Mining Top-K Sequential Rules

Philippe Fournier-Viger and Vincent S. Tseng

Department of Computer Science and Information Engineering
National Cheng Kung University
philippe.fv@gmail.com, tsengsm@mail.ncku.edu.tw

Abstract. Mining sequential rules requires specifying parameters that are often difficult to set (the minimal confidence and minimal support). Depending on the choice of these parameters, current algorithms can become very slow and generate an extremely large amount of results or generate too few results, omitting valuable information. This is a serious problem because in practice users have limited resources for analyzing the results and thus are often only interested in discovering a certain amount of results, and fine-tuning the parameters can be very time-consuming. In this paper, we address this problem by proposing TopSeqRules, an efficient algorithm for mining the top-k sequential rules from sequence databases, where *k* is the number of sequential rules to be found and is set by the user. Experimental results on real-life datasets show that the algorithm has excellent performance and scalability.

1 Introduction

Nowadays, huge amounts of sequential information are stored in databases (e.g. stock market data, biological data and customer data). Discovering patterns in such databases is important in many domains, as it provides a better understanding of the data. For example, in international trade, one could be interested in discovering temporal relations between the appreciations of currencies to make trade decisions. Various methods have been proposed for mining patterns in sequential databases such as mining repetitive patterns, trends and sequential patterns (see [1] for a survey). Among them, mining sequential patterns is probably the most popular set of techniques (e.g. [2, 3, 4]). It consists of finding subsequences appearing frequently in a database. However, knowing that a sequence appear frequently in a database is not sufficient for making prediction [5]. An alternative that addresses the problem of prediction is *sequential rule mining* [5-12]. A sequential rule indicates that if some item(s) occurred, some other item (s) are likely to occur with a given confidence or probability afterward. Sequential rule mining has many applications (e.g. stock market [7], weather observation [9], drought management [10] and e-learning [5, 6]).

Sequential rule mining algorithms have been developed for discovering rules in a single sequence (e.g. [9, 12]) or in multiple sequences (e.g. [5, 6, 7, 10, 11]). To mine sequential rules, users typically have to set two parameters: (1) a minimum support threshold and (2) a minimal confidence threshold. But one important question that has not been addressed in previous research is: "How can we choose appropriate values for these parameters if we don't have any background knowledge about the data-

J. Tang et al. (Eds.): ADMA 2011, Part II, LNAI 7121, pp. 180–194, 2011.

base?" It is an important question because if these parameters are set too high, few patterns are found and algorithms have to be rerun to find more patterns, and if parameters are set too low, algorithms become incredibly slow and generate an extremely large amount of results. In practice, to find appropriate values for these parameters, people generally successively try different values by guessing and executing the algorithms over and over until being satisfied by the results, which can be very time-consuming. However, in data mining, users are often only interested in discovering the "top" patterns in a database because they have limited resources for analyzing patterns that are found [13-16]. In this paper, we address this issue for the task of mining sequential rules in sequence databases. We propose *TopSeqRules*, an algorithm for mining only the top-k sequential rules, where *k* is a parameter set by the user. This allows the user to specify for example, that he wants to discover the top 500 rules. Although several top-k pattern mining algorithms have been designed for mining patterns like frequent itemsets (e.g. [13, 14, 16]) and sequential patterns (e.g. [15]), we are the first to address the problem for sequential rules. The rest of this paper is organized as follows. Section 2 reports related work and defines the problem of top-k sequential rule mining. Section 3 describes TopSeqRules, optimizations and extensions. Then, section 4 presents the evaluation. Finally, Section 5 presents the conclusion.

2 Problem Definition and Related Work

There exist several definitions of what is sequential rule mining [5-12] (see [5] for a literature review). In this paper, we use the definition of [6] for discovering sequential rules common to multiple sequences because it has two reported real applications [6] and because not many works have addressed the case of multiple sequences, despite that it has many potential applications. According to this definition, a *sequence database* (defined as in sequential pattern mining [2]) is a set of sequences $S=\{s_1, s_2 \ldots s_s\}$ and a set of items $I=\{i_1, i_2, \ldots i_t\}$ occurring in these sequences. A *sequence* is defined as an ordered list of *itemsets* (sets of items) $s_x=I_1, I_2, \ldots I_n$ such that $I_1, I_2, \ldots I_n \subseteq I$, and where each sequence is assigned a unique *sid* (sequence id). As an example, figure 1.a depicts a sequence database containing four sequences with sids *seq1*, *seq2*, *seq3* and *seq4*. In this example, each single letter represents an item. Items between curly brackets represent an itemset. For instance, the sequence *seq1* means that items *a* and *b* occurred at the same time, and were followed successively by *c*, *f*, *g* and *e*. A *sequential rule* $X \Rightarrow Y$ is defined as a relationship between two itemsets X, $Y \subseteq I$ such that $X \cap Y = \emptyset$ and $X, Y \neq \emptyset$. Note that X and Y are unordered. The interpretation of a rule $X \Rightarrow Y$ is that if the items of X occur in a sequence, the items in Y will occur afterward in the same sequence. Formally, a rule $X \Rightarrow Y$ is said to *occur* in a sequence $s_x=I_1, I_2 \ldots I_n$ if there exists an integer *u* such that $1 \leq u < n$, $X \subseteq \bigcup_{i=1}^{u} I_i$ and $Y \subseteq \bigcup_{i=u+1}^{n} I_i$. For example, the rule $\{a, b, c\} \Rightarrow \{e, f, g\}$ occurs in the sequence $\{a, b\}$, $\{c\}$, $\{f\}$, $\{g\}$, $\{e\}$, whereas the rule $\{a, b, f\} \Rightarrow \{c\}$ does not because item *c* does not occur after *f*. A rule $X \Rightarrow Y$ is said to be of size $v*w$ if $|X| = v$ and $|Y| = w$. For example, the rules $\{a, b, c\} \Rightarrow \{e, f, g\}$ and $\{a\} \Rightarrow \{e, f\}$ are of size 3*3 and 1*2 respectively.

Furthermore, a rule of size $f*g$ is said to be *larger than* another rule of size $h*i$ if $f > h$ and $g \geq i$, or if $f \geq h$ and $g > i$. For a given sequence database and a rule X⇒Y, the notation *sids*(X⇒Y) represents the *sids set* (the set of sequence ids) of the sequences where the rule occurs. For instance, *sids*({*a*}⇒{*b*}) = {*seq2*, *seq3*}. For an itemset X and a sequence database, the notation *sids*(X) denotes the sids set corresponding to sequences where all the items of X appears. For example, *sids*({*a*, *b*, *c*}) = {*seq1*, *seq2*}. For the sake of brevity, in the rest of this paper, curly brackets will be omitted when using the "*sids"* notation with itemsets containing a single item. Two interestingness measures are defined for sequential rules, which are similar to those used in association rule mining [17]. The *support* of a rule X⇒Y is defined as sup(X⇒Y) = |*sids*(X⇒Y)| / |S|. The *confidence* is defined as conf(X⇒Y) = |*sids*(X⇒Y)| / |*sids*(X)|. The *problem of mining sequential rules common to multiple sequences* is to find all valid rules in a sequence database [6]. A *valid rule* is a rule such that its support and confidence are respectively no less than user-defined thresholds *minsup* and *minconf*. For example, figure 1.b illustrates some valid rules found in the database shown in figure 1.a for *minsup* = 0.5 and *minconf* = 0.5. Moreover, a rule having a support higher or equal to *minsup* is said to be a *frequent rule*. Thus, by definition, valid rules are a subset of frequent rules.

a)

ID	Sequences
seq1	{*a*, *b*},{*c*},{*f*},{*g*},{*e*}
seq2	{*a*, *d*},{*c*},{*b*},{*a*, *b*, *e*, *f*}
seq3	{*a*},{*b*},{*f*},{*e*}
seq4	{*b*},{*f*, *g*}

b) →

ID	Rule	Support	Conf.
r1	{*a*, *b*, *c*}⇒{*e*}	0.5	1.0
r2	{*a*}⇒{*c*, *e*, *f*}	0.5	0.66
r3	{*a*, *b*}⇒{*e*, *f*}	0.5	1.0
r4	{*b*}⇒{*e*, *f*}	0.75	0.75
r5	{*a*}⇒{*e*, *f*}	0.75	1.0
r6	{*c*}⇒{*f*}	0.5	1.0
r7	{*a*}⇒{*b*}	0.5	0.66
…	…	…	…

Fig. 1. A sequence database (left) and some sequential rule found (right)

To define an algorithm for discovering the "top-k" sequential rules, we first need to define what a "top-k sequential rule" is. In frequent pattern mining, top-k pattern mining algorithms have been defined principally for mining frequent itemsets [13, 14, 16] and sequential patterns [15]. For discovering these types of patterns only the minimum support is generally used. Consequently, the problem of mining the top-k patterns for these types of patterns is defined as discovering the *k* patterns having the highest support [13- 16]. For sequential rule mining, however, the problem of mining the top-k sequential rules could be stated in different ways because two interestingness measures are used (the support and the confidence) instead of one. We thus see two possible definitions: (I) to discover the *k* rules having the highest support such that their confidence is higher than *minconf,* (II) to discover the *k* rules having the highest confidence such that their support is higher than *minsup*.

For this paper, we choose definition I over definition II because in practice the parameter *minconf* is much easier to set than *minsup* because *minconf* represents the minimum confidence that a user want in rules, while choosing an appropriate value for *minsup* depends solely on the characteristics of the database and it is impossible to

know a priori what is an appropriate value for *minsup*. However, later on, in section 3.4 we will explain how TopSeqRules could be adapted for definitions I. Based on definition II, we define the problem of mining *the top-k sequential rules* as follows:

Definition 1 (top-k sequential rule mining): To discover in a sequence database a set L containing k rules such that for each rule $r_m \in$ L, $conf(r_m) \geq minconf$, and there does not exist a rule $r_n \notin$ L such that $sup(r_n) > sup(r_m)$ and $conf(r_n) \geq minconf$.

To mine top-k patterns, all top-k pattern mining algorithms (e.g. [13-16]) follow a same general process, although they have several differences. The *general process for mining* top-k *patterns* from a database is the following. Initially, a top-k algorithm sets the minimum interestingness criterion (e.g. *minsup*) to the lowest possible value to ensure that all the top-k patterns will be found. Then, the algorithm starts searching for patterns by using a search strategy. As soon as a pattern is found, it is added to a list of patterns L ordered by the interestingness of patterns. The list is used to maintain the top-k patterns found until now. Once k patterns are found, the value for the minimum interestingness criterion is raised to the interestingness value of the least interesting pattern in L. Raising the minimum interestingness value is used to prune the search space when searching for more patterns. Thereafter, each time a pattern is found that meets the minimum interestingness criterion, the pattern is inserted in L, the pattern(s) in L not respecting the minimum interestingness criterion anymore are removed from L, and the minimum interestingness criterion is raised to the value of the least interesting pattern in L. The algorithm continues searching for more patterns until no pattern are found by the search strategy.

What distinguish top-k pattern mining algorithms are their data structures, input, output and search strategies to discover patterns. As any other data mining algorithms, a top-k algorithm needs to use appropriate data structures and search strategies to be efficient in both memory and execution time. But besides that, the efficiency of a top-k algorithm depends largely on how fast it can raise the minimum interestingness criterion (e.g. *minsup*) to prune the search space. To raise the support quickly, it is desirable that a top-k pattern mining algorithm uses strategies to find the most interesting patterns as early as possible. Hence, to design an efficient top-k sequential rule mining algorithm, several questions have to be addressed such as "Which search strategy and data structures should be used?", and "What optimizations can be applied?"

3 The TopSeqRules Algorithm

To answer this challenge, we propose TopSeqRules, a top-k algorithm based on the search strategy of RuleGrowth for generating valid rules [5]. RuleGrowth is the current best algorithm for mining sequential rules according to the definition of section 2. Its search strategy consists of first finding rules containing only two items and then to find larger rules by recursively growing the rules by scanning the sequences containing them to find single items that could expand their left or right parts. These two processes for expanding rules are named *left expansion* and *right expansion*. TopSeqRules integrates these processes with the *general process for mining* top-k *patterns*

described in section 2. Furthermore, to mine the top-k rules efficiently, it also add optimizations and the strategy of always trying to generate the most promising rules first, to try to prune the search space quickly by raising *minsup*.

Before presenting TopSeqRules, we introduce preliminary definitions and properties related to left/right expansions. A *left expansion* is the process of adding an item i to the left side of a rule X⇒Y to obtain a larger rule X∪{i}⇒Y. Similarly, a *right expansion* is the process of adding an item i to the right side of a rule X⇒Y to obtain a rule X⇒Y∪{i}. Left/right expansions have four important properties.

Property 1 (left expansion, effect on support): If an item i is added to the left side of a rule *r:*X⇒Y, the support of the resulting rule *r'*:X∪{i}⇒Y can only be lower or equal to sup(*r*). **Proof:** The support of *r* and *r'* are respectively |*sids*(X⇒Y)| / |S| and |*sids*(X∪{i}⇒Y)| / |S|. Since |*sids*(X⇒Y)| ≥ |*sids*(X∪{i}⇒Y)|, sup(*r*) ≥ sup(*r'*).

Property 2 (right expansion, effect on support): If an item i is added to the right side of a rule *r:*X⇒Y, the support of the resulting rule *r'*:X⇒Y∪{i} can only be lower or equal to sup(*r*). **Proof:** The support of *r* and *r'* are respectively |*sids*(X⇒Y)| / |S| and |*sids*(X⇒Y∪{i})| / |S|. Since |*sids*(X⇒Y)| ≥ |*sids*(X⇒Y∪{i})|, sup(*r*) ≥ sup(*r'*).

Properties 1 and 2 imply that the support is monotonic with respect to left/right expansions. In other words, performing any combinations of left/right expansions of a rule can only result in rules having a support that is lower or equal to the original rule. Therefore, all the frequent can be found by recursively performing expansions on frequent rules of size 1*1. Moreover, property 1 and 2 guarantee that expanding a rule having a support less than *minsup* will not result in a frequent rule. The confidence is not monotonic with respect to expansions, as next properties demonstrate.

Property 3 (left expansion, effect on confidence): If an item i is added to the left side of a rule *r*:X⇒Y, the confidence of the resulting rule *r'*: X∪{i}⇒Y can be lower, higher or equal to the confidence of *r*. **Proof:** The confidence of *r* and *r'* are respectively |*sids*(X⇒Y)| / |*sids*(X)| and |*sids*(X∪{i}⇒Y)| / |*sids*(X∪{i})|. Because |*sids*(X⇒Y)| ≥ |*sids*(X∪{i}⇒Y)| and |*sids*(X)| ≥ |*sids*(X∪{i})|, conf(*r*) can be lower, higher or equal to conf(*r'*).

Property 4 (right expansion, effect on confidence): If an item i is added to the right side of a rule *r:*X⇒Y, the confidence of the resulting rule *r'*: X⇒Y∪{i} is lower or equal to the confidence of *r*. **Proof:** The confidence of *r* and *r'* are respectively |*sids*(X⇒Y)| / |*sids*(X)| and |*sids*(X⇒Y∪{i})| / |*sids*(X)|. Since |*sids*(X⇒Y)| ≥ |*sids*(X⇒Y∪{i})|, conf(*r*) ≥ conf(*r'*).

TopSeqRules relies on sids sets to calculate the support and confidence of rules obtained by left or right expansions. Sids sets have two important properties.

Property 5 (sids set of a rule and its itemsets): For any sequential rule X⇒Y, *sids*(X⇒Y) ⊆ *sids*(X) ∩ *sids*(Y). **Proof:** A rule can only occur in a sequence if all items from its left and right parts appear in it.

Property 6 (sids set of a rule obtained by left/right expansion): For any sequential rule *r'* obtained by a left or right expansion of a rule *r*, the relationship *sids*(*r'*) ⊆

sids(*r*) holds. **Proof**. If the rule *r* does not occur in a sequence, the rule *r'* also cannot. Therefore, the sids set of *r'* must be a subset of the sids set of *r*.

3.1 The Algorithm

TopSeqRules takes as input a sequence database *S*, a number *k* of rules that the user wants to discover, and the *minconf* threshold. The algorithm uses three main internal variables. The first one is *minsup,* wich is initially set to 0 and is raised dynamically as soon as *k* rules are found, as it will be explained. The second variable is a set named L to keep the top-k rules found until now that have a support and confidence higher or equals to *minsup* and *minconf*. The third variable is a set named R to store the rules that should be expanded to have a chance of finding more valid rules.

The Main Procedure. The main procedure of TopSeqRules is shown in figure 2. The algorithm first scans the database once to calculate *sids*(*c*) for each item *c*. Then, the algorithm generates all valid rules of size 1*1. This is done by taking each pair of items *i, j*, where *i* and *j* each have at least *minsup*×|S| sids (if this condition is not met, no rule having at least the minimum support can be created with *i* and *j*). The algorithm then scans sequences in *sids*(*i*) ∩ *sids*(*j*) to calculate *sids*($i \Rightarrow j$) and *sids*($j \Rightarrow i$), the sids of sequences where the rule $\{i\} \Rightarrow \{j\}$ and $\{j\} \Rightarrow \{i\}$ occur, respectively (because of property 5). After this, the support of the rule $\{i\} \Rightarrow \{j\}$ is obtained by dividing |*sids*($i \Rightarrow j$)| by |S|. For each rule $\{i\} \Rightarrow \{j\}$ or $\{j\} \Rightarrow \{i\}$ that is valid, the procedure SAVE is called with the rule and L as parameters so that the rule is recorded in the set L of the current top-k rules found. Also, each rule $\{i\} \Rightarrow \{j\}$ or $\{j\} \Rightarrow \{i\}$ that is frequent is added to the set R, to be later considered for expansion.

After that, a loop is performed to recursively select the rule *r* with the highest support in R such that *sup*(*r*) ≥ *minsup* and expand it. The idea behind this loop is to always expand the rule from R having the highest support first because it is more likely to generate rules having a high support and thus to allow to raise *minsup* more quickly for pruning the search space. The loop terminates when there is no more rule in R having a support higher or equal to *minsup*. For expanding a rule, a flag *expandLR* indicates if the rule should be left and right expanded by calling the procedure EXPAND-L and EXPAND-R or just left expanded by calling EXPAND-L. For all rules of size 1*1, this flag is set to true. The utility of this flag for larger rules will be explained later.

The Save Procedure. The role of SAVE (figure 3) is to raise *minsup* and update the list L when a new valid rule *r* is found. The first step of SAVE is to add the rule *r* to L. Then, if L contains more than *k* rules and the support is higher than *minsup*, rules from L that have exactly the support equal to *minsup* can be removed until only *k* rules are kept. Finally, *minsup* is raised to the support of the rule in L having the lowest support. By this simple scheme, the top-k rules found are maintained in L.

Now that we have described how rules of size 1*1 are generated and the mechanism for maintaining the top-k rules in L, we explain how rules of size 1*1 are expanded to find larger rules. Without loss of generality, we can ignore the top-k aspect for the explanation and consider the problem of generating all valid rules. To recursively expand rules and find all valid rules starting from rules of size 1*1, a few problems had to be solved.

Problem 1: How can we guarantee that all valid rules are found by recursively performing left/right expansions starting from rules of size 1*1? The answer is found in properties 1 and 2, which states that the support of a rule is monotonic with respect to left/right expansions. This implies that all rules can be discovered by recursively performing expansions starting from frequent rules of size 1*1. Moreover, these properties imply that infrequent rules should not be expanded because they will not lead to valid rules. However, no similar pruning can be done for the confidence because it is not monotonic with respect to left expansion (property 3).

Problem 2: How we can guarantee that no rules are found twice by recursively making left/right expansions? To guarantee this, two sub-problems had to be solved. First, if we grow rules by performing expansions recursively, some rules can be found by different combinations of left/right expansions. For example, consider the rule $\{a, b\} \Rightarrow \{c, d\}$. By performing, a left and then a right expansion of $\{a\} \Rightarrow \{c\}$, one can obtain the rule $\{a, b\} \Rightarrow \{c, d\}$. But this rule can also be obtained by performing a right and then a left expansion of $\{a\} \Rightarrow \{c\}$. A simple solution to avoid this problem is to forbid performing a right expansion after a left expansion but to allow performing a left expansion after a right expansion. An alternative solution is to not allow a left expansion after a right expansion.

TOPSEQRULES (S, *k, minconf*)
1. R := Ø. L := Ø. *minsup* := 0.
2. **Scan** the database S once. Record the sids set of each item *c* in a variable *sids*(c).
3. **FOR** each pairs of items *i, j* such that |*sids(i)*|≥ *minsup* and ∩ |*sids(j)*|≥ *minsup:*
4. *sids(i⇒j)* := Ø. *sids(j⇒i)* := Ø.
5. **FOR** each sid *s* ∈ (*sids(i)* ∩ *sids(j)*:
6. **IF** *i* occurs before *j* in *s* **THEN** *sids(i⇒j)* := *sids(i⇒j)* ∪ {*s*}.
7. **IF** *j* occurs before *i* in *s* **THEN** *sids(j⇒i)* := *sids(j⇒ i)* ∪ {*s*}.
8. **END FOR**
9. **IF** |*sids(i⇒j)*| / |S| ≥ *minsup* **THEN**
10. *conf*({i}⇒{j}) := |*sids(i⇒j)*| / | *sids*(*i*).
11. **IF** *conf*({i}⇒{j}) ≥ *minconf* **THEN SAVE**({i}⇒{j}, L, *k, minsup*).
12. **Set flag** expandLR **of** {i}⇒{j}**to true**.
13. R := R∪{{i}⇒{j}}.
14. **END IF**
… [lines 9 to 14 are repeated here with *i* and *j* swapped] …
15. **END FOR**
16. **WHILE** ∃r ∈ R **AND** *sup*(r) ≥ *minsup* **DO**
17. **Select** the rule *rule* having the highest support in R
18. **IF** *rule*.expandLR = true **THEN**
19. **EXPAND-L**(*rule*, L, R, *k*, *minsup, minconf*).
20. **EXPAND-R**(*rule*, L, R, *k*, *minsup, minconf*).
21. **ELSE EXPAND-R**(*rule*, L, R, *k*, *minsup, minconf*).
22. **REMOVE** *rule* from R. **REMOVE** from R all rules r ∈ R | *sup*(r) <*minsup*.
23. **END WHILE**

Fig. 2. The TopSeqRules algorithm

The second sub-problem is that rules can be found several times by performing left/right expansions with different items. For example, consider the rule $\{b, c\} \Rightarrow \{d\}$.

```
SAVE(r, R, k, minsup)
1.  L := L∪{r}.
2.  IF |L| ≥ k THEN
3.      IF sup(r) > minsup THEN
4.          WHILE |L| > k AND ∃s ∈ L | sup(s) = minsup
5.              REMOVE s from L.
6.      END IF
7.      Set minsup to the lowest support of rules in L.
8.  END IF
```

Fig. 3. The SAVE procedure

A left expansion of $\{b\}\Rightarrow\{d\}$ with item c can result in the rule $\{b, c\}\Rightarrow\{d\}$. But that latter rule can also be found by performing a left expansion of $\{c\}\Rightarrow\{d\}$ with b. To solve this problem, a solution is to only add an item to an itemset of a rule if the item is greater than each item in the itemset according to the lexicographic ordering. In the previous example, this would mean that item c would be added to the left itemset of $\{b\}\Rightarrow\{d\}$. But b would not be added to the left itemset of $\{c\}\Rightarrow\{d\}$ because b is not greater than c. By using this strategy and the previous one, no rules are found twice. We now explain how EXPAND-L and EXPAND-R have been implemented based on these strategies.

The EXPAND-R Procedure. The procedure EXPAND-R (cf. figure 4) takes as parameters a rule I⇒J to be expanded, L, R, k, *minsup* and *minconf*. To expand I⇒J, EXPAND-R has to identify items that can expand the rule I⇒J to produce a valid rule. By exploiting the fact that any valid rule is a frequent rule, this problem is decomposed into two sub-problems, which are (1) determining items that can expand a rule I⇒J to produce a frequent rule and (2) assessing if a frequent rule obtained by an expansion is valid. The first sub-problem is solved as follows. To identify items that can expand a rule I⇒J and produce a frequent rule, the algorithm scans each sequence *sid* from *sids*(I∩J). During this scan, for each item $c\notin$ I appearing in sequence *sid* after I, the algorithm adds *sid* to a variable *sids*(I⇒J∪{c}) if c is lexically larger than all items in J (this latter condition is to ensure that no duplicated rules will be generated, as explained). When the scan is completed, for each item c such that $|sids(I\Rightarrow J\cup\{c\})| / |S| \geq minsup$, the rule I⇒J∪{$c$} is deemed frequent and is added to the set R so that it will be later considered for expansion. Note that the flag *expandLR* of each such rule is set to *false* so that each generated rule will only be considered for right expansions (to make sure that no rules are found twice by different combinations of left/right expansions, as explained). Finally, the confidence of each frequent rule I⇒J∪{c} is calculated to see if the rule is valid, by dividing $|sids(I\Rightarrow J\cup\{c\})|$ by $|sids(I)|$, the value *sids*(I) having already been calculated for I⇒J. If the confidence of I⇒J∪{c} is no less than *minconf,* then the rule is valid and the procedure SAVE is called to add the rule to L, the list of the current top-k rules.

The EXPAND-L Procedure. The procedure EXPAND-L (cf. figure 5) takes as parameters a rule I⇒J to be expanded, L, R, k, *minsup* and *minconf*. This procedure is similar to EXPAND-R. The only extra step that is performed compared to EXPAND-R is that for each rule I∪{c}⇒J obtained by the expansion of I⇒J with an item c, the value *sids*(I∪{c}) necessary for calculating the confidence is obtained by intersecting *sids*(I) with *sids*(c). The value sids(c) is known from the initial database scan.

3.2 Implementing TopSeqRules Efficiently

We implemented TopSeqRules in Java. We used two optimizations, which greatly improve TopSeqRules' execution time in our experiments (cf. section 4).

Optimization 1: Implementing L and R with Efficient Data Structures. TopSeqRules performs three operations on L, which are insertion, deletion and finding the rule having the lowest support. The three same operations are performed on R plus finding the rule having the highest support (to select the most promising rules from R). Because these operations are performed constantly by TopSeqRules, it is important to implement L and R with data structures that support performing these operations efficiently. To address this issue, we implement L with a *Fibonacci heap* sorted by the support of the rules. It has an amortized time cost of $O(1)$ for insertion and obtaining the minimum, and $O(log(n))$ for deletion [18]. For R, we used a *red-black tree* because we also need the operation of finding the maximum. A red-black tree guarantees a $O(log(n))$ worst-case time cost for the four operations [18] .

Optimization 2: Merging Database Scans for the Left/Right Expansions of a Rule. The second optimization reduces the number of database scans. Recall that EXPAND-L and EXPAND-R are both applied to each rule I⇒J having the flag *expandLR* set to true. Performing EXPAND-L and EXPAND-R each requires to scan each sequence from *sids*(I⇒J) once. A simple optimization is to combine the database

EXPAND-R(I⇒J, L, R, *k*, *minsup, minconf*)
1. **FOR** each *sid* ∈ *sids*(I⇒J), scan the sequence *sid*. For each item *c* appearing in sequence *sid* that is lexically larger than all items in J and appear after I, record *sid* in a variable *sids*(I⇒J∪{*c*}).
2. **FOR** each item *c* such that |*sids*(I⇒J∪{*c*})| ≥ minsup×|S| :
3. **Set flag** expandLR **of** I⇒J∪{c} **to false**.
4. R := R∪{I⇒J∪{c}}.
5. **IF** | *sids* (I⇒J∪{*c*})| / | *sids* (I)| ≥ *minconf* **THEN** SAVE(I⇒J∪{*c*}, L, *k, minsup*).
6. **END FOR**
7. **END FOR**

Fig. 4. The EXPAND-R procedure

EXPAND-L(I⇒J, L, R, *k*, *minsup, minconf*)
1. **FOR** each *sid* ∈ *sids*(I⇒J), scan the sequence *sid*. For each item c∉J appearing in sequence *sid* that is lexically larger than all items in I and appear before J, record *sid* in a variable *sids*(I∪{c}⇒J)
2. **FOR** each item *c* such that |*sids* (I∪{*c*}⇒J)| / |S| ≥ *minsup* :
3. **Set flag** expandLR **of** I⇒J∪{c} **to true**.
4. *sids*(I∪{*c*}) := Ø.
5. **FOR** each *sid* ∈ *sids*(I) such that *sid* ∈ *sids*(*c*), *sids*(I∪{*c*}):= *sids*(I∪{*c*}) ∪{*sid*}.
6. SAVE(I∪{*c*}⇒J, L, *k, minsup*).
7. **IF** |*sids*(I∪{*c*}⇒J)| / |*sids*(I∪{*c*})| ≥ *minconf* **THEN** R := R∪{I∪{*c*}⇒J}.
8. **END FOR**

Fig. 5. The EXPAND-L procedure

scans of EXPAND-L and EXPAND-R so that they use the same database scan for identifying left and right expansions, when the flag *expandLR* is set to true.

3.3 Extensions

The TopSeqRules algorithm can be extended in several ways. We list two.

Extension 1: Using a different definition of what is a top-k sequential rule. In section 2, we presented two possible definitions of what is a top-k sequential rule, and selected definition I for presenting the algorithm. However, TopSeqRules could easily be modified so that the support is fixed instead of the confidence for finding the top-k rules. This would result in a mining algorithm for definition II. However, the resulting algorithm would be inefficient unless the support is set high, because the confidence could not be raised dynamically to prune the search space because the confidence is not monotonic with respect to left/right expansions (cf. section 3).

Extension2: Using different interestingness measures. This paper considered the confidence and support because they are the standard measures for sequential rules. But other measures could be used. For example, more than twenty interestingness measures have been proposed for association rule mining [17]. Many of those could be adapted for sequential rule mining and integrated in the TopSeqRules algorithm because the calculation is done similarly to the calculation of the confidence and support. For example, the *lift* [17] could be adapted for sequential rules as *lift*(I⇒J) = *sup*(I⇒J)/ (sup(I) × sup(J)) for a sequential rule I⇒J. Compared to the confidence, using the *lift* would just require to calculate *sup*(J) for each rule in addition to *sup*(I). However, to be able to prune the search space, it is necessary that the "top-k" condition is defined on a monotonic interestingness measure like the support. For example, with just a few modifications, one could mine the *k* rules having a support higher or equal to *minsup* such that their *lift* is no less than a *minlift* threshold.

4 Evaluation

We evaluated TopSeqRules on a notebook with a 2.53 Ghz processor, Windows XP and 1 GB of free RAM. Experiments were carried on three real-life datasets representing three types of data. Table 1 summarizes the characteristics of the datasets. *BMSWebview1* was downloaded from http://fimi.ua.ac.be/data/. *Sign* was downloaded from http://cs-people.bu.edu/panagpap/Research/asl_mining.htm. S*nake* was obtained from the authors of [4].

Table 1. Datasets characteristics

Datasets	|S|	|I|	Avg. item count / sequence	Type of data
BMSWebView1	59601	497	2.5 (σ = 4.85)	click-stream from web store
Sign	730	310	93.39 (σ = 4.59)	language utterances
Snake	163	20	60.61 5 (σ = 0.89)	protein sequence

4.1 Influence of *k*

The first experiment was done to evaluate the influence of *k* on the execution time and the memory consumption. We ran TopSeqRules with *minconf = 0.3* on each dataset while varying *k* from 500 to 5000. Results are shown in figure 6. Our first observation is that the execution time and the maximum memory consumption is excellent for these real-life datasets (in the worst case, the algorithm took a little less than 1 minute to terminate and used about 1 gigabyte of memory). Furthermore, it can be seen that the algorithm performance and memory usage grows linearly with *k*. The only exception is for *k*=3000 to *k*=4500 for the *BMS-WebView1* dataset where the memory usage remains the same and the execution time increases more quickly. We found that this is not caused by the algorithm design. But it is caused by the Java garbage collection mechanism overhead when the memory usage is close to the 1GB limit set for the experiment.

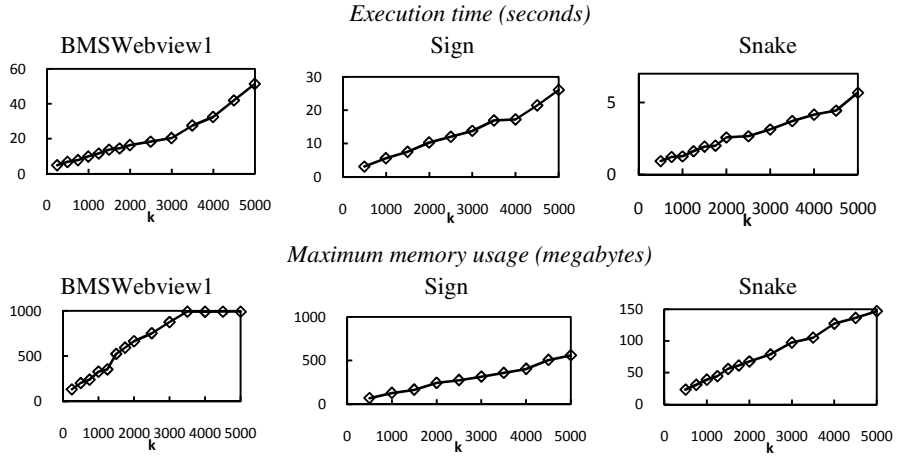

Fig. 6. Results of varying *k*

4.2 Influence of *minconf*

In a second experiment, we tested the influence of *minconf* on the execution time and memory consumption. We ran TopSeqRules on the same datasets with *k* = 1000 while varying *minconf* to observe its influence on the execution time and the memory usage. Results are shown in figure 7. It can be seen that as the confidence increases, the execution time and memory usage increase in an exponential manner. The reason is that setting the confidence higher means that the algorithm has to generate more rules to be able to raise the minimum support threshold when searching for the top-k sequential rules. Nevertheless, the algorithm ran successfully with high *minconf* thresholds in our experiment within the memory limit (up to 0.7, 0.85 and 0.99, respectively for *BMSWebview1, Sign* and *Snake*).

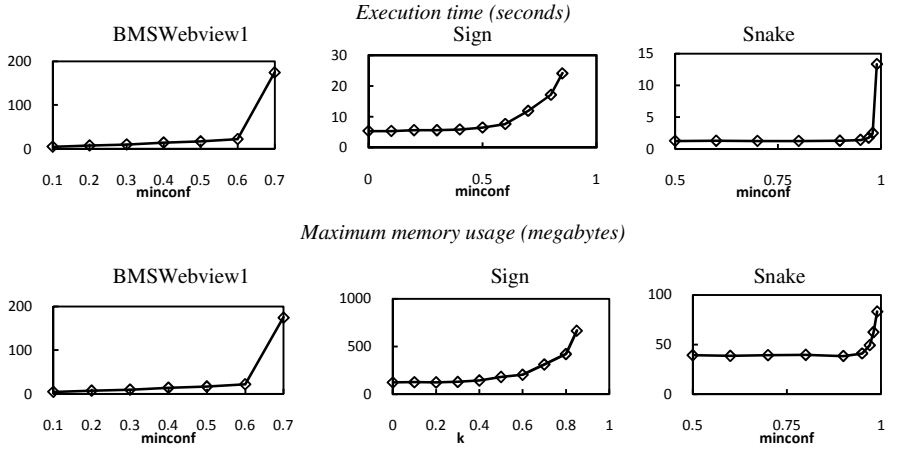

Fig. 7. Results of varying *minconf*

4.3 Influence of |S|

In a third experiment, we tested the scalability with respect to the number of sequences, by applying TopSeqRules with *minconf*=0.3 and *k*=1000 on 50% to 100 % of the sequences of each dataset. Figure 8 shows the results. It can be seen that the execution time and memory usage increase slowly when the number of sequences increases. This is because the performance of TopSeqRules depends more on the number of rules generated and stored in R than the number of sequences, and the number of rules generated remains more or less the same when the database size increase. This shows that TopSeqRules has an excellent scalability.

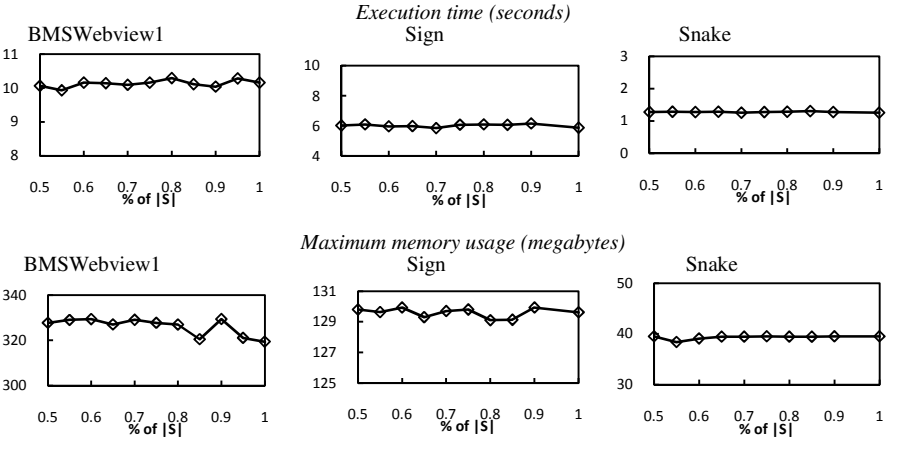

Fig. 8. Results of varying the database size

4.4 Performance Comparison

In a fourth experiment, we compared the performance of TopSeqRules with RuleGrowth [6] (also implemented in Java), the state of the art algorithm for the problem of mining sequential rules presented in section 2. To compare their performance, we first considered the scenario where the user would choose the optimal parameters for RuleGrowth to produce the same amount of result produced by TopSeqRules. For this scenario, we ran TopSeqRules on the three datasets with the parameters used in section 4.1. We then ran RuleGrowth with *minsup* equals to the lowest support for the rules found by TopSeqRules, for each *k* and each dataset. Results are shown in figure 10. It can be observed that the execution time of TopSeqRules is generally close to RuleGrowth's execution time except for *BMSWebView1* where the gap is larger. But the main difference between the performance of TopSeqRules and RuleGrowth is in the memory usage. TopSeqRules uses more memory because it keeps the set R of rules to be expanded into memory. For this reason, as *k* is set to larger value, the memory requirement of TopSeqRules increases. These results are excellent considering that the parameters of RuleGrowth were chosen optimally, which is rarely the case in real-life if the user has no a priori knowledge of the database. If the parameters of RuleGrowth are not chosen optimally, it can run much slower than TopSeqRules, or generate too few or too many results. For example, consider the case where the user wants to discover the top 1000 rules from a database and do not want to find more than 2000 rules. To find this amount of rules, the user needs to choose *minsup* from a very narrow range of values (shown in Table 2 for each dataset). For example, for *BMSWebView1*, the range of *minsup* values that will satisfy the user is 0.0011 to 0.0009. This means that a user having no a priori knowledge of the database has only a 0.02 % chance of selecting a *minsup* value that will make him satisfied. If the users choose a higher *minsup*, not enough rules will be found, and if *minsup* is set lower, too many rules will be found and the algorithm may become slow. This clearly shows the benefits of using TopSeqRules.

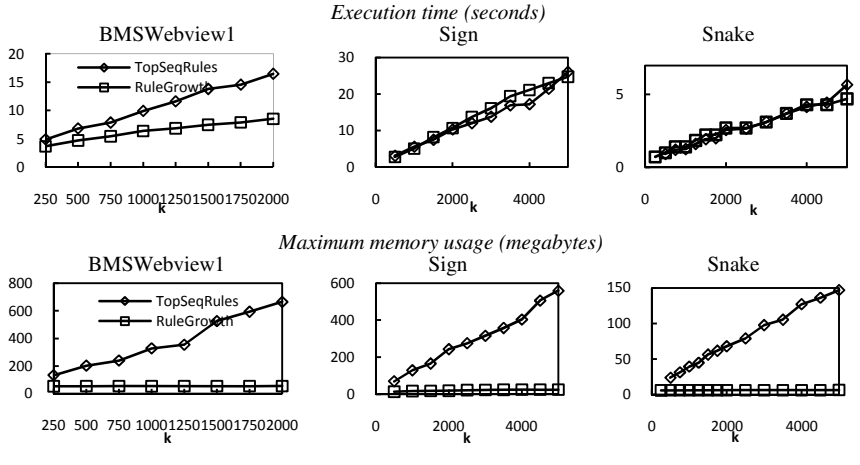

Fig. 9. Performance comparison for optimal parameters selection

Table 2. Interval of *minsup* values to find the top 1000 to 2000 rules for each dataset

Datasets	*minsup* for k=1000	*minsup* for k=2000	interval size
BMSWebView1	0.0011	0.0009	0.0002
Sign	0.420	0.384	0.036
Snake	0.960	0.944	0.016

4.5 Influence of Optimizations and of Expanding the Most Promising Rules First

Lastly, we evaluated the benefit of using the optimizations and of expanding the most promising rules first. Due to space limitations, we do not show the results as charts. We observed that *optimization 1* (data structures) and *optimization 2* (merging database scans) each reduce the execution time by about 20% to 40 % on all datasets. For the strategy of expanding the most promising rules first with the set R, we found that if this strategy is deactivated, the algorithm cannot terminate within 1 hour on all datasets because the algorithm cannot prune the search space efficiently. This is because in the worst case a top-k algorithm has to explore the whole search space before finding the top-k rules. If there are d items, the number of rules to consider is in the worst case $\sum_{k=1}^{d-1}\left[\binom{d}{k}\times\sum_{j=1}^{d-k}\binom{d-k}{j}\right]=3^d-2^d+1$, which is exponential. For this reason, it is necessary to use the set R to expand the most promising rules first to try to raise *minsup* as fast as possible and prune the search space.

5 Conclusion

Mining sequential rules requires specifying parameters (e.g. the minimal confidence and the minimal support) that are difficult to set. To address this problem, we propose an efficient algorithm named TopSeqRules that let the user specify *k,* the amount of sequential rules to be output. Experimental results with real-life datasets show that the algorithm has excellent scalability, that its execution time linearly increases with the parameter *k* and that the algorithm had no problem running in reasonable time and memory limits for *k* values of up to 5000 for all datasets. Results also show that when parameters are chosen optimally, RuleGrowth can be slightly faster than TopSeqRules. However, if *minsup* is set higher than a narrow range of values, RuleGrowth generates too few results and if it is set lower, it generates too many results and can become much slower. This clearly shows the benefit of using TopSeqRules when the user has no a priori knowledge about a sequence database.

References

1. Laxman, S., Sastry, P.: A survey of temporal data mining. Sadhana 3, 173–198 (2006)
2. Agrawal, R., Srikant, R.: Mining Sequential Patterns. In: Proc. ICDE 1995, pp. 3–14 (1995)

3. Zaki, M.J.: SPADE: An Efficient Algorithm for Mining Frequent Sequences. Machine Learning 42(1/2), 31–60 (2001)
4. Jonassen, I., Collins, J.F., Higgins, D.G.: Finding flexible patterns in unaligned protein sequences. Protein Science 4(8), 1587–1595 (1995)
5. Fournier-Viger, P., Nkambou, R., Tseng, V.S.: RuleGrowth: Mining Sequential Rules Common to Several Sequences by Pattern-Growth. In: Proc. SAC 2011, pp. 954–959 (2011)
6. Fournier-Viger, P., Faghihi, U., Nkambou, R., Mephu Nguifo, E.: CMRules: Mining Sequential Rules Common to Several Sequences. Knowledge-based Systems 25(1), 63–76 (2012)
7. Das, G., Lin, K.-I., Mannila, H., Renganathan, G., Smyth, P.: Rule Discovery from Time Series. In: Proc. ACM SIGKDD 1998, pp. 16–22 (1998)
8. Deogun, J.S., Jiang, L.: Prediction Mining – An Approach to Mining Association Rules for Prediction. In: Ślęzak, D., Yao, J., Peters, J.F., Ziarko, W.P., Hu, X. (eds.) RSFDGrC 2005, Part II. LNCS (LNAI), vol. 3642, pp. 98–108. Springer, Heidelberg (2005)
9. Hamilton, H.J., Karimi, K.: The TIMERS II Algorithm for the Discovery of Causality. In: Ho, T.-B., Cheung, D., Liu, H. (eds.) PAKDD 2005. LNCS (LNAI), vol. 3518, pp. 744–750. Springer, Heidelberg (2005)
10. Harms, S.K., Deogun, J.S., Tadesse, T.: Discovering Sequential Association Rules with Constraints and Time Lags in Multiple Sequences. In: Hacid, M.-S., Raś, Z.W., Zighed, D.A., Kodratoff, Y. (eds.) ISMIS 2002. LNCS (LNAI), vol. 2366, pp. 432–441. Springer, Heidelberg (2002)
11. Lo, D., Khoo, S.-C., Wong, L.: Non-redundant sequential rules – Theory and algorithm. Information Systems 34(4-5), 438–453 (2009)
12. Mannila, H., Toivonen, H., Verkano, A.I.: Discovery of frequent episodes in event sequences. Data Mining and Knowledge Discovery 1(1), 259–289 (1997)
13. Wang, J., Han, J., Lu, Y., Tzvetkov, P.: TFP: An Efficient Algorithm for Mining Top-k Frequent Closed Itemsets. IEEE TKDE 17(5), 652–664 (2005)
14. Pietracaprina, A., Vandin, F.: Efficient Incremental Mining of top-K Frequent Closed Itemsets. In: Corruble, V., Takeda, M., Suzuki, E. (eds.) DS 2007. LNCS (LNAI), vol. 4755, pp. 275–280. Springer, Heidelberg (2007)
15. Tzvetkov, P., Yan, X., Han, J.: TSP: Mining top-k closed sequential patterns. Knowledge and Information Systems 7(4), 438–457 (2005)
16. Chuang, K.-T., Huang, J.-L., Chen, M.-S.: Mining top-k frequent patterns in the presence of the memory constraint. VLDB 17(5), 1321–1344 (2008)
17. Tan, P.-N., Kumar, V., Srivastava, J.: Selecting the right objective measure for association analysis. Information Systems 29(4), 293–313 (2004)
18. Cormen, T.H., Leiserson, C.E., Rivest, R.L., Stein, C.: Introduction to Algorithms, 3rd edn. MIT Press (2009)