World Academy of Science, Engineering and Technology
International Journal of Computer and Information Engineering
Vol:1, No:8, 2007

Automata Theory Approach for Solving Frequent Pattern Discovery Problems

Renáta Iváncsy, and István Vajk

***Abstract*—**The various types of frequent pattern discovery problem, namely, the frequent itemset, sequence and graph mining problems are solved in different ways which are, however, in certain aspects similar. The main approach of discovering such patterns can be classified into two main classes, namely, in the class of the level-wise methods and in that of the database projection-based methods. The level-wise algorithms use in general clever indexing structures for discovering the patterns. In this paper a new approach is proposed for discovering frequent sequences and tree-like patterns efficiently that is based on the level-wise issue. Because the level-wise algorithms spend a lot of time for the subpattern testing problem, the new approach introduces the idea of using automaton theory to solve this problem.

***Keywords*—**Frequent pattern discovery, graph mining, pushdown automaton, sequence mining, state machine, tree mining.

I. Introduction

THE problem of discovering frequent patterns can be divided in many classes regarding the type of the pattern searched for. Two types of patterns are in the focus of this paper, namely, sequences and trees that are often the target of recent data mining researches.

Frequent sequence mining is used in several real world problems like DNA sequence mining, WEB log mining, customer sequence mining and many more. Also frequent trees are useful in many applications where the data cannot be modeled with simple transactions like itemsets or sequences, but more complex structures are needed such as trees (HTML and XML structures, user navigation patterns, chemical compounds etc.).

The algorithms that solve the frequent pattern mining problem are similar in the two mentioned field. Both in sequence discovery and in tree pattern discovery the two main approaches are the level-wise and the database projection-based approaches. Both of them have their advantages and disadvantages regarding the computational cost and memory requirements. In general the level-wise methods are slower but need less memory, while the memory requirements of the database projection-based methods is high, and their execution time is moderated.

Because in most cases the memory requirements of the applications have to be limited, the new algorithms suggested in this paper for solving both types of pattern mining problems are based on the level-wise approach. The new idea of the novel algorithms is to use automaton theory for discovering the support of the candidate patterns efficiently.

The organization of the paper is as follows. Section II introduces the problem of frequent sequence and tree mining. Section III presents the backgrounds and motivations to the new approach that uses automaton theory for the mining process. In Sections IV-A and IV-B the details of creating the automatons for sequence and tree mining are explained. Experimental results are shown in Section V. Conclusion can be found in Section VI.

Manuscript received August 25, 2005. This work was supported by the fund of the Hungarian Academy of Sciences for control research and the Hungarian National Research Fund (grant number: T042741).

R. Iváncsy is with the Department of Automation and Applied Informatics and HAS-BUTE Control Research Group, Budapest University of Technology and Economics, Goldmann Gy. ter 3, Budapest, Hungary, H-1111 (Corresponding author to provide phone: +36(1)4631668, e-mail: renata.ivancsy@aut.bme.hu).

I. Vajk is with the Department of Automation and Applied Informatics and HAS-BUTE Control Research Group, Budapest University of Technology and Economics, Goldmann Gy. ter 3, Budapest, Hungary, H-1111 (e-mail: vajk@aut.bme.hu).

II. Problem Definition

The problem of sequential pattern mining was first introduced by Agrawal and Srikant in [1]. A *sequence* is denoted by $s=<s_1, s_2, ...,s_n>$, where s_i is an itemset. An item can occur in an itemset only once, but multiple times in a sequence. A sequence $<a_1, a_2, ...,a_n>$ is *contained* by another sequence $<b_1, b_2, ..., b_m>$ if there exist integers $i_1< i_2< ...< i_n$ such that $a_1 \subseteq b_{i1}, a_2 \subseteq b_{i2}, ..., a_n \subseteq b_{in}$.. The *length* of a sequence is defined as the number of items in the whole sequence. If the size of the sequence is k, then it is called k-sequence.

The *support* of a sequence s (denoted $\sigma(s)$) is the number of sequence transactions which contain the sequence s. A sequence is called *frequent* if it is contained by more transactions than a user-given minimum support threshold, (σ_{min}). The task of sequential pattern mining is to discover the frequent sequences in the database when given a user-defined minimum support threshold.

In several applications the problem to be handle cannot be modeled with such simple transactions like sequences, but more complex structures are needed like graphs or trees.

A *tree* is an acyclic connected graph. A *rooted*, *labeled tree* is a 5-tuple $T(V, E, \lambda, f_\lambda, v_0)$ where (1) V is the set of nodes; (2) E denotes the set of edges in a tree; (3) λ is the set of labels for any node $u \in V$, (4) f_λ is a function which maps for each

node a label ($\forall u \in V$, $f_\lambda(u) \in \lambda$); (5) $v_0 \in V$ is a dedicated node in the tree called the root.

A tree is *ordered* if it is a rooted tree, and the children of any node in the tree construct an ordered set. The *size* of a tree equals to the number of vertices in the tree. A tree $S(V_s, E_s)$ is an *embedded subtree* of $T(V_t, E_t)$ if $V_s \subseteq V_t$ and a branch appears in S if and only if the two vertices are on the same path from the root to a leaf in T.

Given a database *D* containing trees, the *support* of a tree *T* ($\sigma(T)$) is the number of the trees in *D* which has *T* as an embedded subtree. In this case the number of the occurrences of *T* in a given tree is irrelevant. Given a user specified minimum support threshold (σ_{min}) a tree is called *frequent* if it is contained by more trees in the database than the threshold.

III. Automaton Theory

The two novel methods presented for solving the frequent sequence and frequent tree mining problems are both level-wise, and uses a "candidate generate and test" approach. The main contribution of the novel methods presented in this paper is how they determine the support of the candidates, i.e. how the subpattern inclusion test is achieved.

The outline of the level-wise approach is as follows. The algorithm discovers the *1*-frequent pattern during one database scan. The *2*-candidates are created from the *1*-frequent patterns, and the support of the *2*-candidates is determined during a further database scan. In general, the *k*-candidates are generated from the (*k-1*)-frequent patterns, and during a database scan the support of the candidates is determined (using subpattern inclusion test), and the infrequent patterns are discarded.

Both proposed algorithms use basically different approach for the subpattern inclusion test than the algorithms in the related work. The most important sequential pattern mining algorithms are the AprioriAll [1], GSP [2], SPIRIT [3], SPADE [4], FreeSpan [5], PrefixSapn [6] and SPAM [7] algorithms. The best-known tree miner algorithms are TreeMiner [8], FREQT [9] and FreeTreeMiner [10]. The contribution of the novel approach proposed in this paper is to use automaton theory for testing the subpattern inclusion.

A. Subsequence Inclusion Test

For testing subsequence inclusion deterministic finite state machines can be used.

Definition 1: A *deterministic finite state machine* is a 5-tuple, $(Q, \Sigma, \delta, q_0, F)$ consisting of:

- a finite set of states (Q),
- a finite set called the alphabet (Σ),
- a transition function ($\delta: Q \times \Sigma \rightarrow Q$),
- a start state ($q_0 \in Q$),
- and a set of accept states ($F \subseteq Q$).

The state machine accepts the input string if the string contains the candidate sequence. For this reason the sequences are represented with strings from the alphabet $\Sigma = \Sigma' \cup \{-\}$, where $\{-\}$ is the character that separates the itemsets in the sequence, and Σ' is the set of items which can appear in the sequences. For example the string representation of the sequence <(ab)(c)(de)> is ab–c–de.

Definition 2: 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 I, where Q_i ($Q_i \in Q$, i=0…s+1) denotes the states of the machine, and $\Sigma \backslash c_i$ denotes all characters in the alphabet Σ except c_i and the following conditions hold: $Q_0 = q_0$ and $Q_{s+1} \in F$.

TABLE I
Transition Functions of the Finite State Machine of the Candidate Sequence $C=C_0,C_1,\ldots,C_S$

Input item	Transition function
$c_0 \in \Sigma \setminus \{-\}$	$\delta(Q_0, c_0) = Q_1$ $\delta(Q_0, \Sigma \setminus c_0) = Q_0$
$c_i \in \Sigma \setminus \{-\}$	$\delta(Q_i, c_i) = Q_{i+1},\ i = 1 \ldots s$ $\delta(Q_i, \Sigma \setminus \{c_i, -\}) = Q_i$ $\delta(Q_i, \{-\}) = Q_p$ where $Q_p = \max_{j<i}\{\delta(Q_{j-1}, \{-\}) = Q_j, q_0\}$
$c_i = \{-\}$	$\delta(Q_i, c_i) = Q_{i+1},\ i = 1 \ldots s$ $\delta(Q_i, \Sigma \setminus c_i) = Q_i$

The machine starts in the start state Q_0. The further conditions can be interpreted as follows. For each new character a new state is created and the transition between the states contains the character. These are represented in the state diagram of the finite state machines as forward edges. There are several backward edges as well. A backward edge is created between the state having no transition with a minus sign and the state immediately after the last minus sign so far. From each state there exist transitions to all the items such that the state will be the same (self loops). The accept state of the machine is the state for the last item of the sequence. As an example Fig. 1 shows the state diagram of the finite state machine for the sequence <(ab)(c)(de).

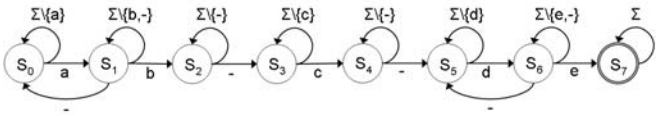


Fig. 1 State diagram for the sequence <(ab)(c)(de)

Proposition 1: The deterministic finite state machine created for the candidate sequence *C* accepts the input string κ if and only if κ contains *C*

Proof: In the one hand if κ contains *C*, then the automaton gets into its accept state because of the forward edges, and because there is no backward transition from the accept state it remains there. On the other hand if κ does not contain *C* then because of the backward edges the machine cannot access its accept state. The self loops enables that items can be omitted as defined in the subsequence definition.

B. Subtree Inclusion Test

Trees are more complex structures than sequences. The

subtree inclusion test cannot be achieved using deterministic finite state machines because trees cannot be described with regular languages. It means that the machine should be able to count the length of a possible branch when searching for a subtree in the input tree. It can be solved only when using pushdown automaton instead of a state machine.

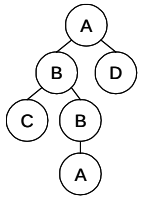
Definition 4: A pushdown automaton is a finite automaton outfitted with access to a potentially unlimited amount of memory called the stack. The pushdown automaton can be defined with a 7-tuple as follows: P (Q, Σ, Γ, δ, q_0, Z_0, F)

- the final set of states, Q,
- the alphabet of the input, Σ,
- the alphabet of the stack, Γ,
- the set of transition functions, δ (Q × (Σ ∪ε) × Γ → Q × Γ^k)
- the start state, $q_0 \in Q$
- the initial stack symbol, $Z_0 \in \Gamma$
- 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 automaton 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 the 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, τ = ∅. Afterwards the tree is traversed in preorder manner starting at the root, and the label x 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 [8], thus the minus signs from the end of the string are omitted. As an example Fig. 2 shows a tree with its string encoding.



τ = ABC–BA– – – D

Fig. 2 A sample tree with its tree encoding

The rules that have to be used for generating the pushdown automaton for detecting a subtree in an input tree is described in Table III. The notations for the rules can be found in Table II. As described in Table III 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.

TABLE II
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 III
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 $q_{pj-1} = max_{k<j}(q_{kj-1})$
$\tau_i = \{-\}$	$(q_{ij}, -, *) \rightarrow (q_{i+1j+1}, \varepsilon)$ $(q_{ij}, \{\lambda \setminus -\}, *) \rightarrow (q_{ij}, \{\lambda \setminus -\} \varepsilon)$

Proposition 5. 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 inputκ 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. 3 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.

IV. SM-Tree and PD-Tree Structures

In Section III the process is shown how an automaton has to be created for detecting whether a candidate pattern (sequence or tree) is contained by another pattern.

Using the finite state machines created for the candidate sequences, and the pushdown automatons created for the candidate trees, it becomes possible to process the transaction in such way that its items are processed exactly once. When processing the items of the transactions the current states of the several finite state machines or pushdown automatons are set according to their transition functions, and the counter of those candidates are incremented whose state machine or pushdown automaton is in the accept state after the last item of the transaction was read. This means, however, that having for instance 500.000 candidates, each time when an item is read 500.000 transition functions have to be followed, which can be very inefficient.

For this reason the state machines or the pushdown automatons have to be joined in order to handle those states together which are the same in the state machines or in the pushdown automatons of the candidates. In this way the computational cost of handling the automatons can be reduced significantly. After joining at least two state machines, a new object, namely, the State Machine-Tree (SM-Tree) is created. And similarly, after joining at least two pushdown automatons, a new object, called Pushdown Automaton-Tree (PD-Tree) is created.

A. State Machine-Tree

Definition 6: Let $M_1 = (Q_1, \Sigma, \delta_1, q_{1_0}, F_1)$ and $M_2 = (Q_2, \Sigma, \delta_2, q_{2_0}, F_2)$ be two finite state machines created for two candidate sequences. The State Machine-Tree (SM-Tree) created by joining these two finite state machines is defined as follows: $SM_3 = M_1 \otimes M_2 = (Q_3, \Sigma, \delta_3, q_{3_0}, F_3)$. The only difference to the finite state machine is in the transition function. The transition function of the State Machine-Tree is defined as follows: $\delta_3 : Q_3 \times \Sigma \to Q_3^2$.

The definition of the transition function of the SM-Tree means that there are states whose transition function results not only one but two states. This means that processing the machine, it can have more than one current states at a time. The reason is that not only one candidate should be found which is contained by the transaction but all of them at the same time. For this reason using this machine is a bit different than using a simple finite state machine. A list is needed in order to keep track of the current states in the machine. When a new item is read, the whole list is to be traversed, and for each state a new state has to be found. This should be made by applying the transition function to the given state and to the most recent item. Two finite state machines can be joined if they share a prefix in common. Because all the machines have a start state which is in common in case of all the candidate sequences, all the machines can be joined to establish an SM-Tree.

The rules for joining two finite state machines are the following. Let l_1 and l_2 denote the number of states of the two state machines M_1 and M_2 respectively. The number of the states of the SM-Tree is denoted with l_3. Let r be the number of those states in the two joined state machines which are the same. Then the number of the states in the resulting State Machine-Tree will be $l_3 = l_1+l_2-r-1$. The states are created from the states of the two finite state machines M_1 and M_2 as shown in Eq.1.

$$Q_{3_i} = \begin{cases} Q_{1_i} \text{ if } 0 \le i < l_1 \\ Q_{2_{i-r-2}} \text{ if } l_1 \le i \le l_3 \end{cases} \quad (1)$$

The accept states of the resulting State Machine-Tree are the union of the accept states of M_1 and M_2 (Eq. 2). The start states are joined, thus all the start states are the same as shown in Eq. 3.

$$F_3 \in Q_3,\ F_3 = F_1 \cup F_2 \quad (2)$$

$$q_{3_0} = q_{1_0} = q_{2_0} \quad (3)$$

The transition functions of M_3 are created from the transition functions of M_1 and M_2.

Proposition 7: 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.

Proof: The first part of the proposition can be proven using Proposition 3, because the SM-Tree is created from the finite state machines created for the candidate sequences. Using the tokens it accepts all the sequences which are contained by the input. If from a state more than one transition functions exist, then a token is placed on the new state and a token remains on the given state such that later from this state another transition function can be used.

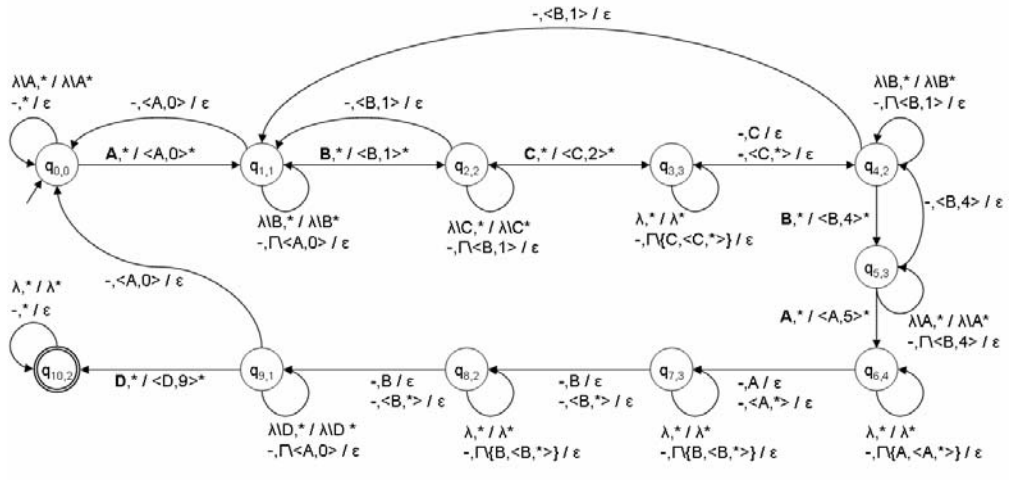


Fig. 3 Sample pushdown automaton for the candidate tree ABC–BA – – D

B. Pushdown Automaton-Tree

Similarly to the sequence mining, in subtree discovery is also 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 considerations is the number of the stack which has to be used. If this cannot be decreased, the benefit of joining the PDA-s is questionable.

Definition 8: 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

- $Q_3 = Q_1 \cup Q_2$,
- $\Sigma_3 = \Sigma_1 \cup \Sigma_2$,
- Γ_3 = extended $\Gamma_1 \cup \Gamma_2$ as described later
- $\delta_3\ (Q_3 \times (\Sigma_3 \cup \varepsilon) \times \Gamma 3 \rightarrow Q_3^2 \times \Gamma_3^k)$,
- $q_{03} = q_{01} = q_{02}$
- $Z_{03} = Z_{01} = Z_{02}$ *and*
- $F_3 = F_1 \cup F_2$.

As it can be seen from the definition, joining two pushdown automatons results in a PD-Tree similarly to joining two state machines that results in an SM-Tree. The rules for generating the new states in the PD-Tree, the accept states and the start states are identical when generating these for the SM-Tree (Eqs. 1, 2 and 3). The only difference is in the stack symbols and of course in the transition functions. In case of PD-Tree the transition function results in not only one state but in some cases also in two, however in case of PD-Tree also the content of the stack has to be used.

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.

Definition 9: 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 10: The PD-Tree created for the 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 PD-Tree 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.

V. Experimental Results

The simulations were executed on a Pentium 4 CPU, 2.4GHz, and 1GB of RAM computer. The SM-Tree and the GSP algorithm was implemented in C++, the PD-Tree algorithm in C{\#}. The SPAM was downloaded from the Himalaya Data Mining Project's website[1]. The Pushdown-Automatons algorithm is an implementation of the pushdown automatons for each candidate without joining them.

Fig. 4 shows the execution time of the GSP, SPAM and SM-Tree algorithms in logarithmic scale. It can be seen well that the SM-Tree algorithm is the fastest one. Fig. 5 shows the execution time of the PushdownAutomatons and the PD-Tree algorithms. It is obvious that the PD-Tree is an order of magnitude faster than the PushdownAutomatons. The reason for that can be observed on Fig. 6 where the number of active states is depicted which were checked for possible transitions during the mining process.

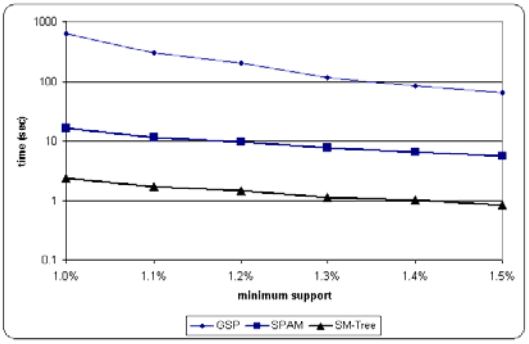


Fig. 4 Execution time of the GSP, SPAM and SM-Tree algorithms

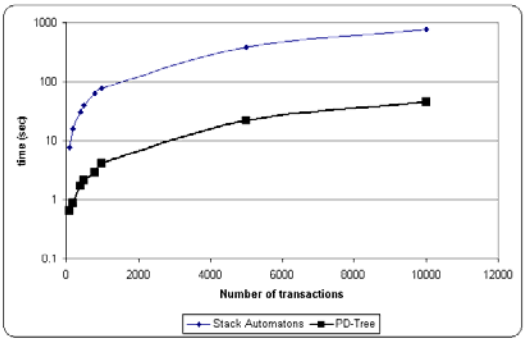


Fig. 5 Execution time of the PushdownAutomatons and the PD-Tree

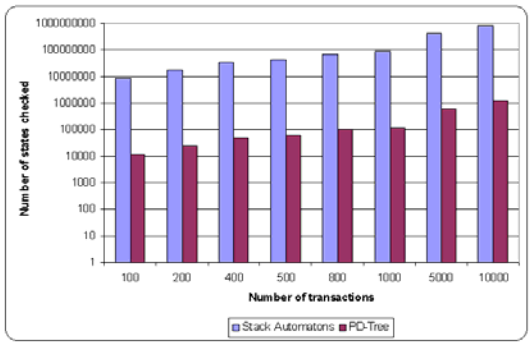


Fig. 6 Number of active states during the mining process

VI. Conclusion

This paper presented a novel approach to frequent pattern mining, namely, using automaton theory for subpattern inclusion testing. The method of constructing the automatons for the candidates was presented. In order to handle the several automatons efficiently two new structures (SM-Tree and PD-Tree) were presented that arises by joining the automatons. Experimental results presented the efficiency of the new algorithms.

References

[1] R. Agrawal and R. Srikant, "Mining sequential patterns," in *ICDE '95: Proceedings of the Eleventh International Conference on Data Engineering*. Washington, DC, USA: IEEE Computer Society, 1995, pp. 3–14.

[2] R. Srikant and R. Agrawal, "Mining sequential patterns: Generalizations and performance improvements," in *Proc. of the 5th International Conference on Extending Database Technology*, 1996.

[3] M. N. Garofalakis, R. Rastogi, and K. Shim, "Spirit: Sequential pattern mining with regular expression constraints." in VLDB, M. P. Atkinson, M. E. Orlowska, P. Valduriez, S. B. Zdonik, and M. L. Brodie, Eds. Morgan Kaufmann, 1999, pp. 223–234.

[4] M. J. Zaki, "Spade: An efficient algorithm for mining frequent sequences," Machine Learning, vol. 42, no. 1-2, pp. 31–60, 2001.

[5] J. Han, J. Pei, and B. M.-A. et al., "Freespan: frequent pattern-projected sequential pattern mining," in KDD '00: Proceedings of the sixth ACM SIGKDD international conference on Knowledge discovery and data mining. New York, NY, USA: ACM Press, 2000, pp. 355–359.

[6] J. Pei, J. Han, B. Mortazavi-Asl, H. Pinto, Q. Chen, U. Dayal, and M.-C. Hsu, "PrefixSpan mining sequential patterns efficiently by prefix projected pattern growth," in In Proc. of Int. Conf. on Data Engineering, 2001, pp. 215–226.

[7] J. Ayres, J. Flannick, J. Gehrke, and T. Yiu, "Sequential pattern mining using a bitmap representation," in In Proc. of the 8th ACM SIGKDD Int.Conf. on Knowledge Discovery and Data Mining, 2002, pp. 429–435.

[8] M. Zaki, "Efficiently mining frequent trees in a forest," in Proc. of the 8th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining, July 2002, pp. 71–80.

[9] T. Asai, K. Abe, S. Kawasoe, H. Arimura, H. Satamoto, and S. Arikawa, "Efficient substructure discovery from large semi-structured data." In SDM, R. L. Grossman, J. Han, V. Kumar, H. Mannila, and R. Motwani, Eds. SIAM, 2002.

[10] U. Rckert and S. Kramer, "Frequent free tree discovery in graph data," in SAC '04: Proceedings of the 2004 ACM symposium on Applied computing. New York, NY, USA: ACM Press, 2004, pp. 564–570.

[1] http://www.cs.cornell.edu/database/himalaya