FREQUENT PATTERN MINING IN TRANSACTIONAL AND STRUCTURED DATABASES

GYAKORI MINTÁK BÁNYÁSZATA TRANZAKCIÓS ÉS STRUKTURÁLT ADATBÁZISOKBAN

Ph.D. Thesis

Renáta Iváncsy

Advisor:
István Vajk Ph.D.
Renáta Iváncsy

Budapest University of Technology and Economics
Faculty of Electrical Engineering and Informatics
Department of Automation and Applied Informatics

1111 Budapest, Goldmann Gy. tér 3.

e-mail: renata.ivancsy@aut.bme.hu
tel: +36(1)4631668
fax: +36(1)4633478

Advisor:
István Vajk Ph.D.
1 Introduction

The propagation of the usage of databases and the expansion of the information community furnished a basis for developing a new interdisciplinary research field, namely, for developing the research field of data mining.

The process of data mining is to find hidden, previously unknown and potentially useful information in large amounts of data. Since the application of information systems became general in all situations of every day life, the importance of dealing with the huge amounts of stored data is, without doubt, a key issue of the research and the practice. Data mining is an iterative and interactive process, where also human interaction is needed between certain steps. Thus there is no known solution for performing data mining automatically. However, there are some subtasks which can be solved partially or totally in an automatic way.

The field of data mining can be classified into many areas, for example frequent pattern discovery, classification, clustering, outlier analysis, prediction. Frequent pattern mining is an essential part in data mining. The aim of discovering frequent patterns in large databases is to find patterns which appear frequently in the whole dataset that can be used in several tasks, for example in association rule mining, in classification and so on. The thesis work deals with the problem of discovering frequent patterns in transactional and in structured databases.

The thesis work deals with three main areas of frequent pattern mining: frequent itemset, frequent sequence and frequent subtree mining. Some of these areas are based on others forming more and more complex open problems.

In case of frequent itemset mining the aim is to determine the items that appear together frequently. A typical example for this problem is the market basket analysis, where the transactions are the market baskets of the customers, and the items are the goods bought by the customer. A transaction contains the goods bought by the customers without information about the quantity.

Several application areas are known for frequent sequence mining, for example text mining, DNA sequence mining, Web log mining etc. However, sequence mining can also be interpreted as the generalization of itemset mining, where the customers are distinguished from each other. Thus further information can be gained from the database as in case of frequent itemset mining. The frequent patterns found in this case are the sequence of frequently bought goods.

In many cases the real world problems can not be modeled with itemsets or sequences, thus, a more complex structure is needed, like labeled graphs. Many structures can be described with labeled graphs, such as chemical compounds, XML or HTML documents, the structure of a Web portal and many more. A subproblem of the general graph mining problem is to discover frequently appearing subtrees in a tree database. Because trees are adequate models for other several problems, and they can be handled easier than general graphs, thus, mining frequent trees are researched with justification.
2 Research Methodology

In my research my goal was to find more efficient solutions, approaches and algorithms for different data mining problems (frequent itemset, sequence and tree discovery), than the existing algorithms can supply.

The efficiency of an algorithm can be investigated from many aspects. In case of algorithms and softwares, basically two features are aimed of in my investigation. This two features are the execution time and the memory requirements. In my work these two properties were also the subjects of my investigation.

In case of a task having a given complexity, it can be said that either the execution time of the memory requirement can be minimized, but the two in the same time cannot. For this reason the concept of efficient algorithms is a relative concept. We have to investigate how much time and memory is available for a solution of a certain problem. An optimal solution under these conditions can be found only after these investigations. Of course, most of the researchers aims at finding algorithms that are efficient in both aspects.

After I have elaborated the related work, I classified the most important algorithms, regarding different aspects. The classification of these algorithms can help us to understand the different approaches used by them, and forms a basis for developing a novel approach. My classifying and analyzing work can be found in [4][10][18].

After the classification, I investigated the execution time and the memory requirements of the most important algorithms. Using analytical methods I determined the most important advantages and disadvantages of the algorithms and the reason for these drawbacks. This work formed the basis for further research and novel approaches.

In all the three mentioned problem areas (itemset, sequence and tree mining) it is a general observation that the algorithms can be classified into two main classes. In the two classes belong the level-wise and the database projection-based algorithms. The drawback of the level-wise methods is their high I/O cost, however they have the advantageous property using relatively small amounts of memory, which is independent of the number of the transactions. The database projection-based algorithms project the database into the main memory after one or two database reading in a clever way. Their result is the high memory requirements that is dependent of the number of the transactions. However, the advantage of this kind of algorithms is their low execution time in case when the projected database fits in the memory.

By investigating the general memory capacity of the computers, and the probable sizes of the databases to be mined, I decided to use level-wise algorithms that do not use so huge memory. A crucial point of view in the choice was the memory behavior of the database projection-based algorithms that I assumed a problematic property of these algorithms.

Taking into consideration the properties mentioned before, I developed algorithms which are based on the level-wise approach, but they can exploit the benefit of having nowadays more memory capacity than earlier.
3 Novel Scientific Results

The results of my research work are summarized in three theses according to the three different frequent pattern mining problems, namely, frequent itemset, frequent sequence and frequent tree mining problems.

The first thesis contains two algorithms which solve the frequent itemset mining problem in an efficient way. I have shown how a novel coding mechanism and an index structure can be used in order to efficiently determine the support of the small candidates. The second algorithm also solves the problem of discovering small frequent itemsets that uses a cubic-based index structure.

The second thesis proposes a new algorithm for subsequence discovery. The fundamental approach of the new algorithm is to use deterministic finite state machines for testing subsequence inclusion. Joining the several automatas a new structure is created, namely, the State Machine-Tree (SM-Tree for short) that can be used for determining the support of the candidate sequences at the same time by reading the items of the transactions exactly once.

My third thesis provides a solution for discovering frequent subtrees in a forest. In my thesis work I have shown how pushdown automatons can be used for testing subtree inclusion, and I have introduced a new structure called Pushdown Automaton-Tree (PD-Tree) that uses only one stack for determining the support of the several candidate trees at the same time by processing the nodes of the input tree only once.

From now on the three theses are summarized in a way that the main contribution of each thesis is emphasized with bold characters. The title of each thesis is immediately followed by my publications related to the specific thesis. The definitions and propositions necessary to understand the operation of the algorithms are also enlisted.

Thesis I
Frequent itemset mining
[1][2][3][10][13][15]

My first thesis deals with the problem of efficiently discovering frequent itemsets in transactional databases. I propose two methods, the ItemsetCode and Cubic algorithms for discovering small frequent itemsets in an efficient way. Thus, the process of mining larger itemsets is enhanced as well.

Thesis I.A

I have proposed a new algorithm called ItemsetCode algorithm for discovering frequent itemsets efficiently. The proposed method is a level-wise algorithm and its contribution is to reduce the problem of discovering the 3 and 4-frequent itemsets back to the problem of discovering 2-frequent itemsets by using a coding mechanism. The ItemsetCode algorithm discovers the 1 and 2-frequent itemsets in the quickest way by directly indexing a matrix. The 2-frequent itemsets are coded and the 3 and 4-candidates are created by pairing the codes. The counters for the 3 and 4-candidates are stored in a juggled array in order to have a storage structure of moderate memory requirements. The
way in which the candidates are created enables us to use the juggled array in a very efficient way by using two indirections only. Furthermore, the memory requirement of the structure is also low. The algorithm only partially exploits the benefits of the Apriori hypothesis. The reason is the compact storage structure for the candidates. The ItemsetCode algorithm discovers the large itemset efficiently because of the quick discovery of the small itemsets. Its level-wise approach ensures the fact that its memory requirement does not depend on the number of transactions.

- I defined a code set and I have proven that this code set is a total ordered set.

**Definition 3.1.** Let $\Xi = \xi_1, \xi_2, \ldots, \xi_k$ be the set of codes where $n$ denotes the number of frequent items, and the following conditions hold:

$$\begin{align*}
\xi_1 &= n + 1 \quad (3.1) \\
\xi_k &= n + k, \ (k = |L_2^{|I|}) \quad (3.2) \\
\xi_{i+1} - \xi_i &= 1, \ \forall i \ (0 < i < k) \quad (3.3)
\end{align*}$$

**Proposition 3.2.** The set of codes defined in Definition 3.1 is a total ordered set.

- I defined a coding and a decoding function between the set of frequent two itemsets ($L_2^{|I|}$) and the set of codes ($\Xi$). I have shown that using the coding mechanism each candidate appears exactly three times, and I provided two rules for generating all candidates exactly once.

**Definition 3.3.** Let $F_\xi : L_2^{|I|} \rightarrow \Xi$ be a bijective coding function between the 2-frequent itemsets ($L_2^{|I|}$) and the codes ($\Xi$). Because $L_2^{|I|}$ is lexicographically ordered, it is a total ordered set and the set of codes is also a total ordered set according to Proposition 3.2, the number of elements in both sets is identical, it is possible to define a bijective function according to the following rule: $\forall i (0 < i < k) \ F_\xi : L_2^{|I|} \rightarrow \xi_i$.

**Definition 3.4.** Let $F_D = F_\xi^{-1} : \Xi \rightarrow L_2^{|I|}$ be the decoding function. It always exists because $F_\xi$ is a bijective function. If $\xi \in \Xi$ is a code, and $x,y \in I$ are items, the following equations are true:

$$\begin{align*}
F_{D,1}(\xi) &= x \text{ iff } F_\xi(x,y) = \xi \quad (3.4) \\
F_{D,2}(\xi) &= y \text{ iff } F_\xi(x,y) = \xi \quad (3.5)
\end{align*}$$

**Proposition 3.5.** If all the code pairs $\xi_i \xi_j$ ($0 < i, j \leq k$) are generated as 3- and 4-candidates from the codes, the decoded form of the 3- and 4-candidates appears exactly three times.

**Rule 1.** If and only if $F_D(\xi_i) = F_D(\xi_j)$, then the code pair $\xi_i \xi_j$ is a 3-candidate.

**Rule 2.** If and only if $F_D(\xi_i) < F_D(\xi_j)$, then the code pair $\xi_i \xi_j$ is a 4-candidate.

**Proposition 3.6.** The candidates generated from the codes regarding Rule 1 and Rule 2 contain all the 3- and 4-candidates, and no redundant candidates are present. Furthermore the resulting candidate set partially exploits the Apriori hypothesis.
In order to store and handle the code pairs (thus the 3- and 4-candidates) efficiently I proposed a new storage structure for storing the counters. This structure is a juggled array. I suggested a method for creating the juggled array and I have proven that this structure holds the counters of all the 3- and 4-candidates.

**Definition 3.7.** Let $\Lambda$ be a list of objects. Let $\Lambda_p$ be the list belonging to the $p^{th}$ item of $\Lambda$, and the code indexing the $p^{th}$ item is $\xi_i$. ($\xi_i = p + n$). The items in $\Lambda_p$ (denoted by $\Lambda_p(i)$) are created according to Eqs. 3.6, 3.7 and 3.8.

\[
\Lambda_p[0] = \left\{ \begin{array}{ll}
\min_j \{\xi_j\} & \text{where } F_{D,1}(\xi_j) = F_{D,2}(\xi_i) \text{ if exists} \\
null & \text{otherwise}
\end{array} \right. \tag{3.6}
\]

\[
\Lambda_p[\text{max}] = \left\{ \begin{array}{ll}
\xi_k & \text{if } \Lambda_p[0] \neq \text{null} \\
null & \text{otherwise}
\end{array} \right. \tag{3.7}
\]

\[
\Lambda_p[q] = \left\{ \begin{array}{ll}
\Lambda_p[q-1] + 1 & \text{if } \Lambda_p[0] \neq \text{null} \\
null & \text{otherwise}
\end{array} \right. , \quad 0 < q < \text{max} \tag{3.8}
\]

**Proposition 3.8.** The list $\Lambda$ created according to Definition 3.7 contains all the 3- and 4-candidates.

Through several experimental results I have shown that the ItemsetCode algorithm is fast and uses less memory than the Apriori or the FP-growth algorithms.

**Thesis I.B**

The Cubic algorithm is another method for discovering the small frequent itemsets efficiently. It also uses a direct indexing method for discovering the 3 and 4-candidates. The algorithm uses the Apriori hypothesis only partially in order to have a structure which can be indexed continuously. However, it uses the Apriori hypothesis when different cubes are created for those itemsets whose first item is different. The greater itemsets can be discovered by calling the Apriori (in case of Cubic Apriori algorithm) or the FP-growth (in case of Cubic FP-Growth) algorithms. It is worth using the Apriori algorithm if the distribution of the dataset is such that the number of the four frequent itemsets are great. In the case of invoking the FP-growth algorithm the benefit is to have a smaller FP-tree, because the FP-tree is built only from those transactions which contain at least one frequent four itemset. Experimental results show the memory saving in this case.
My second thesis deals with the problem of discovering frequent sequences efficiently. The main idea of the new method called SM-Tree algorithm is to test the subsequence inclusion in such a way that the items of the input sequence are processed exactly once. The basis of the new approach is the deterministic finite state machines created for the candidates. By joining the several automata a new structure called SM-Tree is created such that handling a large number of candidates is faster than in the case of using different state machines for each candidate. By analyzing the new structure I have proposed a method for handling the SM-Tree efficiently. This can be done by exploiting the benefits of having two types of states, namely the fixed and the temporary states. The further benefit of the suggested algorithm is that its memory requirement is independent from the number of transactions which comes from the level-wise approach.

- I have proposed a new method for testing subsequence inclusion, namely, using a deterministic finite machine. I have showed how the candidate sequences can be represented as strings and how a machine can be built for this reason.

**Definition 3.9.** Let \( C = c_0, c_1, \ldots, c_s \) be the string representation of a candidate sequence of size \( k \), where \( s + 1 \) equals to the length of the string \( C \). The rules for generating a deterministic finite state machine for the sequence \( C \) are given in Table 3, where \( Q_i (Q_i \in Q, i = 0 \ldots s + 1) \) denotes the states of the machine, and \( \Sigma \setminus c_i \) denotes all characters in the alphabet \( \Sigma \) except \( c_i \), and \( Q_0 = q_0 \) and \( Q_{s+1} \in F \).

**Table 1: Transition functions of the finite state machine of the candidate sequence \( C = c_0, c_1, \ldots, c_s \).**

<table>
<thead>
<tr>
<th>Input item</th>
<th>Transition function</th>
</tr>
</thead>
<tbody>
<tr>
<td>( c_0 \in \Sigma \setminus {-} )</td>
<td>( \delta(Q_0, c_0) = Q_1 ) ( \delta(Q_0, \Sigma \setminus c_0) = Q_0 )</td>
</tr>
<tr>
<td>( c_i \in \Sigma \setminus {-} )</td>
<td>( \delta(Q_i, c_i) = Q_{i+1} ), ( i = 1 \ldots s ) ( \delta(Q_i, \Sigma \setminus {c_i, -}) = Q_i ) ( \delta(Q_i, {-}) = Q_p ) where ( Q_p = \max_{j &lt; i} {\delta(Q_{j-1}, {-}) = Q_j, q_0} )</td>
</tr>
<tr>
<td>( c_i = {-} )</td>
<td>( \delta(Q_i, c_i) = Q_{i+1} ), ( i = 1 \ldots s ) ( \delta(Q_i, \Sigma \setminus c_i) = Q_i )</td>
</tr>
</tbody>
</table>

**Proposition 3.10.** *The deterministic finite state machine created for the candidate sequence \( C \) based on Definition 3.9 accepts the input string \( \kappa \) if and only if \( \kappa \) contains \( C \).*
• In order efficiently to handle the several finite state machines created for the candidates, I have introduced a new structure called SM-Tree. Using the SM-Tree, the counters of all the candidates which are contained by the input sequence are incremented such that the items of the sequence are processed exactly once.

**Definition 3.11.** Let $M_{DFS_1} = (Q_1, \Sigma, \delta_1, q_{10}, F_1)$ and $M_{DFS_2} = (Q_2, \Sigma, \delta_2, q_{20}, F_2)$ be two finite state machines created for two candidate sequences based on Definition 3.9. The join operation on $M_{DFS_1}$ and $M_{DFS_2}$ results in a so-called State Machine-Tree (SM-Tree) which is defined as follows: $M_{SMTree_3} = M_{DFS_1} \Join M_{DFS_2} = (Q_3, \Sigma, \delta_3, q_{30}, F_3)$ where the notations are the following:

- a finite set of states $Q_3 = Q_1 \cup Q_2$,
- a finite set called the alphabet $\Sigma$, which is the same as the alphabet of the finite state machines to be joined,
- a transition function $\delta_3 : Q_3 \times \Sigma \rightarrow Q_3^2$,
- a start state $q_{30} = q_{10} = q_{20}$,
- and a set of accept states, $F_3 \subseteq Q_3, F_3 = F_1 \cup F_2$

**Proposition 3.12.** The SM-Tree created for the candidate sequences of the same size increments the counter of a candidate sequence if and only if the candidate is contained by the input sequence. Furthermore the SM-Tree increments the counters of those and only those candidate sequences which are contained by the input sequence. For this purpose the items of the input sequence has to be read exactly once.

• I have proposed a method for efficiently handling the SM-Tree by classifying its states into two classes. I have proven that using these two classes, only two arrays of fixed size have to be used for storing the tokens of the tree.

**Definition 3.13.** A state of an SM-Tree is called fixed state if it is the start state of the SM-Tree, or it is a state immediately after a $\lbrace - \rbrace$ transition.

**Definition 3.14.** A state of an SM-Tree is called temporary state if it is not a fixed state.

**Proposition 3.15.** The set of the fixed states can only be increased during the subsequence checking process. The set of the temporary states can be decreased as well, but in this case all the items are removed from it simultaneously.

**Collorary 3.16.** The lists for the fixed states and for the temporary states can be realized by using two arrays of fixed size. The upper bound of the two arrays is the number of the states of the SM-Tree.

• Using experimental results I have shown the time efficiency of the SM-Tree algorithm.
Thesis III
Frequent subtree mining
[5][18][22][23][25][27]

In my third thesis I propose a new method for determining whether a tree is contained by another tree. This can be done by using a pushdown automaton. In order to provide an input to the automaton, the tree is represented as a string. For handling the large number of candidates efficiently I have introduced the join operation between the automatons, and the resulting new structure is called PD-Tree. The new structure makes it possible to discover the support of each candidate at the same time by processing the items of a transaction exactly once. The benefit of the PD-Tree is that it uses only one stack to accomplish the mining process. Experimental results show the time saving when using the PD-Tree instead of using several pushdown automatons.

- I have shown how to represent a tree in order to have an appropriate input for a pushdown automaton.
- I have shown how to create a pushdown automaton for testing subtree inclusion. The rules for creating the automaton is depicted in Table 3 and the notations are explained in Table 2.

Table 2: Notations for the rules

<table>
<thead>
<tr>
<th>Notation</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>$\lambda$</td>
<td>The set of labels for labeling the trees</td>
</tr>
<tr>
<td>$\Sigma = \lambda \cup {-}$</td>
<td>The alphabet for the automation.</td>
</tr>
<tr>
<td>$\Gamma = \lambda \cup Z_0 \cup (\lambda, i)$</td>
<td>Stack symbols, where $(\lambda, i)$ denotes a structure containing a symbol and a number of a state</td>
</tr>
<tr>
<td>$\tau = {\tau_0, \tau_1, \ldots, \tau_{k-1}}$</td>
<td>The string encoding of the candidate tree for which the automaton is created, where $\tau_i$ is the $i^{th}$ character in $\tau$.</td>
</tr>
<tr>
<td>$Q = {q_{j_0}, q_{j_1}, \ldots, q_{j_k}}$</td>
<td>The states of the automaton where $j_i$ denotes the level of the node in the tree for which the given state was created</td>
</tr>
<tr>
<td>$*$</td>
<td>Any symbol on the top of the stack</td>
</tr>
</tbody>
</table>

Proposition 3.17. Let $\pi_1$ and $\pi_2$ denote two trees with their string encodings $\tau$ and $\kappa$ respectively. The pushdown automaton created for $\tau$ according to Table 3 accepts its input $\kappa$ if and only if $\pi_1$ is an embedded subtree of $\pi_2$.

- I have shown how to join several pushdown automatons in order to have one automaton called PD-Tree such that only one stack is needed for the whole PD-Tree.

Definition 3.18. Let two deterministic pushdown automatons be given: $M_{PDA_1}(Q_1, \Sigma_1, \Gamma_1, \delta_1, q_{0_1}, Z_{0_1}, F_1)$ and $M_{PDA_2}(Q_2, \Sigma_2, \Gamma_2, \delta_2, q_{0_2}, Z_{0_2}, F_2)$. The join
Table 3: Rules for creating a pushdown automaton for a candidate tree

<table>
<thead>
<tr>
<th>Input character</th>
<th>Transition function</th>
</tr>
</thead>
<tbody>
<tr>
<td>( \tau_0 \in \lambda )</td>
<td>((q_{00}, \tau_0, \ast) \rightarrow (q_{11}, (\tau_0, 0) \ast)) &lt;br&gt; ((q_{00}, \lambda \setminus \tau_0, \ast) \rightarrow (q_{00}, \lambda \setminus \tau_0) \ast) &lt;br&gt; ((q_{00}, -, \ast) \rightarrow (q_{00}, \varepsilon))</td>
</tr>
<tr>
<td>( \tau_i \in \lambda )</td>
<td>((q_{ij}, \tau_i, \ast) \rightarrow (q_{i+1j+1}, (\tau_i, \ast) \ast)) &lt;br&gt; ((q_{ij}, \lambda \setminus \tau_i, \ast) \rightarrow (q_{ij}, \lambda \setminus \tau_i) \ast) &lt;br&gt; ((q_{ij}, -, (\tau_p, p)) \rightarrow (q_{pj}, \varepsilon)) where (q_{pj-1} = \max_{k&lt;j} (q_{kj-1})) &lt;br&gt; ((q_{ij}, -, \lambda \setminus (\tau_p, p)) \rightarrow (q_{ij}, \varepsilon)) where (q_{pj-1} = \max_{k&lt;j} (q_{kj-1}))</td>
</tr>
<tr>
<td>( \tau_i = { - } )</td>
<td>((q_{ij}, -, \ast) \rightarrow (q_{i+1j+1}, \varepsilon)) &lt;br&gt; ((q_{ij}, \lambda \setminus -, \ast) \rightarrow (q_{ij}, \lambda \setminus -) \varepsilon))</td>
</tr>
</tbody>
</table>

Operation on \( M_{PDA_1} \) and \( M_{PDA_2} \) results in a Pushdown Automaton-Tree (PD-Tree) which is defined as follows: \( M_{PDTree_3} = M_{PDA_1} \pitchfork M_{PDA_2} = (Q_3, \Sigma_3, \Gamma_3, \delta_3, q_{03}, Z_{03}, F_3) \).

- \( Q_3 = Q_1 \cup Q_2 \)
- \( \Sigma_3 = \Sigma_1 = \Sigma_2 \)
- \( \Gamma_3 = \text{extended } \Gamma_1 \text{ as described in Definition 3.19} \)
- \( \delta_3(Q_3 \times (\Sigma_3 \cup \varepsilon) \times \Gamma_3 \rightarrow Q^2 \times \Gamma^k) \)
- \( q_{03} = q_{01} = q_{02} \)
- \( Z_{03} = Z_{01} = Z_{02} \)
- \( F_3 = F_1 \cup F_2 \)

Definition 3.19. Let \( \Gamma_3 = \Gamma_{PDTree} = Z_0 \cup \lambda \cup \langle \lambda, \{q_{i1}, q_{i2}, \ldots, q_{ip}\} \rangle \) denote the stack symbols of the Pushdown-Tree automaton where \( Z_0 \) is the initial stack symbol, \( \lambda \) denotes the labels of the tree and \( \langle \lambda, \{q_{i1}, q_{i2}, \ldots, q_{ip}\} \rangle \) is a structure where \( \{q_{i1}, q_{i2}, \ldots, q_{ip}\} \) (called state list) it the list of all the states from which \( \lambda \) causes a forward transition in the PD-Tree.

Proposition 3.20. The PD-Tree created for several candidate trees according to Definitions 3.18 and 3.19 increments the counters of a candidate tree if and only if the candidate is contained by the input. Furthermore the counters of all these candidates are incremented by processing the characters of the input string exactly once only.
4 Possibilities of Practical Applications

Discovering frequent itemsets is a basis of several real world problems. For example frequent patterns are used when discovering association rules, but they can be used by clustering and episode mining.

One of the application areas that use frequent sequence mining is the chemistry and medical sciences, where the sequences are used for DNA pattern analysis, or for discovering symptoms which follow each other frequently. Sequences can be used when investigating network events, and their temporal behavior.

Graph mining is also used in chemistry, then the structure of the chemical compounds can be investigated in detail. It is used also for Web log and structure analysis.

The case study introduced in my thesis work is a system that can exploit the benefit of all the three problems mentioned in the theses. The case study is the investigation of a Web usage log. The log can be used for discovering those pages that are visited frequently together within a period of time. This problem can be solved using frequent itemset mining. When not only the pages are relevant, but also the temporal behavior of the users, namely, in which order were the pages visited, then the frequent sequences should be discovered. The forward navigation of the users can be interpreted as tree structures of the Web portal, and in this case the frequent tree mining can be used for obtaining useful information about navigational behavior of the user.

5 Scientific Publications

**International Journals**


2 **Iváncsy, R.** and I. Vajk, “A Time and Memory Efficient Frequent Itemset Discovering Algorithm for Association Rule Mining”, International Journal of Computer Applications in Technology, Special Issue on "Data Mining Applications" by Inderscience Enterprises Ltd. (in press)

3 **Iváncsy, R.** and I. Vajk, “Fast Discovery of Frequent Itemsets: a Cubic-Structure-based Approach”, Computational Intelligence in Data mining Special Issue of the Informatica Journal (ISSN 0350-5596) (accepted)


Hungarian Journals


Edited Books


Conference Proceedings


