = rI[i][j]
          item3 = rI[i][k]
          L3.Add(i, item2, item3)
          for (l=k + 1;l < rI[i].count;l++)
            if cubeL[i][j, k, l] > minSup
            then
              item2 = rI[i][j]
              item3 = rI[i][k]
              item4 = rI[i][l − 1]
              L4.Add(i, item2, item3, item4)
```

Table 6: Pseudo code of the candidate checking procedure of the Cubic algorithm

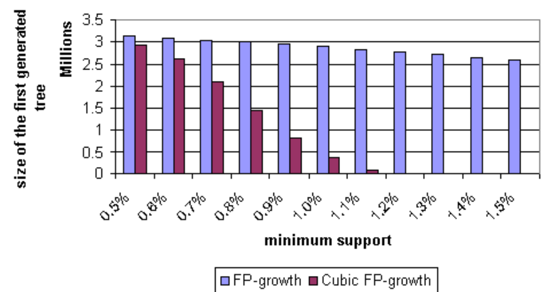

Figure 8: Sizes of the first generated tree of the FP-growth and of the Cubic FP-growth algorithm when using T20I7D200K

The two most representative classes are the one which contains the level-wise methods and the class that contains the two-phase methods. Two basic algorithms of these classes were explained in detail. After investigating the execution time behavior and the memory requirement of the Apriori and the FP-growth algorithm the advantages and disadvantages of them were illustrated. The main drawback of the Apriori algorithm is its relatively slow candidate testing method using the hash-tree data structure in case of small candidates, when the number of these candidates is high. The memory requirement dependency on the number of transactions is proved as the major problem of the FP-growth algorithm.

A novel method, the Cubic algorithm is presented in order to enhance the Apriori algorithm by finding the short frequent patterns quickly, using an index table-based cubic structure. The algorithm exploits the benefits of direct indexing over the hash tree-based searching. Experimental

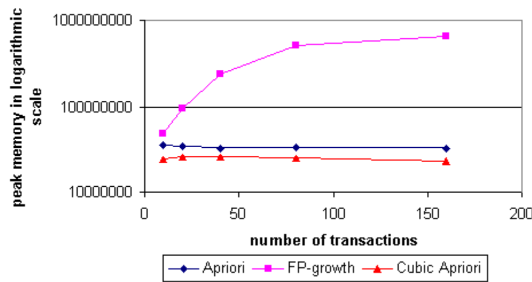

Figure 9: Peak memory of the algorithms as a function of the number of transactions by 0.9% minimum support threshold

results show the time saving when replacing the first four steps of the Apriori algorithm with the novel method. In this way, the Cubic Apriori algorithm is even faster than the FP-growth algorithm, and the memory requirement of the novel method does not depend on the number of transactions.

Using the Cubic algorithm the performance of the FP-growth algorithm can be enhanced as well. When a preprocessing step is inserted before the FP-growth algorithm, namely discovering the frequent 4-itemsets using the Cubic algorithm, the size of the FP-tree can be reduced. In this case the memory requirement is reduced.

Acknowledgement

This work has been supported by the fund of the Hungarian Academy of Sciences for control research and the Hungarian National Research Fund (grant number: T042741)

References

[1] R. Agrawal, T. Imielinski and A.Swami (1993) Mining association rules between sets of items of large databases, *Proc. of the ACM SIGMOD Intl'l Conf. On Management of Data*,Washington D.C.,USA, pp. 207–216.

[2] R. Agrawal and R. Srikant (1994) Fast algorithms for mining association rules, *Proc. 20th Very Large Databases Conference*, Santiago, Chile, pp. 487–499.

[3] J. S. Park, M. Chen, and P. S. Yu (1995) An effective hash based algorithm for mining association rules, *Proc. of the 1995 ACM Int. Conf. on Management of Data*, San Jose, California, USA, pp. 175–186.

[4] S. Brin, R. Motawani, J.D. Ullman and S. Tsur (1997) Dynamic Item set counting and implication rules for market basket data, *Proc of the ACM SIGMOD Intl'l Conf. On Management of Data*, Tucson, Arizona, USA, pp. 255–264.

[5] M. J. Zaki (2000) Scalable algorithms for association mining, *IEEE Transaction on Knowledge and Data Engineering. Vol 12. No 3. May/June 2000*, pp. 372–390.

[6] V. S. Ananthanarayana, D. K. Subramanian and M. N. Murty (2000) Scalable, distributed and dynamic mining of association rules *Proceedings of the 7th International Conference on High Performance Computing - HiPC 2000*, Bangalore, India, pp. 559–566.

[7] R. J. Bayardo (1998) Efficiently mining long patterns from databases *Proceedings of the ACM SIGMOD international conference on management of data*, Seatle, WA, pp 85–93.

[8] P. Shenoy, J.R. Haritsa, S. Sundarshan, G. Bhalotia, M. Bawa and D. Shah (2000) Turbo-charging vertical mining of large databases *Proceedings of the ACM SIGMOD*, Dallas, TX, pp. 22–33.

[9] R Ivancsy, F. Kovacs and I. Vajk (2004) An Analysis of Association Rule Mining Algorithms, *In CD-ROM Proc. of Fourth International ICSC Symposium on Engineering of Intelligent Systems (EIS 2004)*, Island of Madeira, Portugal.

[10] J.Han, J. Pei and Y. Yin (2000) Mining frequent patterns without candidate generation, *Proc. of the 2000 ACM-SIGMOD Int'l Conf. On Management of Data*, Dallas, Texas, USA, pp. 1–12.

[11] M. El-Hajj and O. R. Zaïane (2003) Non Recursive Generation of Frequent K-itemsets from Frequent Pattern Tree Representations, *Proc. of 5th International Conference on Data Warehousing and Knowledge Discovery (DaWak'2003)*, Prague, Czech Republic

[12] J. Pei, J. Han, H. Lu et al (2001) H-Mine: Hyper-structure mining of frequent patterns in large databases, *In Proceedings of the 2001 IEEE International Conference on Data Mining (ICDM'01)*, San Jose, California, pp. 441–448.

[13] Q. Zou, W. Chu, D. Johnson and H. Chiu (2002) Pattern decomposition algorithm for data mining of frequent patterns *Journal of Knowledge and Information System, Volume 4, Issue 4* pp. 466–482.

[14] R. Ivancsy and I. Vajk (2004) Fast Discovery of Frequent Patterns in Market Basket Data *In. Proc. of 4th International Conference on Intelligent Systems Design and Applications (ISDAŠ04)*, Budapest, Hungary, pp. 575–580.