Proceedings of the 5th WSEAS Int. Conf. on SIMULATION, MODELING AND OPTIMIZATION, Corfu, Greece, August 17-19, 2005 (pp190-195)

Efficient Subtree Inclusion Testing in Subtree Discovering Applications

RENATA IVANCSY, ISTVAN VAJK
Department of Automation and Applied Informatics and HAS-BUTE Control Research Group
Budapest University of Technology and Economics
1111 Goldmann Gy. ter 3., Budapest
HUNGARY

Abstract: - Frequent pattern mining as part of the data mining process can be used in many applications. The type of the patterns can be various regarding the problem to be solved. In several cases the problem can be modeled with graphs only, thus methods are needed which can discover such patterns from large databases. Trees are special graphs where one path exists only between two arbitrary nodes. Trees can be handled easier than a general graph; however trees can model several problems, thus discovering frequent trees has justification. In this paper a novel approach is presented for efficiently discovering frequent subtrees from a tree database. The main contribution of the new method is to use pushdown automaton for subtree inclusion testing. In order to enhance the performance of counting the support of the candidate trees, the several automatons are joined such that only one stack is needed for handling the joined automatons.

Key-Words: - Data mining, Frequent pattern mining, Graph mining, Subtree discovery, Pushdown Automaton

1 Introduction

Labeled graph is an appropriate tool for modeling several real world data like Web links, chemical compounds, academically citations etc. For this reason the studies of pattern mining have been extended to process not only transactional data, like in frequent itemset mining, but also semi-structured and structured data.

When extending the patterns to be searched to graphs, the patterns and the rules that can be created based on the patterns get more complex. Complex rules can describe the problem more precisely; however the process of discovering such rules is complex as well. Thus beside exploiting the benefits using structured data one have to be care of keeping the complexity of the problem as low as possible.

Trees can be considered as the medium between graphs and transactional data. Handling trees is easier than handling general graphs, but one can have more complex rules when using trees than using flat data. Furthermore several real world problems can be modeled with tree data as well.

This paper deals with the problem of discovering frequent subtrees in a forest. The trees are rooted, labeled, ordered and the algorithm presented in this paper searches for embedded subtrees. A new method is proposed which uses pushdown automaton in order to enhance the subtree inclusion testing process.

The organization of the paper is as follows. Section 2 describes briefly the problem of frequent subtree discovery. Section 3 presents the most important algorithms. In Section 4 the new approach is presented, namely how to create pushdown automaton for subtree inclusion detecting. In Section 5 the way is shown how to join the several pushdown automatons to create an automaton for detecting the subtree inclusions for all the candidate trees at the same time. Experimental results are shown in Section 6, and conclusions are in Section 7.

2 Problem Statement

This section introduces the most important notations and definitions regarding frequent subtree discovery.

A tree can be denoted with its set of vertices (V) (also called nodes) and the set of edges (E) between the nodes. A rooted, labeled tree is a 5-tuple $\pi(V, E, \lambda, f_\lambda, v_0)$ where (i) V is the set of vertices; (ii) E denotes the set of edges in a tree; (iii) λ is the set of labels for any node $v \in V$, (iv) f_λ is a function which maps for each node a label ($\forall v \in V, f_\lambda(v) \in \lambda$); (v) $v_0 \in V$ is a dedicated node in the tree called the root. A tree is ordered if it is a rooted tree and the children of a given node have an ordering. The size of a tree equals to the number of vertices in the tree.

A tree $\pi_1(V_{\pi 1}, E_{\pi 1})$ is an embedded subtree of $\pi_2(V_{\pi 2}, E_{\pi 2})$ if $V_{\pi 1} \subseteq V_{\pi 2}$ and a branch appears in π_1 if and only if the two vertices are on the same path from the root to a leaf in π_2. Given a database D containing trees, the support of a tree π is the number of the trees in D which has π as an embedded subtree. In this case the number of occurrences of π in a given tree is irrelevant. Given

a user specified minimum support threshold (*minsup*) a tree is called frequent if it is contained by more trees in the database than the threshold, in other words the support of the tree exceeds the minimum support threshold.

3 Related Work

The TreeMiner algorithm proposed in [1] is one of the first contributed tree miner algorithms. It discovers the frequent embedded subtrees in a forest, where a forest is a set of rooted, labeled, ordered trees. The main contribution of the algorithm is to define equivalent classes and to use so-called scope-lists for each frequent tree in order to keep track which trees contain a given frequent subtree. The algorithm uses a DFS traversal for generating the frequent trees, and using the scope-list, the support of the subtrees can be calculated only by comparing the scope-lists of two subtrees from which the new subtree is created.

Another algorithm called FREQT [2] discovers also the frequent tree patterns from a set of labeled, ordered trees. The algorithm uses a rightmost expansion technique, which means that a tree is created from a frequent tree by attaching new nodes only to the rightmost branch of the tree. FREQT uses a BFS approach, and during the candidate generation it also generates the rightmost occurrence list for the candidates.

The algorithms mentioned so far use a "candidate generate and test" approach based on the Apriori hypothesis to discover frequent rooted, labeled trees. The Chopper [3] and XSpanner [3] algorithms use a new approach which is based on sequence mining. In Chopper the sequence mining step and the extraction of frequent tree patterns from the sequences are separated, while in XSpanner these two steps are integrated.

The FreeTreeMiner [4] is a method for discovering free trees, which is a more complex problem than discovering rooted trees. Thus in case of free trees, the canonical center of the tree has to be found which is used further as the root. Afterwards the nodes in the tree should be ordered, such that the result is a rooted, ordered tree, that is, the canonical form. The algorithm can be implemented either as an Apriori-like level-wise search or as a depth-first search. It searches for a range from small trees to large trees according to the "issubtree-of" relation. Further tree mining algorithms are presented in [5, 6] and a more detailed overview of the different tree mining algorithms can be found in [7].

4 Pushdown Automaton Approach

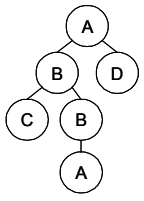
One of the most time consuming and computationally complex tasks is the subtree inclusion testing step. In case of algorithms which project the database into the memory like the TreeMiner algorithm, it can be done by using the scope lists of the candidate trees. However, these algorithms have the drawback that the memory requirement depends strongly on the number of transactions, which can be critical. Thus a level-wise method is needed which can test the subtree inclusion in an efficient way. For this reason this section introduces a novel method which exploits the benefits of using a pushdown automaton for detecting subtrees in an input tree.

A pushdown automaton (PDA) is a finite automaton outfitted with access to a potentially unlimited amount of memory called the stack. The pushdown automaton can be defined with a 7-tuple as follows: $P\ (Q, \Sigma, \Gamma, \delta, q_0, Z_0, F)$, (i) the final set of states, Q, (ii) the alphabet of the input, Σ, (iii) the alphabet of the stack, Γ, (iv) the set of transition functions, $\delta\ (Q \times (\Sigma \cup \varepsilon) \times \Gamma \rightarrow Q \times \Gamma^k)$, (v) the start state, $q_0 \in Q$, (vi) the initial stack symbol, $Z_0 \in \Gamma$ and (vii) the set of accept states, $F \subset Q$.

The automaton starts in its start state and the stack contains only the initial symbol. When reading one character from the input string, the set of transition functions are checked whether one of them can be used. If there is a transition rule which contains the state in which the automation resides, and the same symbol is on the top of the stack as in the rule, and the same character is just read, the automaton moves into its new state and a new symbol is pushed into the stack. By default the symbol from the stack is removed, when it is used by the transition function. If one wants to keep the symbol in the stack, one has to push it again. The ε symbol is used when no symbol is pushed into the stack. The input string is accepted if the automaton is in an accept state after its last character has been processed.

The new idea is to use a pushdown automaton for detecting whether a tree is contained by another tree. Using this approach the support counting of the candidates can be achieved by processing the input tree only once. At the end of the transaction the counters of those candidates should be incremented whose automaton is in an accept state. For this reason the trees are represented with strings as follows. The string encoding τ is initialized to an empty sting, $\tau = \varnothing$. Afterwards the tree is traversed in preorder manner starting at the root, and the label λ_i of the current node is added to the end of τ. Whenever the algorithm has to backtrack from a

child to its parent a $\{-\}$ sign is added to τ. In this case it is assumed that the $\{-\}$ sign is not in the label set of the tree. After the last label was reached the algorithm terminates, which means that the algorithm does not traverse back to the root as described in [1], thus the minus signs from the end of the string are omitted. As an example Fig. 1 shows a tree with its string encoding.



τ = ABC–BA– – – D

Fig.1. A sample tree with its string encoding

The rules generating the pushdown automaton for detecting a subtree in an input tree is described in Table 2. The notations for the rules can be found in Table 1.

Table 1. Notations for the rules

Notation	Meaning
λ	The set of labels for labeling the trees
$\Sigma = \lambda \cup \{-\}$	The alphabet for the automaton.
$\Gamma = \lambda \cup Z_0 \cup \langle \lambda, i \rangle$	Stack symbols, where $\langle \lambda, i \rangle$ denotes a structure containing a symbol and a number of a state
$\tau = \{\tau_0, \tau_1, \ldots, \tau_{k-1}\}$	The string encoding of the candidate tree for which the automaton is created, where τ_i is the i^{th} character in τ.
$Q = \{q_{0j_0}, q_{1j_1}, \ldots, q_{kj_k}\}$	The states of the automaton where j_i denotes the level of the node in the tree for which the given state was created
$*$	Any symbol on the top of the stack

Table 2. Rules for creating a pushdown automaton for a candidate tree

Input character	Transition function
$\tau_0 \in \lambda$	$(q_{00}, \tau_0, *) \rightarrow (q_{11}, \langle \tau_0, 0 \rangle *)$ $(q_{00}, \{\lambda \setminus \tau_0\}, *) \rightarrow (q_{00}, \{\lambda \setminus \tau_0\} *)$ $(q_{00}, -, *) \rightarrow (q_{00}, \varepsilon)$
$\tau_i \in \lambda$	$(q_{ij}, \tau_i, *) \rightarrow (q_{i+1j+1}, \langle \tau_i, i \rangle *)$ $(q_{ij}, \{\lambda \setminus \tau_i\}, *) \rightarrow (q_{ij}, \{\lambda \setminus \tau_i\} *)$ $(q_{ij}, -, \langle \tau_p, p \rangle) \rightarrow (q_{pj-1}, \varepsilon)$ where $q_{pj-1} = max_{k<j}(q_{kj-1})$ $(q_{ij}, -, \lambda \setminus \langle \tau_p, p \rangle) \rightarrow (q_{ij}, \varepsilon)$ where $q_{pj-1} = max_{k<j}(q_{kj-1})$
$\tau_i = \{-\}$	$(q_{ij}, -, *) \rightarrow (q_{i+1j+1}, \varepsilon)$ $(q_{ij}, \{\lambda \setminus -\}, *) \rightarrow (q_{ij}, \{\lambda \setminus -\} \varepsilon)$

As described in Table 2 each state can have two transitions which results in a different state. One of them is a forward transition and the other is a backward transition. The forward transition is used when the input tree seems to contain the candidate. The backward transitions are for those cases when the input tree does not contain the candidate yet.

Proposition 1. *Let π_1 and π_2 denote two trees with their string encodings τ and κ respectively. The PDA created for τ according to Table 2 accepts its input κ if and only if π_1 is an embedded subtree of π_2.*

Proof. The proof is given constructively by describing the process of the algorithm. The pushdown automaton for detecting a tree in the input tree works as follows. The automaton starts in its start state q_{00}. It reads the characters of the input string $\kappa = \kappa_0\ \kappa_1 \ldots\ \kappa_s$ one by one. If $\cdot\kappa_i = \tau_0$, the automaton gets into its next state (q_{11}) and the symbol and the number of the start state as a structure ($<\kappa_i,0>$) are pushed into the stack. In other cases the automaton remains in its start state, and if $\cdot\kappa_j \in \lambda$, then the character is pushed, otherwise the topmost symbol is popped. When the automaton is in an arbitrary state q_{ij} , four possibilities exist. If the character just read (κ_l) equals to the character expected by the given state, then the automaton moves in its next state, and the character and the state number are pushed if $\cdot\kappa_l \in \lambda$, otherwise the topmost symbol is popped. In other cases when the input character κ_l was not expected, but it is in λ, only the character is pushed into the stack and the automaton stays in its state. If the input character is a minus sign and it is not expected, two possibilities exist. The backward transition is used if the topmost structure matches the structure assigned to the backward transition. In other cases the self loop is used. In both cases the topmost symbol is popped from the stack.

In order to better understand the process of the pushdown automaton Fig. 2 shows a sample *PDA* for the tree $\tau = ABC–BA– – –D$. The notations on the arrows are in the following form: $\Sigma,\Gamma/\ \Gamma^k$, the first part of the expression (before the / sign) shows which character has just been read and which symbol is on the top of the stack. The second part denotes the symbols that should be pushed into the stack. The * denotes any symbol from the stack.

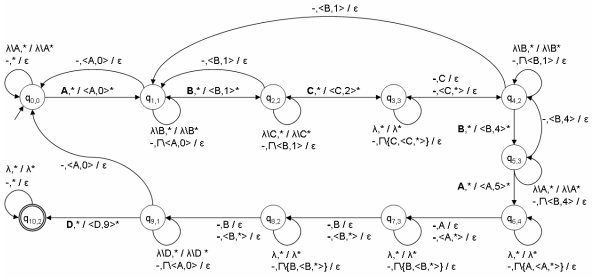


Fig.2. Sample pushdown automaton for the candidate tree ABC – BA – – D

5 PD-Tree Approach

Because of the possible large number of candidate trees on each level, it is worth joining the several automatons of the candidates in order to reduce computational cost. However it is not trivial how to join two pushdown automatons. It is expected that the resulting object needs less memory and its operation should be more efficient than that of the

separated automatons. One of the most important consideration is the number of the stack which have to be used. If this cannot be decreased the benefits of joining the PDA-s is questionable. In the sequel it is shown how the pushdown automatons can be joined such that the execution time is reduced. Furthermore the number of the stacks to be used remains one. Let two pushdown automatons be given $P_1(Q_1, \Sigma_1, \Gamma_1, \delta_1, q_{01}, Z_{01}, F_1)$ and $P_2(Q_2, \Sigma_2, \Gamma_2, \delta_2, q_{02}, Z_{02}, F_2)$. The *join* operation on P_1 and P_2 results in a so-called *Pushdown Automaton-Tree* (PD-Tree) which is defined as follows: $P_3(Q_3, \Sigma_3, \Gamma_3, \delta_3, q_{03}, Z_{03}, F_3)$ where (i) $Q_3 = Q_1 \cup Q_2$, (ii) $\Sigma_3 = \Sigma_1 = \Sigma_2$, (iii) Γ_3 = extended Γ_1, as described later (iv) $\delta_3 (Q_3 \times (\Sigma_3 \cup \varepsilon) \times \Gamma_3 \rightarrow Q_3^2 \times \Gamma_3^k)$, (v) $q_{03} = q_{01} = q_{02}$ (vi) $Z_{03} = Z_{01} = Z_{02}$ and (vii) $F_3 = F_1 \cup F_2$.

The main problem one faces when joining two pushdown automatons originates from the fact that during the process not only one accept state exists but several, and using the tree all accept states have to be accessed, i.e. all counters for those trees have to be incremented which are contained by the input tree. The other problematic fact is that because of space saving only one stack has to be used, thus the characters pushed into the stack are mixed up regarding the different candidate trees. Furthermore because not only one active state exists at the same time, but several, also when pushing a character into the stack not only one state has to be inserted into the structure but all from which a new state is reached.

For this reason the definition of the stack symbols has to be modified as follows. Let $\Gamma_3 = Z_0 \cup \lambda \cup < \lambda,q_{i1},q_{i2},\ldots,q_{ip}>$ denote the stack symbols of the PD-Tree, where $< \lambda,q_{i1},q_{i2},\ldots,q_{ip}>$ is a structure where $\{q_{i1},q_{i2},\ldots,q_{ip}\}$ (called *state list*) is the list of all the states from which λ causes a forward transition in the PD-Tree.

Proposition 2. *The PD-Tree created for several candidate trees 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.*

Proof. The modified definition of the stack symbols means that for each label those states are stored in the stack from which a transition were proceeded. This is necessary because of the following. A state in the PDTree can have one forward transition and one backward transition. The forward transition is used independently of the content of the stack. The backward transition is followed only when the stack contains the same label as the label in the transition rule is. However this is not the only condition, because the backward transition has to be followed only, if using a backtracking in the tree such a node is reached which causes that the candidate is not possible to be contained by the input. In this case the automaton gets in its previous state. Thus we have to know which label has caused the forward step in order to know which has to be caused the backward as well. This is marked with the state in the simple PDA in case of single subtree inclusion testing, and with a state list in case of the joined PDAs.

Thus when processing the PD-Tree, when the automaton reads a {-} character, for each state, the possible state for a backward transition has to be calculated, and it has to be checked whether it is contained in the topmost state list of the stack. If it is contained, the automaton must step back, otherwise it remains in the current state or it also steps forward as well.

As an example Fig. 3 shows a PD-Tree when joining the following trees: *ABC – B*, *ABC – –B*, *BA – AB* and *ABCD*. The figure does not contain the self-loops in order to have a clearer view of the relevant part of the automaton.

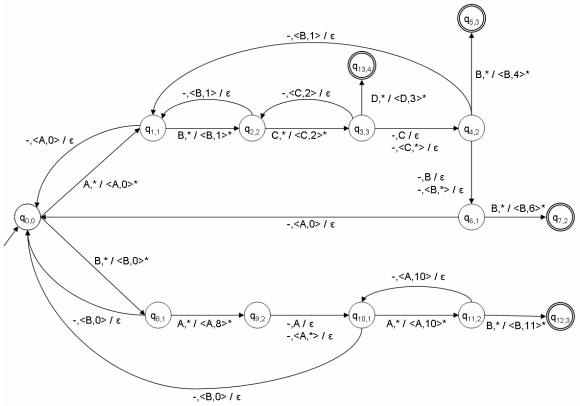


Fig.3 Sample PD-Tree

6 Experimental Results

In order to show the efficiency of the PD-Tree algorithm several experimental results were created. The simulations were executed on a Pentium 4 CPU, 2.4GHz, and 1GB of RAM computer. The algorithms were implemented in C#.

The two algorithms are the PD-Tree and the *Stack Automatons* where the latter is an implementation of the pushdown automatons for each candidate without joining them. The algorithm works as follows. A separate pushdown automaton is created for each candidate. When a transaction is processed, all the automatons are checked for each item of the transaction, whether it can take a transition into a new state.

Two types of datasets were used in the experiments. One of them contains the candidates that are to be checked whether they are contained by the transaction. The transaction database belongs to the second type of dataset. The meaning of the notations is as follows: *D* stands for the maximum depth, *F* means the number of fan-out, *L* is the number of labels, *T* stands for the number of transactions and *K* means thousands.

Fig. 4 shows the execution time of the two algorithms when using the candidate dataset D4F3L10T10K, and D5F3L10Tx as the transaction database, where *x* is in the range of 100 and 10000. It can be seen well that the *PD-Tree* algorithm is an order of magnitude faster than the *Stack Automatons* algorithm in the whole range. The reason can be observed in Fig. 5 where the number of those states are depicted which were active during the mining process. It is observable that the number of active states is more orders of magnitude greater in case of the *Stack Automatons* algorithm. This is the expected result, as the joining step and its consequences were declared as the main benefit of the *PD-Tree* algorithm, because one of the consequences were to reduce the number of the active states.

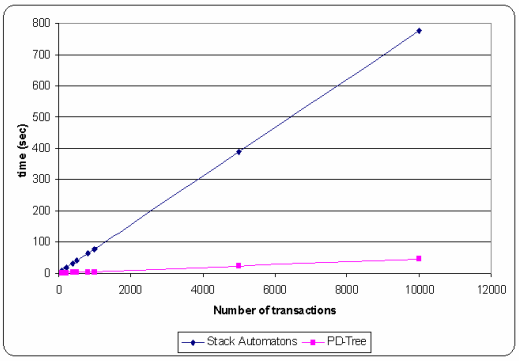


Fig.4. Execution time of the two algorithms

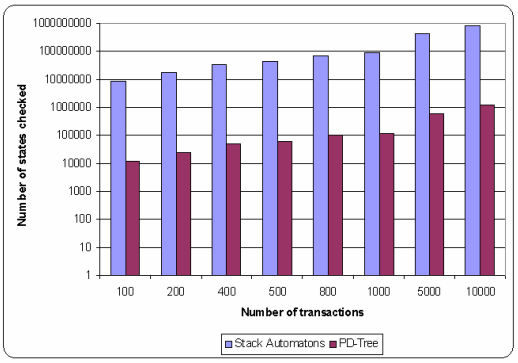


Fig.5. Number of active states during the process

Further measurement results are shown in Table 3, where both the name of the candidate datasets and the name of the transaction datasets are depicted. The execution times of the two algorithms are enumerated as well. The last column stores the speed up ratio when using the PD-Tree algorithm. It can be observed that the speed up ratio is greater, when the number of the candidates is high, while it is small, when the number of the candidates is limited. This can also be explained with the joining step.

Table 3. Execution times of the two algorithms

	Candidate database	Tree database	Stack Auto. (sec)	PD-Tree (sec)	Speed-up Ratio
1	D4F3L10T5K	D5F4L10T1K	160.72	6.50	24.73
2	D4F3L10T5K	D5F4L10T10K	1630.44	74.31	21.94
3	D4F3L10T50K	D5F4L10T1K	160.17	5.02	31.93
4	D5F3L10T1K	D4F3L10T10K	43.36	13.23	3.27
5	D5F3L10T1K	D4F3L10T5K	43.42	18.83	2.31
6	D5F3L10T1K	D4F3L10T50K	43.26	12.38	3.50
7	D5F3L10T10K	D4F3L10T5K	613.44	98.42	6.23
8	D5F4L10T1K	D5F4L10T10K	255.52	40.59	6.29
9	D5F3L10T10K	D5F4L10T1K	354.48	37.44	9.47

In Fig. 6 the execution times depicted in Table 3 are shown with cube diagrams. Fig. 7 shows the number of the states which were checked during the mining process. The numbering on the *x* axis of Fig. 6 corresponds to the numbering of the different cases in Table 3.

One can draw the conclusion that joining the automatons results in a significant time saving. However the number of the states saved in this way is less significant as shown in Fig. 8, where the state numbers created by the two algorithms are depicted for the different candidate datasets.

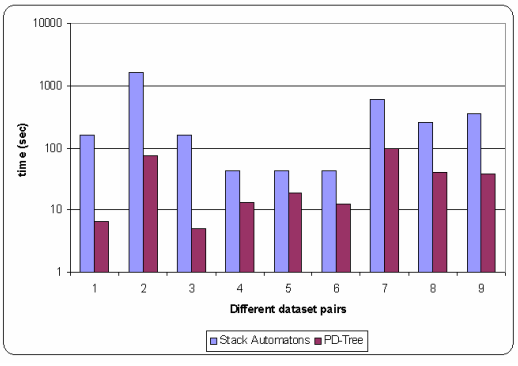


Fig.6. Execution time in the cases shown in Table 3

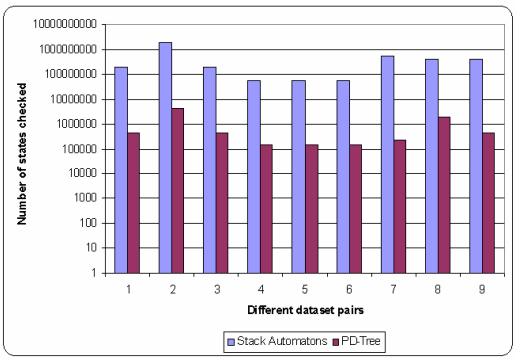


Fig.7. Number of the states checked during the mining process in the cases shown in Table 3

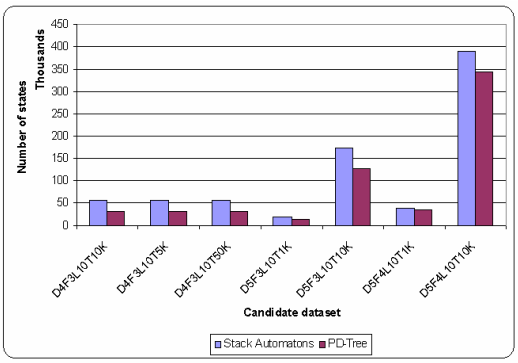


Fig.8. Number of states created by the two algorithms for the different candidate datasests shown in Table 3

7 Conclusion

This paper deals with the problem of efficient subtree discovery in tree databases. In this paper a novel approach is presented which uses pushdown automatons for subtree inclusion testing. After introducing the basic concepts and the related work, it was shown how to create pushdown automatons for the candidates, and how to join them to create a so-called PD-Tree structure, such only one stack is needed for the whole process. The active states of the new automatons can be handled using tokens. Experimental measures show the efficiency of the PD-Tree structure over the several automatons.

Acknowledgments

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] M. Zaki, Efficiently mining frequent trees in a forest, *In Proc. of the 8th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, Edmonton, Alberta, Canada, 2002, pp. 71-80.

[2] Tatsuya Asai, Kenji Abe, Shinji Kawasoe, Hiroki Arimura, Hiroshi Satamoto, and Setsuo Arikawa. Efficient substructure discovery from large semistructured data. In Robert L. Grossman, Jiawei Han, Vipin Kumar, Heikki Mannila, and Rajeev Motwani, editors, *SDM*. SIAM, 2002.

[3] C. Wang, M. Hong, J. pei et al, Efficient Pattern-Growth Methods for Frequent Tree Pattern Mining, *In. Proc of the 8th Pacific-Asia Conference of Advances in Knowledge Discovery and Data Mining, PAKDD 2004,* Sydney, Australia, May 26-28, 2004. pp. 441-451.

[4] U. Ruckert, S. Kramer, Frequent free tree discovery in graph data, *In Proc. of the 2004 ACM Symposium on Applied Computing (ACM, 2004)*, Nycosia, Ciprus, 2004.

[5] A. Termier, M.C. Rousset and M. Sebag, TreeFinder: a first step towards XML Data Mining, *In Proc. Of 2002 IEEE International Conference on Data Mining (ICDM'02)*, Maebashi city, Japan, 2002, pp. 450-457.

[6] Dimitrios Katsaros, Alexandros Nanopoulos, and Yannis Manolopoulos. Fast mining of frequent tree structures by hashing and indexing. *Information & Software Technology*, 47(2):129–140, 2005.

[7] Yun Chi, Richard R. Muntz, Siegfried Nijssen, and Joost N. Kok. Frequent subtree mining – an overview. *Fundamenta Informaticae, Special issue on Advances in Mining Graphs, Trees and Sequences*, 66(12):161–198, March-Apr 2005.