Vietnam J Comput Sci (2014) 1:97–105
DOI 10.1007/s40595-013-0012-3

REGULAR PAPER

IMSR_PreTree: an improved algorithm for mining sequential rules based on the prefix-tree

Thien-Trang Van · Bay Vo · Bac Le

Received: 30 September 2013 / Accepted: 19 December 2013 / Published online: 30 January 2014

Abstract Sequential rules generated from sequential patterns express temporal relationships among patterns. Sequential rule mining is an important research problem because it has broad application such as the analyses of customer purchases, web log, DNA sequences, and so on. However, developing an efficient algorithm for mining sequential rules is a difficult problem due to the large size of the sequential pattern set. The larger the sequential pattern set, the longer the mining time. In this paper, we propose a new algorithm called *IMSR_PreTree* which is an improved algorithm of *MSR_PreTree* that mines sequential rules based on prefix-tree. *IMSR_PreTree* also generates rules from frequent sequences stored in a prefix-tree but it prunes the sub trees which give non-significant rules very early in the process of rule generation and avoids tree scanning as much as possible. Thus, *IMSR_PreTree* can significantly reduce the search space during the mining process. Our performance study shows that *IMSR_PreTree* outperforms *MSR_PreTree*, especially on large sequence databases.

Keywords Frequent sequence · Prefix-tree · Sequential rule · Sequence database

T.-T. Van
Faculty of Information Technology, Ho Chi Minh City University of Technology, Ho Chi Minh, Vietnam
e-mail: vtt.trang@hutech.edu.vn

B. Vo (✉)
Faculty of Information Technology, Ton Duc Thang University, Ho Chi Minh, Vietnam
e-mail: bayvodinh@gmail.com

B. Le
Faculty of Information Technology, University of Science, VNU, Ho Chi Minh, Vietnam
e-mail: lhbac@fit.hcmus.edu.vn

1 Introduction

In sequence databases, there are researches on many kinds of rules, such as recurrent rules [13], sequential classification rules [4], sequential rules [14,18,21,22], and so on. In this paper, we focus on usual sequential rule mining. We try to address the problem of effectively generating a full set of sequential rules.

Sequential rule mining is to find the relationships between occurrences of sequential events. A sequential rule is an expression that has form X→Y, i.e., if X occurs in any sequence of the database then Y also occurs in that sequence following X with the high confidence. The mining process is usually decomposed into two phases:

(1) Mining all frequent sequences that have supports above the minimum support threshold (*minSup*)
(2) Generating the desired rules from the frequent sequences if they also satisfy the minimum confidence threshold (*minConf*).

Most of the previous researches with regard to sequential rules require multiples of passes over the full set of frequent sequences. Recently, a sequential rule mining method based on prefix-tree [21] achieves high efficiency. Using prefix-tree, it can immediately determine which sequences contain a given sequence as a prefix. So the rules can be generated directly from those frequent sequences without browsing over the whole set of frequent sequence many times.

The rest of paper is organized as follows: Sect. 2 introduces the basic concepts related to sequence mining and some definitions used throughout the paper. Section 3 presents the related work. The improved algorithm of *MSR_PreTree* is presented in Sect. 4 and an example is given in Sect. 5. Exper-

imental results are conducted in Sect. 6. We summarize our study and discuss the future work in Sect. 7.

2 Preliminary concepts

Let I be a set of distinct items. An itemset is a subset of items (without loss of generality, we assume that items of an itemset are sorted in lexicographic order). A sequence $s = \langle s_1 s_2 \ldots s_n \rangle$ is an ordered list of itemsets. The size of a sequence is the number of itemsets in the sequence. The length of a sequence is the number of items in the sequence. A sequence with length k is called a k-sequence.

A sequence $\beta = \langle b_1 b_2 \ldots b_m$ is called a subsequence of another sequence $\alpha = \langle a_1 a_2 \ldots a_n \rangle$, denoted as $\beta \subseteq \alpha$, if there exist integers $1 \leq i_1 < i_2 < \cdots < i_m \leq n$ such that $b_1 \subseteq a_{i1}, b_2 \subseteq a_{i2}, \ldots, b_m \subseteq a_{im}$. Given a sequence database, the support of a sequence α is defined as the number of sequences in the sequence database that contains α. Given a *minSup*, we say that a sequence is frequent if its support is greater than or equal to *minSup*. A frequent sequence is also called a sequential pattern.

Definition 1 *(Prefix, incomplete prefix and postfix).* Given a sequence $\alpha = \langle a_1 a_2 \ldots a_n \rangle$ and a sequence $\beta = \langle b_1, b_2, \ldots b_m \rangle$ $(m < n)$, (where each a_i, b_i corresponds to an itemset). β is called prefix of α if and only if $b_i = a_i$ for all $1 \leq i \leq m$. After removing the prefix β of the sequence α, the remaining part of α is a postfix of α. Sequence β is called an incomplete prefix of α if and only if $b_i = a_i$ for $1 \leq i \leq m-1$, $b_m \subset a_m$ and all the items in $(a_m - b_m)$ are lexicographically after those in b_m.

Note that from the above definition, a sequence of size k has $(k-1)$ prefixes. For example, sequence $\langle$(A)(BC)(D)$\rangle$ have two prefixes: $\langle$(A)$\rangle$, $\langle$(A)(BC)$\rangle$. Consequently, $\langle$(BC)(D)$\rangle$ is the postfix with respect to prefix $\langle$(A)$\rangle$ and $\langle$(D)$\rangle$ is the postfix w.r.t prefix $\langle$(A)(BC)$\rangle$. Neither $\langle$(A)(B)$\rangle$ nor $\langle$(BC)$\rangle$ is considered as a prefix of given sequence; however, $\langle$(A)(B)$\rangle$ is an incomplete prefix of given sequence.

A sequential rule is built by splitting a frequent sequence into two parts: prefix *pre* and postfix *post* (concatenating *pre* with *post*, denoted as *pre* ++ *post*, we have the same pattern as before [14]). We denote a sequential rule as *rule* = *pre* → *post (sup, conf)*.

The support of a *rule*: *sup* = *sup(pre++post)*.

The confidence of a *rule*: *conf* = *sup(pre++post)/sup(pre)*.

Note that *pre* ++ *post* is a frequent sequence; consequently *pre* is also a frequent sequential pattern (by the Apriori principle [2]). For each frequent sequence f of size k, we can possibly form $(k-1)$ rules. For example, if we have a frequent sequence $\langle$(A)(BC)(D)$\rangle$ whose size is 3, then we can generate two rules $\langle$(A)$\rangle \rightarrow \langle$(BC)(D)$\rangle$, $\langle$(A)(BC)$\rangle \rightarrow \langle$(D)$\rangle$.

Sequential rule mining is to find out all significant rules that satisfy *minSup* and *minConf* from the given database. The support and confidence thresholds are usually predefined by users.

3 Related work

3.1 Mining sequential patterns

Sequential pattern mining is to find all frequent sub sequences as sequential patterns in a sequence database. A sequence database is a large collection of records, where each record is an ordered list of events. Each event can have one or many items. For examples, the customer purchase database is a popular database in which each event is a set of items (itemset) and some other databases such as DNA sequences and Web log data are typically examples of the databases in which all events are single items.

The problem of mining sequential patterns was first introduced by Agrawal and Srikant in [2]. They also presented three algorithms to solve this problem. All these algorithms are variations of the Apriori algorithm [1] proposed by the same authors and used in association rule mining. Many other approaches have been developed since then [2,3,11,12,15,16,19]. The general idea of all methods is outlined as follows: starting with more general (shorter) sequences and extending them towards more specific (longer) ones. However, existing methods uses specified data structures to "represent" the database and have different strategies to traverse and enumerate the search space.

GSP [19] which is also a typical Apriori-like method adopts a multiple pass, candidate generation, and test approach in sequential pattern mining. But it incorporates time constraints, sliding time windows, and taxonomies in sequence patterns. PSP [15] which is another Apriori-based algorithm also builds around GSP, but it uses a different intermediary data structure which is proved more efficient than that used in GSP.

Based on the "divide and conquer" philosophy, FreeSpan [12] is the first algorithm which projects the data sequences into smaller databases for reducing the I/O costs. This strategy has been continually used in PrefixSpan [16]. Starting from the frequent items of the database, PrefixSpan generates projected databases with each frequent item. Thus, the projected databases contain only suffixes of the data-sequences from the original database, grouped by prefixes. The pattern is extended by adding one item in frequent itemsets which are obtained from projected databases. The process is recursively repeated until no frequent item is found in the projected database.

All the above methods utilize the same approach in sequence mining: the horizontal data sequence layout. And

the corresponding algorithms make multiples of passes over the data for finding the supports of candidate sequences. Some methods have approached the vertical data layout to overcome this limitation; they are SPADE [22], SPAM [3] and PRISM [10]. Instead of browsing the entire sequence database for each pattern, these algorithms store the information indicating which data sequences contain that pattern so that its support is determined quickly. Moreover, the information of a new pattern is also constructed from old patterns' information.

The SPADE algorithm [22] uses a vertical id-list representation of the database where it stores a list of all input sequence (sequence id) and event identifier (event id) pairs containing the pattern for each level of the search space for each pattern. On the other hand, SPADE uses a lattice-theoretic approach to decompose the search space and uses simple join operations to generate candidate sequences. SPADE discovers all frequent sequences in only three database scans. The SPADE algorithm outperforms GSP by a factor of two at lower support values [22].

SPAM [3] is another method that maintains the information of a pattern. It uses a vertical bitmap representation of the database for both candidate representation and support counting. SPAM outperforms SPADE by about a factor of 2.5 on small datasets and better than an order of magnitude for reasonably large datasets.

PRISM [10] was proposed in 2010. It is one of the most efficient methods for frequent sequence mining [10]. Similar to SPAM, PRISM also utilizes a vertical approach for enumeration and support counting but it is different from the previous algorithms. This algorithm uses the prime block encoding approach to represent candidate sequences and uses join operations over the prime blocks to determine the support of each candidate. Especially, all sequential patterns which are found by this algorithm are organized and stored in a tree structure which is the basis for our proposed algorithm—*MSR_PreTree*.

Besides, there are several sequential pattern mining techniques that are simply applied to web log mining such as WAP-Mine [17] with WAP-tree; and PLWAP [7] with its PLWAP-tree, FS-Miner [6], etc.

3.2 Generating rules from sequential patterns

In this section, we discuss existing contributions in sequential rule mining research. Sequential rules [14,18,21,22] are "if-then rules" with two measures which quantify the support and the confidence of the rule for a given sequence database. For example, if a customer buys a car, then he will also buy a car insurance.

The study in [18] has proposed a method for generating a complete set of sequential rules from frequent sequences and removing redundant rules in the post-mining phase. Based on the description in [18], Lo et al. [14] generalized the *Full* algorithm for mining a full set of sequential rules and it is completely the same algorithm as the RuleGen algorithm proposed by Zaki [22].

Full algorithm: First, the algorithm finds all frequent sequences (*FS*) whose support is no less than *minSup* by using an existing method. For each frequent sequence, the algorithm generates all sequential rules from that frequent sequence. We now describe the process of rule generation in more detail.

For each frequent sequence f of size k, it is possible to generate $(k-1)$ rules. Each rule is expressed as $pre \rightarrow post$, in which *pre* is a prefix of sequence f and $pre{+}{+}post = f$. For example, from a sequence $\langle(\text{AB})(\text{B})(\text{C})\rangle$, we can generate two candidate sequential rules: $\langle(\text{AB})\rangle \rightarrow \langle(\text{B})(\text{C})\rangle$ and $\langle(\text{AB})(\text{B})\rangle \rightarrow \langle(\text{C})\rangle$. For each frequent sequence f, *Full* generates and examines every prefixes of f in turn. For each prefix *pre*, it tries to form a rule $pre \rightarrow post$ where *post* is the postfix of f with respect to prefix *pre*. After that, it passes over *FS* to find that the prefix's support and calculates the confidence of that rule. If the confidence is not less than the minimum confidence threshold, we have a significant sequential rule. This process is repeated until all frequent sequences are considered. Let n be the number of sequences in *FS*, and k be the size of the largest sequence in *FS*; the complexity of this algorithm is $O(n * k * n)$ (without including the time of prefix generation). The major cost of *Full* is the database scan. It has to scan the database many times for counting the support of every prefixe of a frequent sequence.

In order to improve the runtime, we proposed two algorithms: *MSR_ImpFull* and *MSR_PreTree* [21].

MSR_ ImpFull algorithm: It is an improved version of *Full*. As mentioned above, to generate rules from a frequent sequence, *Full* has to split it to get the prefix and has to browse *FS* to get the prefix's support. To overcome this, *MSR_ImpFull* sorts *FS* in ascending order of their size before generating rules. After sorting, we have the fact that the sequences which contain sequence X as a prefix must be after X because their size is larger. Therefore, for a frequent sequence f in *FS, MSR_ImpFull* does not have to split it; it browses the sequences following f to find which sequences contain f as a prefix. After that, it generates rules from the found sequences and f. The generated rule is $f \rightarrow post$, where post is the postfix of the found sequence with prefix f. The confidence is calculated directly from the found sequence's support and f's support. *MSR_ImpFull* reduces k *FS* scans for each sequence of size k when compared with *Full*. So, if we consider only the number of *FS* scans without the time of prefix checking, then the complexity of *MSR_ImpFull* is $O(n * n)$.

MSR_PreTree algorithm: Although *MSR_ImpFull* algorithm has less number of *FS* scans than *Full*, for each

sequence f, it scans all sequences following f to identify which sequence contains f as a prefix. In order to overcome this, *MSR_PreTree* uses a prefix-tree structure presented in Sect. 4. Every node in the prefix-tree is always the prefix of its child nodes (only sequence-extended nodes). Consequently, it is possible to generate rules directly without scanning all frequent sequences for prefix checking.

Besides, Fournier-Viger et al. [8–10] have recently discussed a more general form of sequential rules such that items in the prefix and in the postfix of each rule are unordered. However, our study simply focuses on mining usual sequential rules.

4 An Improved algorithm of MSR_PreTree

4.1 Prefix-Tree

In our approach (*MSR_PreTree*, *IMSR_PreTree*), the set of rules is generated from frequent sequences, which is organized and stored in a prefix-tree structure as illustrated in Fig. 1. In this section, we briefly outline the prefix-tree.

Each node in the prefix-tree stores two pieces of information: label and support, denoted as *label: sup*, in which label is a frequent sequence and support is the support count of that frequent sequence.

The prefix-tree is built recursively as follows: starting from the root of tree at level 0, the root is a virtual node labeled with a null sequence ϕ and the support is 0. At level k, a node is labeled with a k-sequence and its support count. Recursively, we have nodes at the next level $(k+1)$ by extending the k-sequence with a frequent item. If the support of the new extended sequence satisfies *minSup*, then we store that sequence and continue the extension recursively. There are two ways to extend a k-sequence: *sequence extension* and *itemset extension* [10]. In *sequence extension*, we add an item to the sequence as a new itemset. Consequently, the size of a sequence-extended sequence always increases.

Remark 1 In sequence extension, a k-sequence α is a prefix of all sequence-extended sequences. Moreover, α is certainly the prefix of all child nodes of the nodes which are sequence-extended of α.

In *itemset extension*, the item is added to the last itemset in the sequence so that the item is greater than all items in the last itemset. So the size of the itemset-extended sequence does not change.

Remark 2 In itemset extension, α is an incomplete prefix of all itemset-extended sequences. Moreover, α is an incomplete prefix of all child nodes of the nodes which are itemset-extended of α.

For example, Fig. 1 shows the prefix-tree of frequent sequences. ⟨(A)(A)⟩ and ⟨(A)(B)⟩ are sequence-extended sequences of ⟨(A)⟩, and ⟨(AB)⟩ is an itemset-extended sequence of ⟨(A)⟩. Sequence ⟨(A)⟩ is a prefix of all sequences in T1 and it is an incomplete prefix of sequences in T2.

4.2 Theorem

In *MSR_PreTree*, for each node r in the prefix-tree, the algorithm browses all the child nodes of r and considers each candidate rule generated by each child node with respect to prefix *r.sequence*, if the candidate rule's confidence satisfies *minConf* then that rule is outputted. Thus, the algorithm must still browse the child nodes which produce non-significant rules

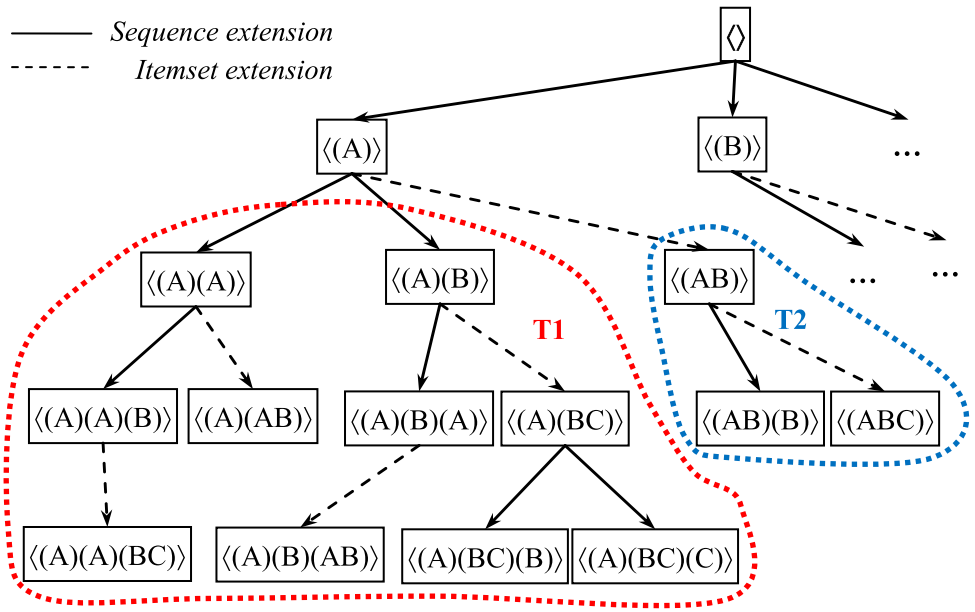


Fig. 1 The prefix-tree

```
Input: FS - the set of frequent sequences stored in pre-
fix-tree, minConf
Output: All significant rules
Algorithm:
GENERATE_RULES_FROM_PREFIX-TREE()
     1. For each node r at level 1 in prefix-tree
     2.   GENERATE_RULES_FROM_ROOT(r)
GENERATE_RULES_FROM_ROOT(r)
     3. Let pre = r.sequence
     4. For each node nseq ∈ r.Seq_Extended_Set
     5.   For each node l ∈ sub-tree nseq
     6.     Try to form rule = pre→post,
                          pre++post = l.sequence
     7.     If sup(l.sequence)/sup(pre) ≥ minConf
     8.         Output rule
     9.     Else Stop generating rules from l.children
              with prefix pre //according to theorem 1
    10. For each node nseq ∈ r.Seq_Extended_Set
    11.   GENERATE_RULES_FROM_ROOT(nseq)
    12. For each node nseq ∈ r.Items_Extended_Set
    13.   GENERATE_RULES_FROM_ROOT(nseq)
```

Fig. 2 IMSR_PreTree—an improved algorithm of MSR_PreTree

with respect to prefix *r.sequence*. For example, if node r has 1,000 child nodes, then it has to browse all those nodes, calculates the confidence of all those nodes, and checks the confidence with *minConf* even if there are certainly nodes which produce non-significant rules among those nodes. Considering the recursive characteristic in the implementation, this cost is too high. So, it is necessary to avoid browsing the child nodes which do not produce significant rules. Consequently, the key problem is that how to know which child nodes certainly produce non-significant rules.

To overcome this problem, we improve *MSR_PreTree* to early prune the sub trees which produce non-significant rules. We have the following theorem [5]:

Theorem 1 *Given three nodes n_1, n_2 and n_3 in the prefix-tree, if n_1 is the parent node of n_2, n_2 is the parent node of n_3 and $\frac{sup(n_2)}{sup(n_1)} < minConf$, then $\frac{sup(n_3)}{sup(n_1)} < minConf$.*

Proof Since n_1 is the parent node of n_2 and n_2 is the parent node of n_3, n_1 sequence $\subset n_2.sequence \subset n_3.sequence$. This implies that $n_1.sup \geq n_2.sup \geq n_3.sup$. Thus, $\frac{n_2.sup}{n_1.sup} \geq \frac{n_3.sup}{n_1.sup}$. Since $\frac{n_2.sup}{n_1.sup} < minconf$, it implies $\frac{n_3.sup}{n_1.sup} < minConf$.

According to the above theorem, if a tree node Y is a child node of X in the prefix-tree and *sup*(Y)/*sup*(X) < *minConf*, then all the child nodes of Y cannot form rules with X because the confidence of each generated rule from Y's child nodes is less than *minConf*. For example, consider two nodes ⟨(A)⟩ and ⟨(A)(B)⟩ in Fig. 4. Assume that *minConf* = 80 %, since *sup*(⟨(A)(B)⟩)/*sup*(⟨(A)⟩) = 3/4 < *minConf*, the child nodes {⟨(A)(B)(B)⟩, ⟨(A)(B)(C)⟩} of ⟨(A)(B)⟩ in prefix-tree cannot generate the rules with prefix ⟨(A)⟩.

4.3 IMSR_PreTree algorithm

The improved algorithm of *MSR_PreTree* is shown in Fig. 2. Using PRISM [10] we have a full set of frequent sequences stored in the prefix-tree structure which gathers all sequences with a common prefix into a sub-tree. Thus, we generate rules within each common prefix sub-tree. Starting from the first level, each sub-tree rooted at a node r labeled with 1−sequence can be processed independently to generate sequential rules using procedure *GENERATE_RULES_FROM_ROOT(r)*. This procedure is performed using the following steps:

Step 1 Generating rules with the prefix that is a sequence at the root. Let *pre* be the sequence at root node*r (line 3)*. From Remarks 1 and 2, we find that sequence *pre* is the prefix of all the sequences in sequence-extended sub-trees of the root r. Hence, we generate only rules from the sequences in these sub-trees in turn. The following are the descriptions of rule generation from each sub-tree with prefix *pre* (lines 3–9).

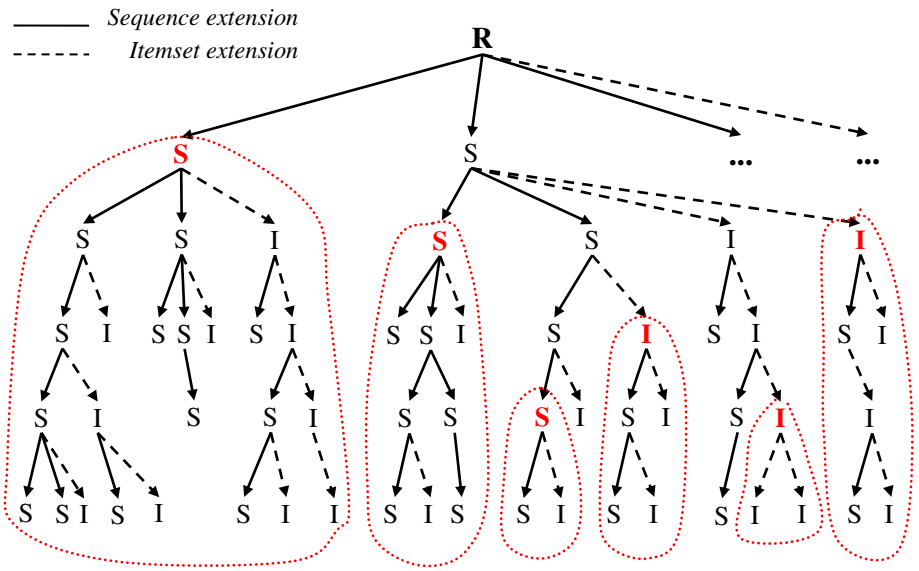


Fig. 3 Some pruning cases when generating rules with the prefix *R.sequence*

For each *l.sequence* at node *l* in the tree rooted at *nseq* (that is the sub-tree as mentioned above) including the root and sequence-extended children and itemset-extended children, we consider the following rule: pre→post where *post* is a postfix of *l.sequence* with respect to the prefix *pre (r.sequence)*. We apply the above theorem when we have already determined the confidence of the rule generated from node *l*. If the confidence satisfies *minConf*, we output that rule and repeat recursively on extended children of *l*. Otherwise, we completely prune all the sub-trees of *l*. It means that we do not need to generate rules from all child nodes and descendents of the node *l* with prefix *pre*. After that, we continue considering the other nodes by backtracking up the tree *nseq* if it still has child nodes.

In this process (lines 5–9), we traverse the sub-tree in a depth-first manner so that the pruning technique can be applied. Figure 3 shows some pruning cases when generating rules with the prefix *R.sequence*. The sub-trees rooted at the nodes (highlighted) which generate rules having the confidence less than *minConf* are pruned. These sub-trees are marked by surrounding line.

Step 2: Because all extended nodes of the current root are the prefix of the sub trees at the next level, we call this procedure recursively for every extended-nodes of the root (lines 10–13). This process is recursively repeated until reaching the last level of the tree.

5 An example

In this section, an example is given to illustrate the proposed algorithm for mining sequential rules. Consider the database shown in Table 1. It consists of five data sequences and three different items.

Table 1 An example database

SID	Data sequence
1	⟨(AB)(B)(B)(AB)(B)(AC)⟩
2	⟨(AB)(BC)(BC)⟩
3	⟨(B)(AB)⟩
4	⟨(B)(B)(BC)⟩
5	⟨(AB)(AB)(AB)(A)(BC)⟩

Using PRISM algorithm [10], all found frequent sequences are stored in a prefix-tree. The prefix-tree built from the database in Table 1 with *minSup* = 50 % is shown in Fig. 4. After that, we generate all rules from those frequent sequences. Assume that *minConf* = 80 %.

Table 2 show the result of sequential rule generation from all frequent sequences stored in the prefix-tree. For details, consider the root node ⟨(A)⟩ shown in Fig. 4; the itemset-extended sequence of ⟨(A)⟩ is ⟨(AB)⟩ and the sequence-extended sequences of ⟨(A)⟩ are ⟨(A)(B)⟩ and ⟨(A)(C)⟩. Because ⟨(A)⟩ is an incomplete prefix of ⟨(AB)⟩ and all child nodes of ⟨(AB)⟩ are in T_2, we do not generate rules from those nodes with prefix ⟨(A)⟩. On the contrary, since ⟨(A)⟩ is a prefix of all the sequences in T_1, we generate rules from the sequences in T_1 with prefix ⟨(A)⟩. First, for sequence ⟨(A)(B)⟩ we have a rule: ⟨(A)⟩ → ⟨(B)⟩. However, its confidence is less than *minConf*; we, therefore, eliminate it. Moreover, we completely prune all sub-trees generated from ⟨(A)⟩ → ⟨(B)⟩. It means that we stop generating rules from all child nodes {⟨(A)(B)(B)⟩, ⟨(A)(B)(C)⟩} of ⟨(A)(B)⟩ (based on the above theorem). Similarly, for sequence ⟨(A)(C)⟩, we have a candidate rule ⟨(A)⟩ → ⟨(C)⟩ and its confidence does not satisfy *minConf*. Thus, we eliminate it.

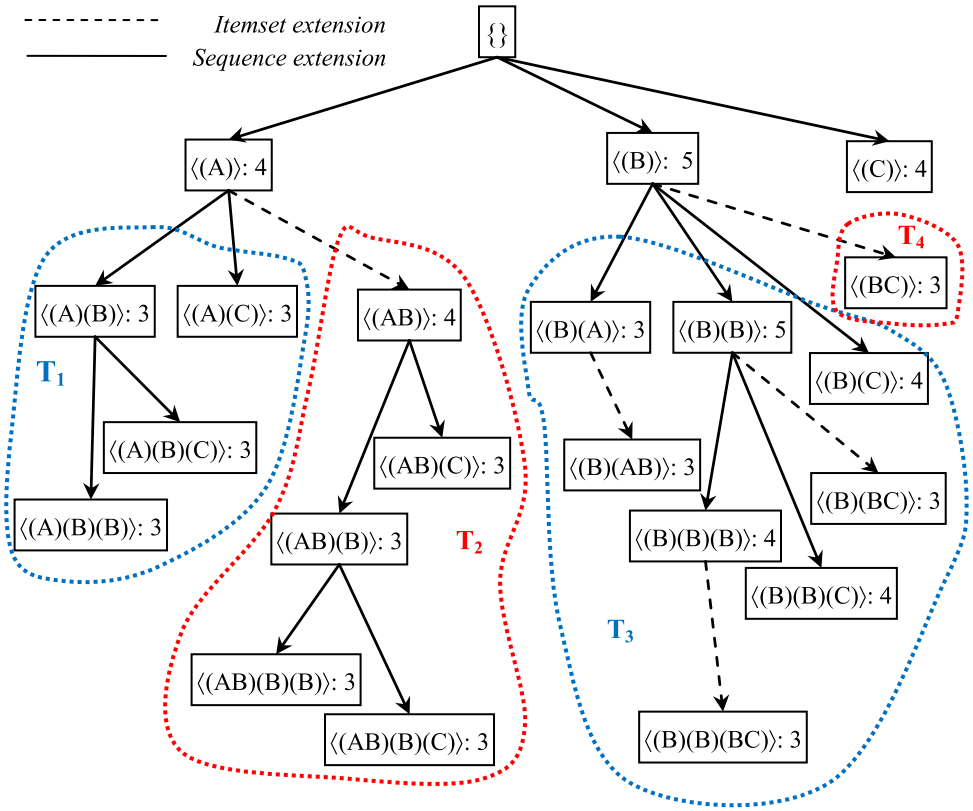


Fig. 4 The prefix-tree built from the database in Table 1 with *minSup* = 50 %

Table 2 The result of sequential rule generation from the frequent sequences stored in prefix-tree, *minConf* = 80 %

Prefix	Sequence	Sequential rule, $conf\,conf = sup(X++Y)/sup(X) \times 100\%$	$conf \geq minConf?$
⟨(A)⟩: 4	⟨(A)(B)⟩: 3	⟨(A)⟩ → ⟨(B)⟩, 75 %	No
	⟨(A)(B)(B)⟩: 3	Stop generating the rules with prefix ⟨(A)⟩	
	⟨(A)(B)(C)⟩: 3		
	⟨(A)(C)⟩: 3	⟨(A)⟩ → ⟨(C)⟩, 75 %	No
⟨(A)(B)⟩:3	⟨(A)(B)(B)⟩: 3	⟨(A)(B)⟩ → ⟨(B)⟩, 100 %	Yes
	⟨(A)(B)(C)⟩: 3	⟨(A)(B)⟩ → ⟨(C)⟩, 100 %	Yes
⟨(AB)⟩: 4	⟨(AB)(B)⟩: 3	⟨(AB)⟩ → ⟨(B)⟩, 75 %	No
	⟨(AB)(B)(B)⟩: 3	Stop generating the rules with prefix ⟨(AB)⟩	
	⟨(AB)(B)(C)⟩: 3		
	⟨(AB)(C)⟩: 3	⟨(AB)⟩ → ⟨(C)⟩, 75 %	No
⟨(AB)(B)⟩: 3	⟨(AB)(B)(B)⟩: 3	⟨(AB)(B)⟩ → ⟨(B)⟩, 100 %	Yes
	⟨(AB)(B)(C)⟩: 3	⟨(AB)(B)⟩ → ⟨(C)⟩, 100 %	Yes
⟨(B)⟩: 5	⟨(B)(A)⟩: 3	⟨(B)⟩ → ⟨(A)⟩, 60 %	No
	⟨(B)(AB)⟩: 3	Stop generating the rules with prefix ⟨(B)⟩	
	⟨(B)(B)⟩: 5	⟨(B)⟩ → ⟨(B)⟩, 100 %	Yes
	⟨(B)(B)(B)⟩: 4	⟨(B)⟩ → ⟨(B)(B)⟩, 80 %	Yes
	⟨(B)(B)(BC)⟩: 3	⟨(B)⟩ → ⟨(B)(BC)⟩, 60 %	No
	⟨(B)(B)(C)⟩: 4	⟨(B)⟩ → ⟨(B)(C)⟩, 80 %	Yes
	⟨(B)(BC)⟩: 3	⟨(B)⟩ → ⟨(BC)⟩, 60 %	No
	⟨(B)(C)⟩: 4	⟨(B)⟩ → ⟨ (C)⟩, 80 %	Yes
⟨(B)(B)⟩: 5	⟨(B)(B)(B)⟩: 4	⟨(B)(B)⟩ → ⟨(B)⟩, 80 %	Yes
	⟨(B)(B)(BC)⟩: 3	⟨(B)(B)⟩ → ⟨(BC)⟩, 60 %	No
	⟨(B)(B)(C)⟩: 4	⟨(B)(B)⟩ → ⟨(C)⟩, 80 %	Yes

Table 3 Database characteristics (http://www.ics.uci.edu/~mlearn)

Databases	*#FS*	#Distinct items	Aver. sequence size
Chess	3,196	75	37
Mushroom	8,124	119	23

We repeat the whole above process for all child nodes ⟨(A)(B)⟩, ⟨(A)(C)⟩ and ⟨(AB)⟩. Similar processing is applied to root nodes ⟨(B)⟩ and ⟨(C)⟩.

6 Experimental results

In this section, we report our experimental results on comparing the performance of *IMSR_PreTree* with *MSR_PreTree*. Results show that *IMSR_PreTree* outperforms *MSR_PreTree* and *IMSR_PreTree* is an efficient and scalable approach for mining sequential patterns in large databases.

Our experiments are performed on a computer with Intel Core 2 Duo CPU T8100 2x2.10GHz, and 2 GB of memory, running Windows XP Professional. All programs are written in C#.

In the past [15], we have already experimented on synthetic databases created from the synthetic data generator provided by IBM and also on real the database Gazelle in which the execution time is less than 1 second. Consequently, to see the difference in time between *MSR_PreTree* and *IMSR_PreTree*, we perform experiments on some databases in which the number of frequent sequences is very large. Relying on that, the number of generated rules is very large too. Chess and Mushroom are the databases used in the problem of mining association rules from frequent itemsets. However, using these databases in the problem of mining sequential rules from frequent sequences, we consider each itemset in these databases is a data sequence. Consequently, the data sequence's itemset now has only one item. In this manner, Chess and Mushroom are considered as the databases in which a data sequence is a list of 1−itemset, for example, DNA databases, web log, etc. Using these databases, the number of generated sequential rules is up to hundreds of millions of rules.

These databases are downloaded from the UCI Machine Learning Repository [20]. Table 3 shows the characteristics of those databases.

Figure 5 shows the performance comparison of the two algorithms on Chess with *minSup* is 35 and 40 % and *minConf* varying from 50 to 99 %. When the confidence threshold is low, the gap between the numbers of candidate rules and significant rules is small and two algorithms are close in terms of runtime. However, when the confidence threshold is increased, the gaps become clear because there are many more pruned rules. Both algorithms generate the same rules, but *IMSR_PreTree* ismore efficient than *MSR_PreTree*. At

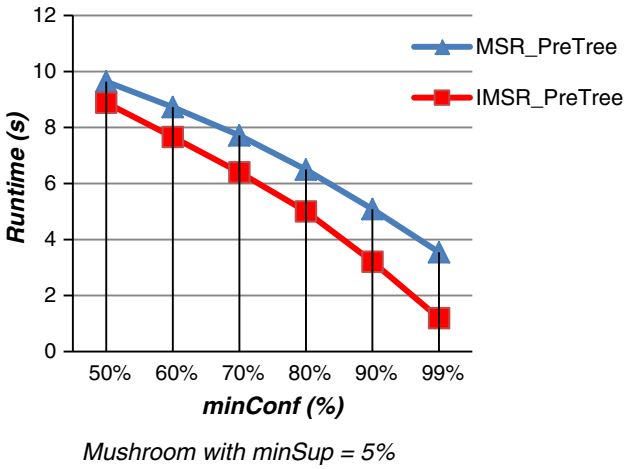
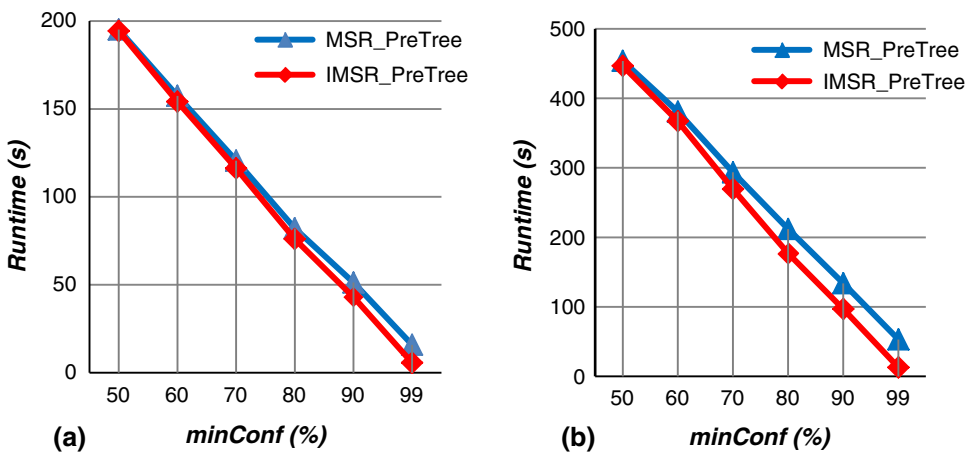


Fig. 6 Performance comparison between two algorithms on mushroom

Fig. 5 Performance comparison between two algorithms on chess



the highest value of confidence, *IMSR_PreTree* is about 3 and 4 times faster than *MSR_PreTree* with *minSup*=40 and 35 %, respectively. In addition, we see that *IMSR_PreTree* outperforms *MSR_PreTree* by an order of magnitude in most cases. We also test two algorithms on Mushroom and obtain similar results (Fig. 6).

7 Conclusions and future work

In this paper, we have developed an efficient algorithm named *IMSR_PreTree* which is an improved algorithm of *MSR_PreTree* to mine sequential rules from sequence databases. The aim of this improvement is to reduce the cost of sequential rule mining process by pruning the sub trees which give non-significant rules in the early stage of mining. It can greatly reduce the search space during the mining process and it is very effective in mining large databases.

Experimental results show that *IMSR_PreTree* is faster than *MSR_PreTree*. In the future, we will apply this approach to rule generation with many kinds of interestingness measures. Especially, we will use the prefix-tree structure to mine sequential patterns with constraints.

Acknowledgments This work was funded by Vietnam's National Foundation for Science and Technology Development (NAFOSTED) under Grant Number 102.05-2013.20.

References

1. Agrawal, R., Srikant, R.: Fast algorithms for mining association rules. In: Proceedings of the 20th international conference very large data, bases, pp. 487–499 (1994)
2. Agrawal, R., Srikant, R.: Mining sequential patterns. In: Proceedings of the 11th international conference on data engineering, pp. 3–14. IEEE (1995)
3. Ayres, J., Flannick, J., Gehrke, J., Yiu, T.: Sequential pattern mining using a bitmap representation. In: Proceedings of the 8th ACM SIGKDD international conference on knowledge discovery and data mining, pp. 429–435. ACM (2002)
4. Baralis, E., Chiusano, S., Dutto, R.: Applying sequential rules to protein localization prediction. Comput. Math. Appl. **55**(5), 867–878 (2008)
5. Vo, B., Hong, T.P., Le, B.: A lattice-based approach for mining most generalization association rules. Knowl. Based Syst. **45**, 20–30 (2013)
6. El-Sayed, M., Ruiz, C., Rundensteiner, E.A.: FS-Miner: efficient and incremental mining of frequent sequence patterns in web logs. In: Proceedings of the 6th annual ACM international workshop on web information and data management, pp. 128–135 (2004)
7. Ezeife, C.I., Lu, Y., Liu, Y.: PLWAP sequential mining: open source code. In: Proceedings of the 1st international workshop on open source data mining: frequent pattern mining implementations, pp. 26–35 (2005)
8. Fournier-Viger, P., Faghihi, U., Nkambou, R., Nguifo, E.M.: CMRules: mining sequential rules common to several sequences. Knowl. Based Syst. **25**(1), 63–76 (2012)
9. Fournier-Viger, P., Nkambou, R., Tseng, V.S.M.: RuleGrowth: mining sequential rules common to several sequences by pattern-growth. In: Proceedings of the 2011 ACM symposium on applied computing, pp. 956–961 (2011)
10. Fournier-Viger, P., Wu, C.W., Tseng, V.S., Nkambou, R.: Mining sequential rules common to several sequences with the window size constraint. Adv. Artif. Intell. 299–304 (2012)
11. Gouda, K., Hassaan, M., Zaki, M.J.: Prism: an effective approach for frequent sequence mining via prime-block encoding. J. Comput. Syst. Sci. **76**(1), 88–102 (2010)
12. Han, J., Pei, J., Mortazavi-Asl, B., Chen, Q., Dayal, U., Hsu, M.C. FreeSpan: frequent pattern-projected sequential pattern mining. In: Proceedings of the 6th ACM SIGKDD international conference on knowledge discovery and data mining, pp. 355–359 (2000)
13. Lo, D., Khoo, S.-C., Liu, C.: Efficient mining of recurrent rules from a sequence database. In: DASFAA 2008, LNCS vol. 4947, pp. 67–83 (2008)
14. Lo, D., Khoo, S.C., Wong, L.: Non-redundant sequential rules—theory and algorithm. Inf. Syst. **34**(4), 438–453 (2009)
15. Masseglia, F., Cathala, F., Poncelet, P.: The PSP approach for mining sequential patterns. In: PKDD'98, Nantes, France, LNCS vol. 1510, pp. 176–184 (1998)
16. Pei, J., Han, J., Mortazavi-Asl, B., Wang, J., Pinto, H., Chen, Q., Hsu, M.C.: Mining sequential patterns by pattern-growth: the prefixspan approach. IEEE Trans. Knowl. Data Eng. **16**(11), 1424–1440 (2004)
17. Pei, J., Han, J., Mortazavi-Asl, B., Zhu, H.: Mining access patterns efficiently from web logs. In: Proceedings of the 4th Pacific-Asia Conference on Knowledge Discovery and Data Mining (PAKDD'00), Kyoto, Japan, pp. 396–407 (2000)
18. Spiliopoulou, M.: Managing interesting rules in sequence mining. In: Proceedings of the Third European Conference on Principles of Data Mining and Knowledge Discovery, Prague, Czech Republic, pp. 554–560 (1999)
19. Srikant, R., Agrawal, R.: Mining sequential patterns: generalizations and performance improvements. In: Proceedings of the 5th International Conference on Extending Database Technology: Advances in Database Technology, Avignon, France, LNCS, pp. 3–17 (1996)
20. UCI Machine Learning Repository. http://www.ics.uci.edu/mlearn/MLRepository.html
21. Van, T.T., Vo, B., Le, B.: Mining sequential rules based on prefix-tree. In: Proceedings of the 3rd Asian Conference on Intelligent Information and Database Systems, Daegu, Korea, pp. 147–156 (2011)
22. Zaki, M.J.: SPADE: an efficient algorithm for mining frequent sequences. Mach. Learn. **42**(1–2), 31–60 (2001)