Mining Partially-Ordered Sequential Rules Common to Multiple Sequences

Philippe Fournier-Viger, Cheng-Wei Wu, Vincent S. Tseng, Longbing Cao and Roger Nkambou

Abstract—Sequential rule mining is an important data mining problem with multiple applications. An important limitation of algorithms for mining sequential rules common to multiple sequences is that rules are very specific and therefore many similar rules may represent the same situation. This can cause three major problems: (1) similar rules can be rated quite differently, (2) rules may not be found because they are individually considered uninteresting, and (3) rules that are too specific are less likely to be used for making predictions. To address these issues, we explore the idea of mining "partially-ordered sequential rules" (POSR), a more general form of sequential rules such that items in the antecedent and the consequent of each rule are unordered. To mine POSR, we propose the RuleGrowth algorithm, which is efficient and easily extendable. In particular, we present an extension (TRuleGrowth) that accepts a sliding-window constraint to find rules occurring within a maximum amount of time. A performance study with four real-life datasets show that RuleGrowth and TRuleGrowth have excellent performance and scalability compared to baseline algorithms and that the number of rules discovered can be several orders of magnitude smaller when the sliding-window constraint is applied. Furthermore, we also report results from a real application showing that POSR can provide a much higher prediction accuracy than regular sequential rules for sequence prediction.

Index Terms— sequential rules, sequential patterns, temporal patterns, pattern mining, sequence, data mining.

1 Introduction

Sequential pattern mining is an important data mining task with wide applications. It consists of discovering subsequences that are common to multiple sequences. Several algorithms have been proposed for this task such as GSP [2], PrefixSpan [14], SPADE [17] and CM-SPADE [18]. However, *sequential patterns* found by these algorithms are often misleading for the user. The reason is that patterns are found solely on the basis of their *support* (the percentage of sequences in which they occur). For instance, consider the sequential pattern {*Vivaldi*}, {*Handel*}, {*Berlioz*} meaning that customer(s) bought the music of Vivaldi, Handel and Berlioz in that order. This sequential pattern is said to have a support of 50 % because it appears in sequences 1, 2 and 4 of the following sequence database containing six sequences.

1: {*Vivaldi*}, {*Mozart*}, {*Handel*}, {*Berlioz*}
2: {*Mozart*}, {Bach},{*Paganini*}, {*Vivaldi*}, {*Handel*}, {*Berlioz*}
3: {*Handel*}, {*Vivaldi*}, {*Mozart*}, {*Ravel*}, {*Berlioz*}
4: {*Vivaldi*}, {*Mozart*}, {*Handel*}, {*Bach*}, {*Berlioz*}
5: {*Mozart*}, {*Bach*}, {*Vivaldi*}, {*Handel*}
6: {*Vivaldi*}, {*Handel*}, {*Mozart*}, {*Bach*}

However, this pattern is misleading because despite that it appears in 50 % of the sequences, there are also two sequences where {*Vivaldi*}, {*Handel*} are not followed by {*Berlioz*} (sequences 5 and 6). Therefore, if someone had to take decisions on the basis of this pattern, it could lead to taking wrong decisions. A solution to this problem would be to add a measure of the confidence or probability that a pattern will be followed. But adding this information to sequential patterns is not straightforward because they can contain multiple items and sequential pattern mining algorithms have just not been designed for that. An alternative that considers the confidence of a sequential pattern is *sequential rule mining* [4], [5], [8], [9], [12], [13], [16], [17], [19], [22]. A *sequential rule* (also called episode rule, temporal rule or prediction rule) indicates that if some event(s) occur, some other event(s) are likely to follow with a given confidence or probability. Sequential rule mining has been applied in several domains such as drought management [5], [9], stock market analysis [4], [16], weather observation [8], reverse engineering [29], e-learning [7], [15], and e-commerce [22]. Algorithms for sequential rule mining are designed to either discover rules appearing in a single sequence [5], [8], [13], [16], across sequences [4], [9], [29] or common to multiple sequences [12], [17], [19], [22], [25]. In this article, we are interested by the task of *mining sequential rules common to multiple sequences*, which is analogous to sequential pattern mining, and is also applied on sequence databases. It consists of finding rules of the form X⇒Y in a sequence database such that X and Y are sequential patterns [12], [17], [19], [22], [25]. Each rule is found on the basis of its *support* (the percentage of sequences that contains the rule) and its *confidence* (the probability that the sequential pattern Y will appear after X)[1]. Those rules are interpreted

- *Philippe Fournier-Viger is with the Department of Computer Science, University of Moncton, Moncton, Canada. E-mail: philippe.fv@gmail.com*
- *Cheng-Wei Wu and Vincent S. Tseng are with the Department of Computer Science and Information Engineering, National Cheng Kung University, Tainan City, Taiwan. E-mails: tsengsm@mail.ncku.edu.tw, silvemoonfox@idb.csie.ncku.edu.tw*
- *L. Cao is with the University of Technology Sydney, Sidney, Australia. E-mail: lbcao@it.uts.edu.au*
- *R. Nkambou is with the Department of Computer Science, University of Quebec at Montreal, Montreal, Canada. E-mail: nkambou.roger@uqam.ca*

Manuscript received 13 December 2014.

[1] Note that some algorithms use variations of these measures and also additional constraints.

as "if X appears, Y is likely to appear with a given confidence afterward". An example is the following rule: {*Vivaldi*}, {*Mozart*}, {*Handel*} ⇒ {*Berlioz*}. It means that customer(s) who bought the music of Vivaldi, Mozart and Handel in that order, have then bought the music of Berlioz. This rule has a support of 33% because it is found in two sequences (sequences 1 and 4) out of six sequences. Moreover, the rule is said to have a confidence of 100% because in each sequence where {*Vivaldi*}, {*Mozart*}, {*Handel*} appears in that order, it is followed by {*Berlioz*}. Mining such rules can be useful to make recommendations, predictions or to analyze customers' behavior. Besides, there are many other applications such as:

- **Web traversal patterns.** Sequential rules can be mined in sequences of webpages visited by users, to make recommendations [30].
- **Educational data.** For example, in e-learning, sequential rules have been mined to understand and predict the behavior of learners [6], and to discover patterns common to several learners' solutions [15].
- **Bioinformatics.** In this field, many different kind of sequential data need to be analyzed (e.g. protein sequences, DNA, microarray data, etc.). Sequential rule mining can be applied for example to discover sequential relationships between gene expressions of various patients using data from microarray experiments (e.g. [26], [28]).

1.1 Problems with Current Definition

We note, however, three important problems with the definition of a sequential rule as a relationship between two sequential patterns:

1) Rules may have many variations with different item ordering. Because sequential patterns specify a strict ordering between items, there might be several rules with the same items but a different ordering. For example, there are 23 variations of {*Vivaldi*}, {*Mozart*}, {*Handel*} ⇒ {*Berlioz*} with the same items ordered differently such as the following rules denoted as R1, R2, … R6:

R1: {*Vivaldi*}, {*Mozart*}, {*Handel*} ⇒ {*Berlioz*},
R2: {*Mozart*}, {*Vivaldi*}, {*Handel*} ⇒ {*Berlioz*},
R3: {*Handel*}, {*Vivaldi*}, {*Mozart*} ⇒ {*Berlioz*},
R4: {*Handel*, *Vivaldi*}, {*Mozart*} ⇒ {*Berlioz*},
R5: {*Handel*}, {*Vivaldi*, *Mozart*} ⇒ {*Berlioz*},
R6: {*Handel*, *Vivaldi*, *Mozart*} ⇒ {*Berlioz*}.

But all these variations describe the same situation (customers who bought music from Vivaldi, Mozart and Handel in any order, then bought music from Berlioz).

2) Rules and their variations may have important differences in how they are rated by the algorithms. For example, rules R1, R2 and R3 respectively have support/confidence of 33%/100%, 16%/50% and 16%/100%, and R4, R5 and R6 do not appear in the database. These differences in how variations of the same rules are rated can give a wrong impression of the sequential relationships contained in the database to the user. In fact, if all the variations of the same rule were taken as a whole, their support and confidence could be much higher. For example, none of the previous rules has a support higher than 33 %. But taken as a whole they appear in four sequences out of six (66 %).

3) Rules are less likely to be useful. Because rules are very specific, each rule is less likely to match with a new sequence to make predictions. For example, consider that a new customer buys {*Vivaldi*}, {*Handel*}, {*Mozart*} in that order. None of the previous rules would match that sequence to predict that the customer may buy {*Berlioz*} next. If a partial matching is used, a problem would be to choose between rules R1, R2 and R3 because they are rated quite differently (the support varies from 16% to 33% and the confidence from 50 % to 100%).

1.2 Contributions

Facing these issues, in this article, we explore the idea of mining "partially-ordered sequential rules" (POSR), a more general form of sequential rules common to multiple sequences such that items in the antecedent and in the consequent of each rule are unordered. This definition has the benefits of summarizing several rules by single rules. For example, the rule {*Mozart*, *Vivaldi*, *Handel*} ⇒ {*Berlioz*} replaces all the previous rules and has a support of 75 % and a confidence of 66%.

To discover POSR, we propose an efficient algorithm named RuleGrowth. It uses a novel approach named "rule expansions" to generate sequential rules and includes several strategies to perform the search efficiently. RuleGrowth is easily extendable. Constraints can be added to the algorithm for the needs of specific applications. For example, we present an extension named TRuleGrowth that finds rules occurring with a sliding-window constraint. This constraint is important for many real applications because users often only wish to discover patterns occurring within a maximum amount of time [14], [20], [30].

To evaluate the proposed algorithms, we conduct an extensive performance study comparing their performance with two baseline algorithms on four real-life datasets representing different types of data (protein sequences, click-stream data, customer data and language utterances). Results show that RuleGrowth outperforms baseline algorithms in all situations in terms of execution time and memory consumption. Moreover, results show that the execution time and the number of rules discovered can be reduced by several orders of magnitude when the sliding-window constraint is used. Experiments also show that the proposed algorithms have excellent scalability. Furthermore, we also report a real application where results have shown that the prediction accuracy obtained using POSR can be much higher than using sequential rules.

The rest of this article is organized as follows. Sections 2 and 3 respectively present background and related work. Section 4 defines the problem of mining partially-ordered sequential rules common to multiple sequences and explains the relationship to related work. Section 5 and Section 6 respectively present RuleGrowth and TRuleGrowth. Section 7 presents the evaluation. Finally, the last section draws a conclusion.

2 Background

A **sequence database** [2] SD is a set of sequences $S=\{s_1, s_2, \ldots s_m\}$ and a set of **items** $I=\{i_1, i_2, \ldots i_n\}$ occurring in these sequences, where each sequence is assigned a unique **SID** (Sequence ID). A **sequence** is an ordered list of **itemsets** (sets of items) $s_x=I_1, I_2, \ldots I_p$ such that $I_1, I_2, \ldots I_p \subseteq I$. For instance, Table 1 depicts a sequence database containing four sequences respectively having the sids *seq1*, *seq2*, *seq3* and *seq4*. In this example, each single letter represents an item. Item(s) 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*.

TABLE 1
A Sequence Database

SID	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*, *h*}

A sequence where each itemsets is annotated with a timestamp is called a **time-sequence**. For example, Fig. 1 shows a time-sequence containing 13 itemsets. A **window** is a group of consecutive itemsets in a sequence. A **sliding-window** is a window that is assumed to move from the beginning of a sequence to its end, one itemset (or one time unit) at a time. For example, a sliding-window with a length of 3 time units can move in 15 different positions w_1, w_2, … w_{15} for the sequence depicted in Fig. 1 (each position is said to be a window). Note that some windows (w_1, w_2, w_{14}, w_{15}) extend outside the sequence so that each itemset appears in the same number of windows.

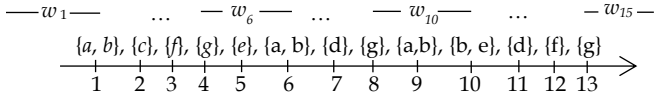

Fig. 1. A time-sequence

3 Related Work

In this section, we systematically review relevant related work.

Sequential pattern mining. It consists of discovering subsequences appearing in a sequence database such that their support is no less than a threshold *minsup* set by the user [2], [14], [17], [18]. The support of a subsequence is defined as the number of sequences that contains it divided by the total number of sequences. For example, a sequential pattern found in the database of Table 1 for *minsup = 50 %* is {*a*}, {*c*}, {*e*}. It has a support of 50% because it appears in two sequences out of four (*seq1* and *seq2*).

Mining sequential rules in a single sequence. Several works were done to discover sequential rules in a single time-sequence. The most famous is the one of Mannila et al [13], to discover rules of the form X⇒Y such that X and Y are unordered itemsets. These rules are interpreted as "if item(s) X appears, item(s) Y will also appear within that window with a given confidence. To mine these rules, the user has to specify a sliding-window size, a minimum support threshold *minsup* and a minimum confidence threshold *minconf*. The *support* of a rule is the percentage of windows in which the rule occurs. The *confidence* is the number of windows containing the rule divided by the number of windows containing its antecedent. For example, for the time-sequence depicted in Fig. 1, the rule {*a*, *b*}⇒{*g*} has a support of 2/15 and a confidence of 2/9 if the sliding-window size is set to 3 time units. To discover this form of rules, two steps are performed: (1) discovering itemsets that appear in at least *minsup* percent of the windows and (2) generating rules by using pairs of those itemsets [13]. Multiple algorithms relying on similar definitions have been proposed, for example, to discover rules with only two items [8] or where consequents are restricted to a single item [5].

Mining sequential rules across sequences. To mine sequential rules in sequence databases, two categories of algorithms have been designed. The first one is algorithms for mining sequential rules *across sequences* [4, 9, 29]. A representative example is MOWCATL [9], which discovers rules of the form X⇒Y, where X and Y are unordered itemsets. To mine rules, the user has to provide a time-sequence database and specify a sliding-window size, a minimum support threshold and a minimum confidence threshold. The support of a rule X⇒Y is defined as in Mannila et al [13] except that X and Y are not required to appear in the same sequence. For example, the rule {*d*} ⇒ {*f*, *g*, *h*} is said to appear once in the database of Table 1 because {*d*} appears in the first itemset of *seq2* and {*f*, *g*, *h*} appear in the second itemset of *seq4*. A similar algorithm was also proposed in [4].

Mining sequential rules common to several sequences. Several algorithms have been proposed to mine sequential rules *common to multiple sequences* in a sequence database. Most of these algorithms consist of applying a sequential pattern mining algorithm and then to perform a post-processing step to generate rules between pairs of sequential patterns [12], [17], [19], [22]. An example is RuleGen [17]. It discovers rules of the form X⇒Y such that X and Y are sequential patterns, and that rules respect a minimum support and a minimum confidence threshold. To mine these rules, RuleGen first discovers sequential patterns with the SPADE algorithm [17] and then combines pairs of sequential patterns to generate rules. For example, the rule {*a*}, {*c*} ⇒ {*e*} can be found by RuleGen in the database of Table 1. It means that *a* followed by *c*, then followed by *e*, has a support of 50 % and a confidence of 100%. However, as mentioned in section 1.1, the definition of a sequential rule as a relationship between two sequential patterns can cause several problems. Facing this issue, in this article, we explore the idea of mining "partially-ordered sequential rules" (POSR), a form of rules such that items in the antecedent and consequent are unordered. We show the benefits of this definition in terms of increased prediction accuracy for a real application in section 7.7.

Note that other definitions of sequential rules could also be studied. For example, instead of mining rules with unordered or with sequentially ordered antecedents and consequents, an in-between solution could be to mine

rules with partial orders as antecedent/consequent akin to the work of [13]. This is an idea that we consider exploring in future work. Moreover, other ways of reducing, summarizing or compressing rules could be studied. For example, a popular idea in pattern mining is to mine a representative subset of patterns instead of all patterns [12], [30]. This idea has been applied for sequential rules and POSR respectively in [12] and [30] by mining rules having a minimal antecedent and a maximal consequent.

4 Problem Definition and Relationship to Related Work

In this section, we define the problem of mining partially-ordered sequential rules.

4.1 Problem Definition

Consider a sequence database SD containing a set of sequences S and a set of items I, as defined in section 2 and illustrated in Table 1. A **partially-ordered sequential rule** X⇒Y is a relationship between two **unordered** itemsets X, Y ⊆ I such that X∩Y = Ø, X ≠ Ø and Y ≠ Ø. The interpretation of a rule X⇒Y is that if items of X occur in a sequence, items of Y will occur afterward in the same sequence. Formally, we say that a **rule** X⇒Y **occurs in a sequence** s_x=I_1, I_2 … I_n if there exists an integer k such that $1 \leq k < n$, $X \subseteq \bigcup_{i=1}^{k} I_i$ and $Y \subseteq \bigcup_{i=k+1}^{n} I_i$. For example, the rule {*a*, *b*, *c*}⇒{*e*, *f*, *g*} occurs in the sequence {*a*, *b*}, {*c*}, {*f*}, {*g*}, {*e*}, whereas the rule {*a*, *b*, *f*}⇒{*c*} does not, because item *c* does not occur after *f*. A rule X⇒Y is said to be **of size *k**m** if |X| = *k* and |Y| = *m*. Note that the notation ***k**m*** is not a product. It is simply a short way of writing that the sizes of the left and right parts of a rule are respectively *k* and *m*. For example, the rules {*a*, *b*, *c*}⇒{*e*, *f*, *g*} and {*a*}⇒{*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 ≥ i, or alternatively if *f* ≥ *h* and g > i.

For a given sequence database and a rule X⇒Y, the notation *sids*(X⇒Y) represents the **sid set** (the set of sequence ids) of the sequences where the rule occurs. For instance, for the database illustrated in Table 1, *sids*({*a*}⇒{*b*}) = {*seq2*, *seq3*}. For an itemset X and a sequence database, the notation *sids*(X) denotes the sid set corresponding to sequences where all the items of X appears. For example, *sids*({*a*, *sup*(*b*, *c*}) = {*seq1*, *seq2*}. For the sake of brevity, in the rest of this article, curly brackets will be omitted when using the "*sids*" notation with itemsets containing a single item. For instance, we will write *sids*(a) instead of *sids*({a}) and *sids*(a⇒b) instead of *sids*({a}⇒{b}). We define two interestingness measures for sequential rules, which are based on the measures used in sequential rule mining. 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)|.

Consider a minimum support threshold *minsup* and a minimum confidence threshold *minconf* set by the user in the [0, 1] interval. A rule is said to be a **frequent rule** if its support is no less than *minsup*. A rule is said to be a **valid rule** if it is a frequent rule and its confidence is no less than *minconf*.

Problem definition. The problem of mining sequential rules common to multiple sequences is to find all valid rules in a sequence database SD, given the thresholds *minsup* and *minconf* set by the user.

Example 1. Table 2 shows some valid rules found in the database of Table 1 for *minsup* = 0.5 and *minconf* = 0.5. For instance, the rule {*a*, *b*, *c*}⇒{*e*} has a support of |*sids*(X⇒Y)| / |S| = 2 / 4 = 0.5 and a confidence of |*sids*(X⇒Y)| / |*sids*(X)| = 2 / 2 = 1. Because those values are respectively no less than *minsup* and *minconf*, the rule is deemed valid.

TABLE 2
SOME SEQUENTIAL RULES

ID	Rule	Support	Confidence
r1	{*a*, *b*, *c*}⇒{*e*}	0.50	1.00
r2	{*a*}⇒{*c*, *e*, *f*}	0.50	0.66
r3	{*a*, *b*}⇒{*e*, *f*}	0.75	1.00
r4	{*b*}⇒{*e*, *f*}	0.75	0.75
r5	{*a*}⇒{*e*, *f*}	0.75	1.00
r6	{*c*}⇒{*f*}	0.50	1.00
r7	{*a*}⇒{*b*}	0.50	0.66

4.2 Relationship to Related Work

The problem of mining partially-ordered sequential rules common to multiple sequences is different from other problems discussed in the related work section. The main differences are as follows. For the problem of mining **sequential rules in a single sequence** or the **problem of mining sequential rules across sequences,** the support and confidence of a rule are defined based on how many windows contain a rule and its antecedent. Algorithms for these problems take advantage of the fact that adjacent windows are overlapping (a window of size *n* shares *n-1* itemsets with the next window). The problem that we address in this paper is different from those problems because the support and confidence are defined based on the number of sequences that contains a rule and its antecedent, and it cannot be assumed that sequences are overlapping.

The problem addressed in this paper is also different from **sequential pattern mining**. The main difference is that a sequential pattern is a list of one or more itemsets such that items inside each itemset have to appear together in a sequence (in the same itemset). Conversely, a partially-ordered sequential rule is a relationship between only two itemsets where items in each itemset are unordered and are not required to appear in the same itemset in a sequence.

4.3 Baseline algorithms

In previous work, we proposed two algorithms named CMRules and CMDeo [7] for mining partially-ordered sequential rules, which will be used as baseline algorithms in this article.

CMRules is based on the idea that partially-ordered sequential rules can be seen as a subset of association rules [7]. CMRules performs two steps to discover sequential rules. First, it ignores the temporal information from the sequence database taken as input to mine association rules [1]. Then, to obtain sequential rules from

association rules, CMRules scans the original database to eliminate rules that do not meet *minsup* and *minconf* according to the sequential ordering. The main benefits of CMRules are that association rule mining algorithms can be reused to implement the algorithm and that it performs better than CMDeo for some datasets [7]. Its main drawback is that its performance depends on the number of association rules. If this set is large, CMRules becomes inefficient [7].

CMDeo proceeds by first scanning the database to generate rules of size 1*1 (containing two items). The algorithm then recursively finds larger candidate rules by combining frequent rules of smaller size in a level-wise manner (similar to Apriori [1]). This is done by two separate processes. Left-side expansion is the process of taking two frequent rules X⇒Y and Z⇒Y, where X and Z are itemsets of size n sharing n-1 items, to generate a larger candidate rule X∪Z⇒Y. Right-side expansion is the process of taking two frequent rules Y⇒X and Y⇒Z, where X and Z are itemsets of size n sharing n-1 items, to generate a larger candidate rule Y⇒X∪Z. After candidate rules are generated, their support and confidence are calculated by scanning original sequences of the database. To prune the search space of candidate rules, CMDeo uses the property that expanding the left side of a rule not respecting *minsup* will not result in valid sequential rules, and that expanding the right side of a rule not respecting *minsup* or *minconf* will not generate valid sequential rules [7]. In [7], we extensively compared the performance of CMDeo and CMRules. It was found that CMDeo performs considerably better than CMRules for some datasets. But for others, the search space is such that CMDeo generates a very large number of candidate rules that are invalid, which makes CMRules more efficient.

5 The RuleGrowth Algorithm

In this section, we present RuleGrowth, a novel algorithm that we have designed to mine partially-ordered sequential rules more efficiently.

5.1 Main Features

A common characteristic of CMRules and CMDeo is that both use a generate-candidate-and-test approach, which consists of generating candidate rules and then to scan the database to determine their support and confidence. The problem with this approach is that it often produces a large amount of candidate rules and that a large proportion are invalid or do not appear in the database. Therefore, these algorithms spend a lot of time to tell apart valid rules from invalid ones.

The RuleGrowth algorithm that we propose in this article avoids this problem of candidate generation by instead relying on a pattern-growth approach partly inspired by the one used in the PrefixSpan algorithm [14] for sequential pattern mining[2]. RuleGrowth first finds rules of size 1*1 and then recursively grows them by scanning the sequences containing them to find single items that can expand their left or right sides. This strategy ensures that only rules occurring in the database are considered as potential valid rules by the algorithm. We name the two processes for expanding rules in RuleGrowth *left expansion* and *right expansion* akin to the homonym processes of CMDeo. Note however that these processes are different; RuleGrowth finds larger rules by adding items to rules by scanning sequences containing the rules (a depth-first search), whereas CMDeo combine pairs of rules to generate candidates (a breadth-first search). Another distinctive feature of RuleGrowth is that it keeps track of the first and last occurrences of each item, and also of antecedents and consequents to avoid scanning sequences completely, as it will be explained.

5.2 Preliminary definitions and properties

Before presenting the algorithm, we introduce important definitions and properties. 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. A **right expansion** is defined as the process of adding an item i to the right side of a rule X⇒Y to obtain a larger rule X⇒Y∪{i}. Left and right expansions have the following 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 $|sids(X⇒Y)| \geq |sids(X∪\{i\}⇒Y)|$, $sup(r) \geq sup(r')$.□

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 $|sids(X⇒Y)| \geq |sids(X⇒Y∪\{i\})|$, $sup(r) \geq sup(r')$.□

The two previous properties imply that the support is monotonic with respect to left and 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 support of the original rule. Therefore, all rules having a support of at least *minsup* can be found by recursively performing expansions starting from frequent rules of size 1*1. Moreover, expanding a rule having a support less than *minsup* will not result in a frequent rule. The confidence, however, is not monotonic with respect to left and right expansions as the 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 $|sids(X∪\{i\}⇒Y)|$ /

[2] The similarity between RuleGrowth and PrefixSpan is that both algorithms create small patterns and "grow" them by recursively adding an item at a time to them to find larger patterns, and that items for growing patterns are selected by scanning the database. However, besides this similarity, the algorithms are different.

$|sids(X \cup \{i\})|$. Because $|sids(X \Rightarrow Y)| \geq |sids(X \cup \{i\} \Rightarrow Y)|$ and $|sids(X)| \geq |sids(X \cup \{i\})|$, $conf(r)$ 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*:$X \Rightarrow Y$, *the confidence of the resulting rule r'*: $X \Rightarrow Y \cup \{i\}$ *is lower or equal to the confidence of r.*

Proof. The confidence of r and r' are respectively $|sids(X \Rightarrow Y)| / |sids(X)|$ and $|sids(X \Rightarrow Y \cup \{i\})| / |sids(X)|$. Since $|sids(X \Rightarrow Y)| \geq |sids(X \Rightarrow Y \cup \{i\})|$, $conf(r) \geq conf(r')$.□

The RuleGrowth algorithm relies on the use of **sid sets** (cf. section 4.1) to calculate the support and confidence of rules obtained by left or right expansions. Sid sets have two important properties for sequential rules.

Property 5 (sid set of a rule and its itemsets) *For any sequential rule* $X \Rightarrow Y$, $sids(X \Rightarrow Y) \subseteq sids(X) \cap sids(Y)$.

Property 6 (sid set of a rule obtained by left or right expansion) *For any sequential rule r' obtained by a left or right expansion of a rule r, the relationship* $sids(r') \subseteq sids(r)$ *holds.*

RuleGrowth also relies on two additional definitions. The **first occurrence of an itemset X in a sequence** $s=I_1, I_2 \ldots I_n$ is the itemset $I_k \in s$ such that $X \subseteq \bigcup_{i=1}^{k} I_i$ and there exists no $g < k$ such that $X \subseteq \bigcup_{i=1}^{g} I_i$. The **last occurrence of an itemset X in a sequence** $s=I_1, I_2 \ldots I_n$ is the itemset $I_k \in s$ such that $X \subseteq \bigcup_{i=k}^{n} I_i$ and there exists no $g > k$ such that $X \subseteq \bigcup_{i=g}^{n} I_i$. For example, the first occurrence of $\{a, b\}$ in the sequence $\{a, d\}, \{b\}, \{a\}, \{b\}, \{e\}$ is the second itemset, whereas the last occurrence of $\{a, b\}$ is the third itemset.

5.3 The Algorithm

The RuleGrowth algorithm takes as input a sequence database S and the *minsup* and *minconf* thresholds.

Main procedure. RuleGrowth's main procedure is shown in Fig. 2. The algorithm first scans the database once to calculate $sids(c)$ for each item c (line 1). Then, the algorithm identifies all items c such that $|sids(c)| / |S| \geq minsup$, because only these items can be part of a valid rule. The algorithm generates all valid rules of size 1*1 using these items (line 2). This is done by considering each pair of items i, j one by one. The algorithm scans sequences in $sids(i) \cap sids(j)$ 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 (line 5 to 7; Property 5). Then, the support of the rule $\{i\} \Rightarrow \{j\}$ is obtained by dividing $|sids(i \Rightarrow j)|$ by $|S|$ (line 8). If the support is no less than *minsup*, the procedure EXPANDLEFT and EXPANDRIGHT are called to try to expand the rule's left and right parts recursively (line 9 to 10), and the confidence of the rule is calculated by dividing $|sids(i \Rightarrow j)|$ by $|sids(i)|$. If the confidence is higher than or equal to *minconf*, the rule is valid and the algorithm outputs the rule (line 11). After this, the same process is repeated for the rule $\{j\} \Rightarrow \{i\}$ (line 12 to 16). Then, the algorithm considers all other pairs of items i, j in the same way.

It can be easily seen that the main procedure of RuleGrowth outputs all and only the valid rules of size 1*1. We next explain how it can also find all valid rules of larger size by recursively adding one item at a time to the left or right side of frequent rules of size 1*1 with the procedures EXPANDLEFT and EXPANDRIGHT (by left and right expansions). To develop these two procedures, the following problems had to be solved.

RULEGROWTH(S, *minsup*, *minconf*)
1. **Scan** the database S once. For each item c found, record the *sids* of the sequences that contains c in a variable $sids(c)$.
2. **FOR** each pair of items i, j such that $|sids(i)| / |S| \geq minsup$
3. and $|sids(j)| / |S| \geq minsup$ {
4. $sids(i \Rightarrow j) := \emptyset$. $sids(j \Rightarrow i) := \emptyset$.
5. **FOR** each sid $s \in (sids(i) \cap sids(j))$ {
6. **IF** i occurs before j in s, $sids(i \Rightarrow j) := sids(i \Rightarrow j) \cup \{s\}$.
7. **IF** j occurs before i in s, $sids(j \Rightarrow i) := sids(j \Rightarrow i) \cup \{s\}$. }
8. **IF** $(|sids(i \Rightarrow j)| / |S|) \geq minsup$ **THEN {**
9. **EXPANDLEFT**($\{i\} \Rightarrow \{j\}$, $sids(i)$, $sids(i \Rightarrow j)$).
10. **EXPANDRIGHT**($\{i\} \Rightarrow \{j\}$, $sids(i)$, $sids(j)$, $sids(i \Rightarrow j)$).
11. **IF** $(|sids(i \Rightarrow j)| / |sids(i)|) \geq minconf$ **THEN OUTPUT** rule $\{i\} \Rightarrow \{j\}$ with its confidence and support. **}**
12. **IF** $(|sids(j \Rightarrow i)| / |S|) \geq minsup$ **THEN {**
13. **EXPANDLEFT**($\{j\} \Rightarrow \{i\}$, $sids(j)$, $sids(j \Rightarrow i)$).
14. **EXPANDRIGHT**($\{j\} \Rightarrow \{i\}$, $sids(j)$, $sids(i)$, $sids(j \Rightarrow i)$).
15. **IF** $(|sids(j \Rightarrow i)| / |sids(j)|) \geq minconf$ **THEN OUTPUT** rule $\{j\} \Rightarrow \{i\}$ with its confidence and support. **}}**

Fig. 2. The RuleGrowth algorithm

How to determine which items to use for performing left and right expansions and obtain valid rules? The first problem is how to identify items that can expand a rule I⇒J left part or right part to produce a valid rule. By exploiting the fact that any valid rule is also 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 r:I⇒J and produce a frequent rule, our solution is to scan the sequences from $sids(I \Rightarrow J)$ (Property 6). During this scan, each item c such that $c \notin I$, $c \notin J$ and c occurs before the last occurrence of J in at least $minsup \times |S|$ sequences from $sids(I \Rightarrow J)$ is noted. Those items are the ones that will produce a frequent rule by a left expansion of r. For right expansions, we note each item $c \notin I$ such that $c \notin J$ and c occurs after the first occurrence of I in at least $minsup \times |S|$ sequences from $sids(I \Rightarrow J)$.

The second sub-problem is to determine if a rule obtained by a left or right expansion of a frequent rule I⇒J with an item c is a valid rule. To do this, the confidence of the rule has to be calculated. There are two cases. For a left expansion, the confidence is obtained by dividing $|sids(I \cup \{c\} \Rightarrow J)|$ by $|sids(I \cup \{c\})|$. The set $sids(I \cup \{c\} \Rightarrow J)$ is determined by noting each sequence where c expands the rule I⇒J when searching items for the left expansion of I⇒J, as explained in the previous paragraph. The set $sids(I \cup \{c\})$ is calculated by scanning each sequence from $sids(I)$ to see if c appears in it. For a rule of size 1*1, $sids(I)$ is determined during the initial database scan of the algorithm (line 1 of Fig. 2), and for larger rules, it can be updated after each left expansion. For a right expansion, the confidence is calculated by dividing $|sids(I \Rightarrow J \cup \{c\})|$ by

$|sids(I)|$. The set $sids(I \Rightarrow J \cup \{c\})$ is determined by noting each sequence where c expand the rule $I \Rightarrow J$ when searching items for the right expansion of $I \Rightarrow J$ as explained in the previous paragraph.

How can we guarantee that all valid rules are found by recursively performing left/right expansions? The next problem is how to guarantee that all valid rules will be found by recursively performing left/right expansions starting from rules of size 1*1. The answer is found in Properties 1 and 2, which state 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 left/right 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 confidence because the confidence of a rule is not monotonic with respect to left expansions (Property 3).

How can we guarantee that no rule is found twice? Previous paragraphs explained how expansions can lead to the discovery of all and only valid rules. Another challenge is to ensure that no rule is found twice. To achieve this, two problems had to be solved. First, if we grow rules by performing left/right 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 $\{a\} \Rightarrow \{c\}$. This problem is illustrated in Fig. 3(A). For example, rules of size 2*2 can be found respectively by left expansions of rules of size 1*2 and by right expansions of rules of size 2*1. A simple solution to avoid this problem is to not allow performing a right expansion after a left expansion but to allow performing a left expansion after a right expansion. This solution is illustrated in Fig. 3(B). Note that an alternative solution is to not allow performing a left expansion after a right expansion but to allow performing a right expansion after a left expansion.

Second, rules can be found several times by performing left/right expansions with different items. For example, consider the rule $\{b, c\} \Rightarrow \{d\}$. A left expansion of $\{b\} \Rightarrow \{d\}$ with item c results in $\{b, c\} \Rightarrow \{d\}$. But that latter rule can also be found by performing a left expansion of $\{c\} \Rightarrow \{d\}$ with b. To solve this problem, we chose 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 rule is found twice.

Pseudo-code of the EXPANDLEFT and EXPANDRIGHT procedures. Fig. 4 and 5 present the pseudo-code of the EXPANDLEFT and EXPANDRIGHT procedures, which incorporate all the above ideas.

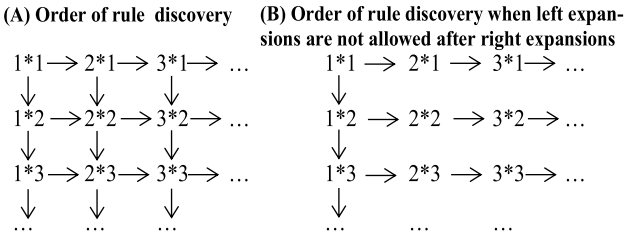

Fig. 3. The order of rule discovery by left/right expansions

EXPANDLEFT(I⇒J, *sids*(I), *sids*(I⇒J))
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 I and appears before J, record *sid* in a variable *sids*(I∪{c}⇒J).
2. **FOR** each item *c* where $|sids(I \cup \{c\} \Rightarrow J)| \geq minsup \times |S|$ {
3. $sids(I \cup \{c\}) := \emptyset$.
4. **FOR** each *sid* ∈ *sids*(I) such that *sid* ∈ *sids*(c){
5. $sids(I \cup \{c\}) := sids(I \cup \{c\}) \cup \{sid\}$. }
6. **EXPANDLEFT**(I∪{*c*}⇒J, *sids*(I∪{*c*}), *sids*(I∪{*c*}⇒J))
7. **IF** $|sids(I \cup \{c\} \Rightarrow J)| / |sids(I \cup \{c\})| \geq minconf$
8. **THEN OUTPUT** rule I∪{c}⇒J. }

Fig. 4. The EXPANDLEFT procedure

EXPANDRIGHT(I⇒J, *sids*(I), *sids*(I⇒J))
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 $|sids(I \Rightarrow J \cup \{c\})| \geq minsup \times |S|$ {
3. **EXPANDLEFT**(I⇒J∪{*c*}, *sids*(I), *sids*(I⇒J∪{*c*})).
4. **EXPANDRIGHT**(I⇒J∪{*c*}, *sids*(I), *sids*(I⇒J∪{*c*})).
5. **IF** $|sids(I \Rightarrow J \cup \{c\})| / |sids(I)| \geq minconf$
6. **THEN OUTPUT** rule I⇒J∪{*c*}. }

Fig. 5. The EXPANDRIGHT procedure

5.4 Optimizing RuleGrowth by keeping track of first and last occurrences

This section describes an optimization that considerably enhances the performance of RuleGrowth. It consists of calculating the first and last occurrences of items in each sequence and then to use this information to avoid scanning sequences completely. Three modifications were done to implement this optimization.

First, RuleGrowth is modified (cf. Fig. 2) so that when the database is scanned for the first time (line 1 of Fig. 2), the first and last occurrences of each item in each sequence is recorded. We represent the occurrence of an item in a sequence as an integer indicating the position of an itemset containing the item. For example, the occurrences of a in sequence $seq2$=$\{a, d\}$,$\{c\}$,$\{b\}$,$\{a, b, e, f\}$ are 1 and 4 (a appears in the first and fourth itemsets), and the occurrences of b are 3 and 4 (b appears in the third and fourth itemsets). In our implementation, the first and last occurrences of items in sequences are stored in hash tables to have O(1) access to this information.

Second, the generation of frequent rules of size 1*1 is modified. By using the information about the first and last occurrences of items, rules of size 1*1 are generated without scanning the original database. This is done by looking in the hash tables for each pairs of items i, j if the

first occurrence of *i* is before the last occurrence of *j* and vice versa in sequences of (*sids*(*i*) ∩ *sids*(*j*)) to calculate *sids*(*i*⇒*j*) and *sids*(*j*⇒*i*). For example, to check if *b* appears before *c* in sequence *seq1* of Table 1, the first occurrence of *b* and the last occurrence of *c* for sequence *seq1* are retrieved from the hash tables. These values are respectively 1 and 2. Because 1 < 2, it is concluded that *b* appears before *c* in *seq1*. Because the database is not scanned for calculating *sids*(*i*⇒*j*) and *sids*(*j*⇒*i*), generating rules of size 1*1 is much faster.

Third, the way left and right expansions are performed is also modified to take advantage of the information about first and last occurrences. Recall that a rule I⇒J is only expanded by RuleGrowth with items appearing after the first occurrence of itemset I for a right expansion, and occurring before the last occurrence of itemset J for a left expansion. Therefore, if the first occurrences of I and the last occurrences of J are known, the procedure EXPANDLEFT and EXPANDRIGHT could avoid scanning sequences completely when searching for items to expand the rule. To provide this information to EXPANDLEFT and EXPANDRIGHT, we have modified the main RULEGROWTH procedure so that for each frequent rule {*i*}⇒{*j*} of size 1*1, the last occurrences of *j* are passed as parameter to EXPANDLEFT and the first occurrences of *i* and last occurrences of *j* are passed as parameters to EXPANDRIGHT (both first and last occurrences are passed to EXPANDRIGHT because it can call both EXPANDLEFT and EXPANDRIGHT). Then, we have modified EXPANDLEFT so that the last occurrences of J are passed to each recursive call to EXPANDLEFT. Similarly, we have modified EXPANDRIGHT so that the first and last occurrences are passed to each recursive call to EXPANDRIGHT and that last occurrences of J are passed to EXPANDLEFT. But note that before the last occurrences are passed to EXPANDLEFT by EXPANDRIGHT, last occurrences of J have to be recalculated because EXPANDRIGHT adds an item to J and this can change the last occurrences of J. This recalculation is done efficiently by using the hash tables containing the occurrences of each item for each sequence.

6 TRuleGrowth: Extending RuleGrowth to Discover Rules with a Sliding-Window

Because RuleGrowth grows rules one item at a time, constraints can be easily added to the algorithm for the needs of specific applications. For example, it would be easy to add constraints on the number of items that rules can contain or to restrict the items that are added to rules.

In this section, we present one particular extension which is to discover rules occurring within a sliding-window, i.e. within a maximum number of consecutive itemsets in each sequence. We present this extension because applying a sliding-window has shown to be very useful for the discovery of temporal patterns for many real-life applications such as analyzing sensor networks and stock market data, because users often only wish to discover patterns occurring within a maximum amount of time [14], [20]. For this reason, several data mining algorithms use a sliding-window (e.g. [4], [5], [9], [13]) or have been modified to accept one (e.g. [20]). We name TRuleGrowth the extension of RuleGrowth that discovers rules while verifying that they occur in a sliding-window. As it will be shown, discovering rules appearing in a sliding-window has several important benefits. First, it can decrease the execution time by several orders of magnitude by pruning the search space. Second, it can produce a much smaller set of rules, thus reducing the disk space requirement for storing rules found and making it easier for the user to analyze results. Third, setting a window constraint can increase prediction accuracy when rules are used for prediction (see section 7.7 for results). Note that adding a sliding-window could also be done with CMRules/CMDeo. However, it is best done with RuleGrowth because it can check the window constraint when it scans sequences to search for items, whereas CMRules/CMDeo can only verify that rules respect the the window after rules have been generated (they would generate many rules not respecting the sliding-window).

6.1 Problem Definition

We define the problem of mining sequential rules common to multiple sequences with a sliding-window as being the same as the problem of mining sequential rules common to multiple sequences except that the definition of the occurrence of a rule in a sequence is changed so that rules have to respect the sliding-window (have to appear within a given number of consecutive itemsets). A rule X⇒Y is said to **occur** in a sequence $s=I_1, I_2 \dots I_n$ if there exist integers *j*, *k*, *m* such that $1 \le j \le k < m \le n$, $X \subseteq \bigcup_{i=j}^{k} I_i$, $Y \subseteq \bigcup_{i=k+1}^{m} I_i$ and that $m - j + 1 \le window\_size$, where *window_size* is defined by the user.

Example 2. Table 3 shows sequential rules found in the database presented in Table 1 for *minsup* = 0.5, *minconf* = 0.5 and *window_size* = 3. Consider the rule {*a*}⇒{*f*}. Because it occurs in sequences *seq1* and *seq3*, it has a support of $|sids(a \Rightarrow f)| \,/\, |S| = 2 / 4$. The confidence of {*a*}⇒{*f*} is $|sids(a \Rightarrow f)| \,/\, |sids(a)| = 2/3$. Note, that {*a*}⇒{*f*} does not occur in *seq2* because the parameter *window_size* is set to 3. If *window_size* was set to a value higher than 3, the rule {*a*}⇒{*f*} would occur in *seq2* and thus the support and confidence of {*a*}⇒{*f*} would be 3/4 and 3/3, respectively.

TABLE 3
SEQUENTIAL RULES FOUND WITH *WINDOW_SIZE* = 3

ID	Rule	Support
r1	{*a*}⇒{*b*}	0.5
r2	{*a*}⇒{*c*}	0.5
r3	{*a*}⇒{*f*}	0.5
r4	{*b*}⇒{*e*}	0.5
r5	{*b*}⇒{*f*}	1.0
r6	{*c*}⇒{*f*}	0.5
r7	{*f*}⇒{*e*}	0.5

6.2 The TRuleGrowth Algorithm

TRuleGrowth is a modified version of RuleGrowth. Two modifications are made to the RULEGROWTH procedure (cf. Fig. 2) to ensure that the sliding-window constraint is taken into account when generating rules of size 1*1.

First, instead of keeping the first and last occurrences of each item for each sequence as explained in section 4.3, all occurrences of each item are now kept for each sequence. Recall that an occurrence of an item for a sequence is represented as an integer indicating the position of an itemset containing the item. For example, the occurrences of *a* in sequence *seq*2 = {*a*, *d*},{*c*},{*b*},{*a*, *b*, *e*, *f*} are 1 and 4 (*a* appears in the first itemset and the fourth itemset), and the occurrences of *b* in *seq*2 are 3 and 4 (*b* appears in the third itemset and the fourth itemset).

The second change is to modify line 6 and line 7 of the RULEGROWTH procedure so that when checking if item *i* occurs before item *j* and if *j* occurs before *i* in a sequence, the check also verifies that it is true within *window_size* consecutive itemsets. This check is performed efficiently by comparing each occurrence of *i* with each occurrence of *j* for the sequence by using the hash tables. If there exists an occurrence x of i and an occurrence y of j such that $y - x > 0$ and $y - x + 1 \leq window\_size$, then it is concluded that i occurs before j in the sequence while respecting the sliding-window. Similarly, if there exists an occurrence x of j and an occurrence y of i such that $y - x > 0$ and $y - x + 1 \leq window\_size$, then it is concluded that j occurs before i in the sequence while respecting the sliding-window. For example, consider items *a* and *b* in sequence *seq*2={*a*, *d*},{*c*},{*b*},{*a*, *b*, *e*, *f*} and *window_size* = 3. By comparing occurrences of *a* and *b*, TRuleGrowth finds that item *a* appears before *b* while respecting the sliding-window because for the occurrence 1 of *a* and the occurrence 3 of *b*, $3 - 1 > 0$ and $3 - 1 + 1 = 3 \leq window\_size$. The algorithm will also discover that *b* appears before *a* while respecting the sliding-window because $4 - 3 > 0$ and $4 - 3 + 1 = 2 \leq window\_size$.

The previous modifications ensure that the sliding-window constraint is enforced for rules of size 1*1. To take this constraint into account in the generation of larger rules, we have modified the EXPANDLEFT and EXPANDRIGHT procedures. We present the modified versions of EXPANDLEFT and EXPANDRIGHT in Fig. 6 and 7 respectively. For convenience of explanations, we explain these modifications based on the original version of EXPANDLEFT and EXPANDRIGHT presented in Fig. 4 and 5, without the optimization from section 5.4.

The first modification to EXPANDLEFT (respectively, EXPANDRIGHT) is to how items are chosen for performing a left (right) expansion. EXPANDLEFT (EXPANDRIGHT) is modified so that the items chosen for a left (right) expansion are those for which the resulting rule satisfies *minsup* while respecting *window_size*. To identify efficiently all such items for the left (right) expansion of a rule I⇒J, each sequence from *sids*(I⇒J) is scanned once. For each sequence, each time that an itemset X is read, each item $c \in I \cap X$ is added to an hash table *hashI* together with the position of X in the sequence, and each item $d \in J \cap X$ is added to an hash table *hashJ* together with the position of X in the sequence. When considering the next itemset of the sequence, all items that were found more than *window_size* - 1 itemsets before are removed from *hashI and hashJ* because we consider them to be falling outside the window defined by the current itemset and the last *window_size* -1 itemsets read. When the sum of the size of *hashI* and *hashJ* equals $|I|+|J|$, it means that all items from I and J are in the current window. However, to be certain that I occurs before J in the current window, items should only be added to *hashI* when $|hashJ| = |J|$ (to *hashJ* if $|hashI| = |I|$) and *hashI* should be emptied as soon as hashJ becomes smaller than $|J|$ (*hashJ* should be emptied as soon as *hashI* becomes smaller than $|I|$). By doing this modification, when the sum of the size of *hashI* and *hashJ* equals $|I|+|J|$, each item $c \notin I \cup J$ occurring before the first item of J (after the last item of I) such that the *window_size* is respected can be added to the set of items that could expand the rule for this sequence. After scanning all sequences, the set of items that can expand I⇒J while respecting *minsup* is known. An important note for implementation is that the hash tables *hashI* and *hashJ* should only keep the most recent position for each item. In the Java programming language, this behavior is the default behavior for the "HashMap" implementation when sequences are scanned from the last itemset to the first one (from the first itemset to the last one).

EXPANDLEFT(I⇒J, *sids*(I), *sids*(I⇒J))
1. **FOR** each *sid* ∈ *sids*(I⇒J){
2. *hashI* := Ø. *hashJ* := Ø.
3. **FOR** each itemset X in sequence *sid*, from the last one to the first one. {
4. **REMOVE** all items from *hashI* and *hashJ* seen more than *window_size* – 1 itemsets before.
5. **IF** |*hashJ*| was equal to |J| and became smaller after removing items **THEN** |*hashI*| := Ø.
6. **IF** |*hashJ*| = |J| **THEN** add each item $c \in I \cap X$ to *hashI* with the position of X in sequence *sid*.
7. **IF** |*hashJ*| < |J| **THEN** add each item $d \in J \cap X$ to *hashJ* with the position of X in sequence *sid*.
8. **IF** |*hashI*| = |I| and |*hashJ*| = |J| **THEN** add *sid* to a variable *sids*(I∪{*c*}⇒J) for each item $c \notin I \cup J$ occurring before the first item of J in the window.
9. }}
10. **FOR** each item *c* where $|sids(I \cup \{c\} \Rightarrow J)| / |S| \geq minsup$ {
11. *sids*(I∪{*c*}) := Ø.
12. **FOR** each *sid* ∈ *sids*(I) such that *sid* ∈ *sids*(c) {
13. **IF** *c* and I occur within a window of size *window_size* **THEN** *sids*(I∪{*c*}):= *sids*(I∪{*c*}) ∪{*sid*}. }
14. **EXPANDLEFT**(I∪{*c*}⇒J, *sids*(I∪{*c*}), *sids*(I∪{*c*}⇒J)).
15. **IF** $|sids(I \cup \{c\} \Rightarrow J)| / |sids(I \cup \{c\})| \geq minconf$ **THEN** **OUTPUT** rule I∪{c}⇒J. }

Fig. 6. The EXPANDLEFT procedure of the TRuleGrowth algorithm

A last modification is done to EXPANDLEFT only. It is to take the sliding-window into account when recalculating *sids*(I∪{*c*}) for each item c that can expand a rule I⇒J. To do this, each sequence from *sids*(I∪{*c*}) are scanned while using hash maps to keep track of *c* occuring within the same window as J, similarly to what has been explained in the previous paragraph.

7 Performance Evaluation

To evaluate RuleGrowth and TRuleGrowth, we compared their performance with CMRules and CMDeo. Experiments were performed on a notebook computer with a 2.53 Ghz P8700 Core 2 Duo processor running Windows XP and 1 GB of free RAM. Algorithms were implemented in Java. Source code and datasets can be downloaded as

part of the SPMF data mining library [33] at http://www.philippe-fournier-viger.com/spmf/. During all experiments, memory measurements were done with the standard Java memory API.

```
EXPANDRIGHT(I⇒J, sids(I), sids(I⇒J))
16.  FOR each sid ∈ sids(I⇒J) {
17.      hashI := Ø.  hashJ := Ø.
18.      FOR each itemset X in sequence sid, from the first one
             to the last one{
19.          REMOVE all items from hashI and hashJ seen more
                 than window_size – 1 itemsets before.
20.          IF |hashI| was equal to |I| and became smaller than it
                 after removing items THEN |hashJ| := Ø.
21.          IF |hashI| = |I| THEN add each item c ∈ J∩X to hashJ
                 with the position of X in sequence sid.
22.          IF |hashI| < |I| THEN add each item d ∈ I∩X to hashI
                 with the position of X in sequence sid.
23.          IF |hashJ| = |J| and |hashI| = |I| THEN add sid to a
                 variable sids(I⇒J∪{c}) for each item c ∉ I∪J
                 occurring after the last item of I in the window. }}
24.  FOR each item c where |sids(I⇒J∪{c})| / |S| ≥ minsup {
25.      EXPANDLEFT(I⇒J∪{c}, sids(I), sids(I⇒J∪{c})).
26.      EXPANDRIGHT(I⇒J∪{c}, sids(I), sids(I⇒J∪{c})).
27.      IF |sids(I∪{c}⇒J)| / |sids(I∪{c})| ≥ minconf
             THEN OUTPUT rule I∪{c}⇒J. }}
```

Fig. 7. The EXPANDRIGHT procedure of the TRuleGrowth algorithm

7.1 Characteristics of real-life datasets

Experiments were carried on four datasets. These datasets were chosen because they are real-life datasets having varied characteristics and represent four kinds of data. The first dataset is **Kosarak** (http://goo.gl/4B6ve5). It contains 990,000 sequences of click-stream data from an online news portal. To make the experiment faster, we have used the first 70,000 sequences. Each sequence has an average length of 7.97 items (σ = 21.14, max = 796) from 21,144 different items. The second dataset is **BMS-WebView1** (**BMS1**). It contains 59,601 sequences of click-stream data from an e-commerce (http://www.ecn.purdue.edu/KDDCUP/). The number of different items is 497 items and the average sequence length is 2.51 items (σ =4.85, max = 267). The third dataset is **Snake** [10]. It contains 192 protein sequences. We have kept only sequences containing more than 50 items to make the dataset more uniform, because a few sequences are much shorter than all other sequences. This results in 163 long sequences containing an average of 60.61 items (σ = 0.6, max = 61). Note however, that after performing these experiments, we have found that experimental results are similar if all sequences are used. A distinctive feature of Snake is that it is very dense. Each item occurs in almost every sequence (there is on average 17.74 different items in each sequence, and only 20 different items for the whole dataset) and each item appearing in a sequence appears on average 3.39 times in the sequence (σ =2.24). The fourth dataset is **Sign** (http://goo.gl/1U61dv). It contains 730 sequences of sign-language utterances transcripted from videos [24]. Sequences contains on average 93.39 items (σ =12.3, max = 94) from 310 items. It is thus a moderately dense dataset with long sequences.

7.2 Experiment to assess the influence of *minsup*

The first experiment consists of running RuleGrowth, CMDeo and CMRules for the four datasets with different values for *minsup* and a fixed value for *minconf*, to assess the influence of *minsup* on the relative performance of the algorithms. Execution times, maximum memory usage of each algorithm, and the number of rules found are illustrated in Fig. 8.

For **Kosarak**, algorithms were run with *minconf* = 0.2, while varying *minsup* from 0.004 to 0.001. Execution times, maximum memory usage of each algorithm, and the number of rules found are illustrated in Fig. 8 (A). For this experiment a time limit of 2,500 seconds was set and a memory usage of 1 GB. Because of these limits, CMDeo and CMRules are unable to provide results for *minsup* values lower than 0.0025 (13,006 rules) and 0.00175 (100,900 rules) respectively, while RuleGrowth can still run at 0.001 (2,910,355 rules). For this dataset, RuleGrowth outperforms CMDeo and CMRules both in terms of memory consumption and execution time for all *minsup* values.

For **BMS1**, algorithms were run with *minconf* = 0.2 while varying *minsup* from 0.00085 to 0.0006. Results are illustrated on Fig. 8 (B). For this experiment, a time limit of 2,500 seconds was set and a memory limit of 1 GB. Again, RuleGrowth outperforms CMDeo and CMRules.

For **Snake**, algorithms were run with *minconf* = 0.2 and *minsup* = 0.96, 0.94 … 0.7. Results are illustrated on Fig. 8 (C). For this experiment a time limit of 700 seconds was set and a maximum memory usage of 1 GB. For this experiment, RuleGrowth is also faster and uses less memory than CMRules and CMDeo.

For **Sign**, algorithms were run with *minconf* = 0.2 and *minsup* = 0.8, 0.7 … 0.2. Results are illustrated on Fig. 8 (D). For this experiment a time limit of 1,000 seconds was set and a maximum memory usage of 1 GB. For this experiment, RuleGrowth is also faster and uses less memory than CMRules and CMDeo.

Overall, RuleGrowth performs better than CMRules and CMDeo on all datasets. Furthermore, as *minsup* is set lower, the performance gap increases.

7.3 Experiment to assess influence of *minconf*

The second experiment consists of assessing the algorithms with different *minconf* values on the same four datasets. For this experiment, *minsup* is set to the same values as in the previous experiment and *minconf* is set to 0.3 and 0.8. Results are shown in Fig. 9. Since the performance of RuleGrowth and CMDeo does not change significantly from *minconf* = 0.3 to *minconf* = 0.8 (because they don't use *minconf* to prune the search space), only their execution times for *minconf* = 0.3 are shown in Fig. 9. The performance of CMRules can however benefit from a high confidence threshold. For this reason, execution times of CMRules for *minconf* = 0.3 and *minconf* = 0.8 are shown in Fig. 9. Though, CMRules' performance increases considerably when the *minconf* threshold is raised, RuleGrowth is still the fastest in all situations.

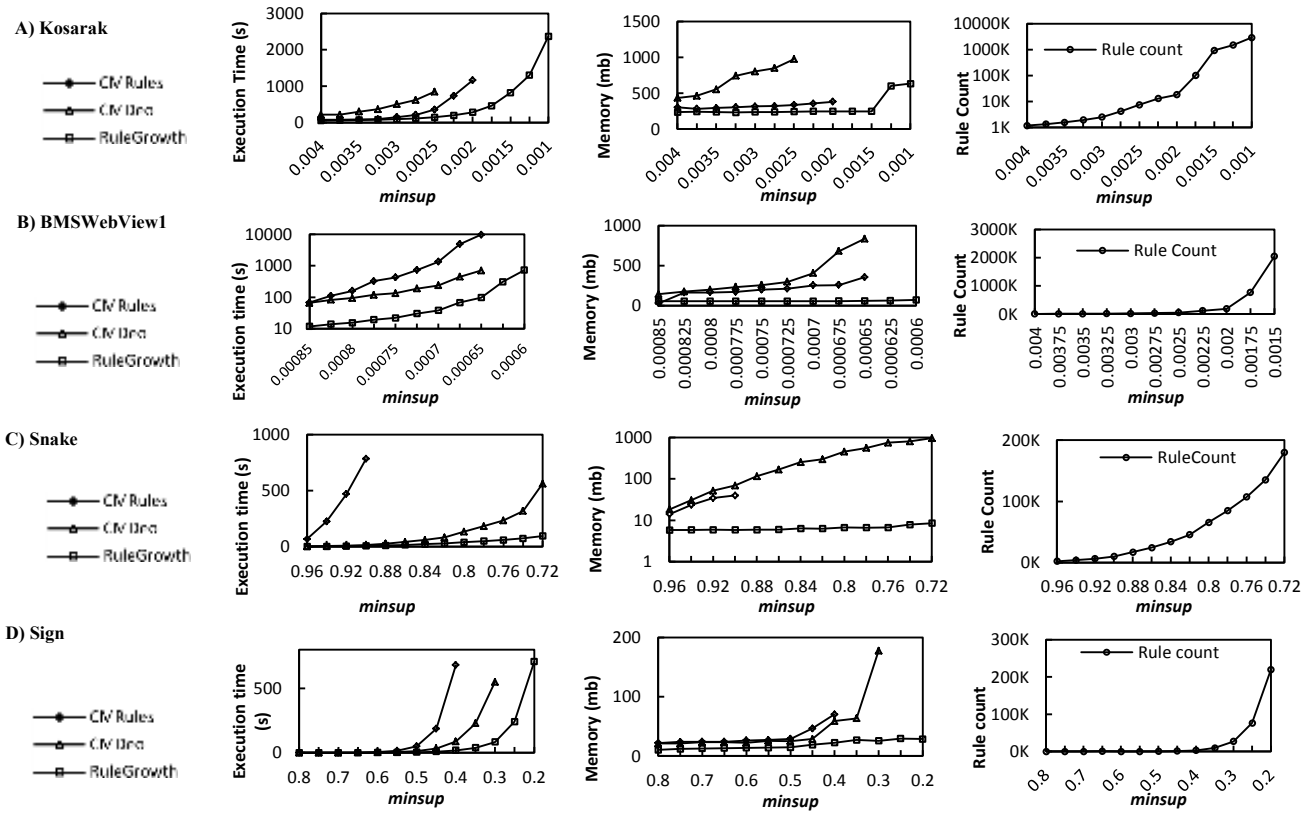

Figure 8. Influence of *minsup*

7.4 Experiment to assess influence of *window_size*

The third experiment compares the performance of RuleGrowth with TRulegrowth on the same four datasets with the same parameter values as in the previous experiments. The goal is to compare the performance of RuleGrowth and TRuleGrowth for different *window_size* values. Since choosing an appropriate *window_size* value is dataset dependent and task dependent, we have tried multiple values for each dataset. Fig. 10 shows the results for Kosarak, BMS1, Snake and Sign. In these charts, the notation W*x* represents TRuleGrowth with *window_size* = *x*. The *window_size* values shown on the charts have been selected for each dataset because they were representative values that illustrate when TRuleGrowth is faster than RuleGrowth and for which values it is slower. Thus for each dataset, we have chosen at least a value where TRuleGrowth is slower and one where it is faster.

In general, we have observed that TRuleGrowth can be several orders of magnitude faster than RuleGrowth and generate several orders of magnitude less rules. But as expected, when *window_size* is set to large values, TRuleGrowth becomes slower than RuleGrowth. For example, TRulegrowth is slower than RuleGrowth on the Snake dataset for *window_size* ≥ 30, whereas for *window_size* ≤ 20, it is faster. This is because TRuleGrowth has to perform extra calculations for verifying the window size constraint. When *window_size* is set above a certain value, this extra calculation is more costly in terms of execution time than what is saved by pruning the search space with the window size constraint. Nonetheless, even if TRuleGrowth takes more time than RuleGrowth for large *window_size*, it generates much less rules. For example, running TRuleGrowth with *window_size* = 60 for the Snake dataset produces up to 50 times less rules than RuleGrowth. For memory usage, RuleGrowth generally uses slightly less memory than TRuleGrowth because this latter keeps track of all occurrences of each frequent item instead of just the first and last occurrences. But in some cases (e.g. *minsup* < 0.0015 on Kosarak), TRuleGrowth uses less memory. This is because RuleGrowth finds larger rules, and therefore RuleGrowth keeps more information in memory because it performs more levels of recursive calls to EXPANDRIGHT and EXPANDLEFT.

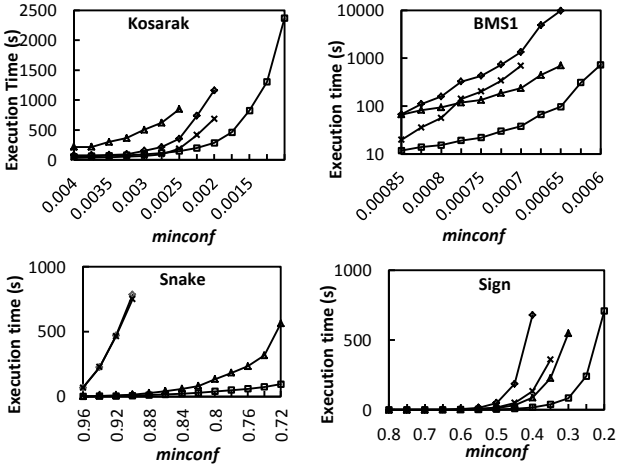

Fig. 9. Influence of *minconf* for Kosarak, BMS1, Snake and Sign

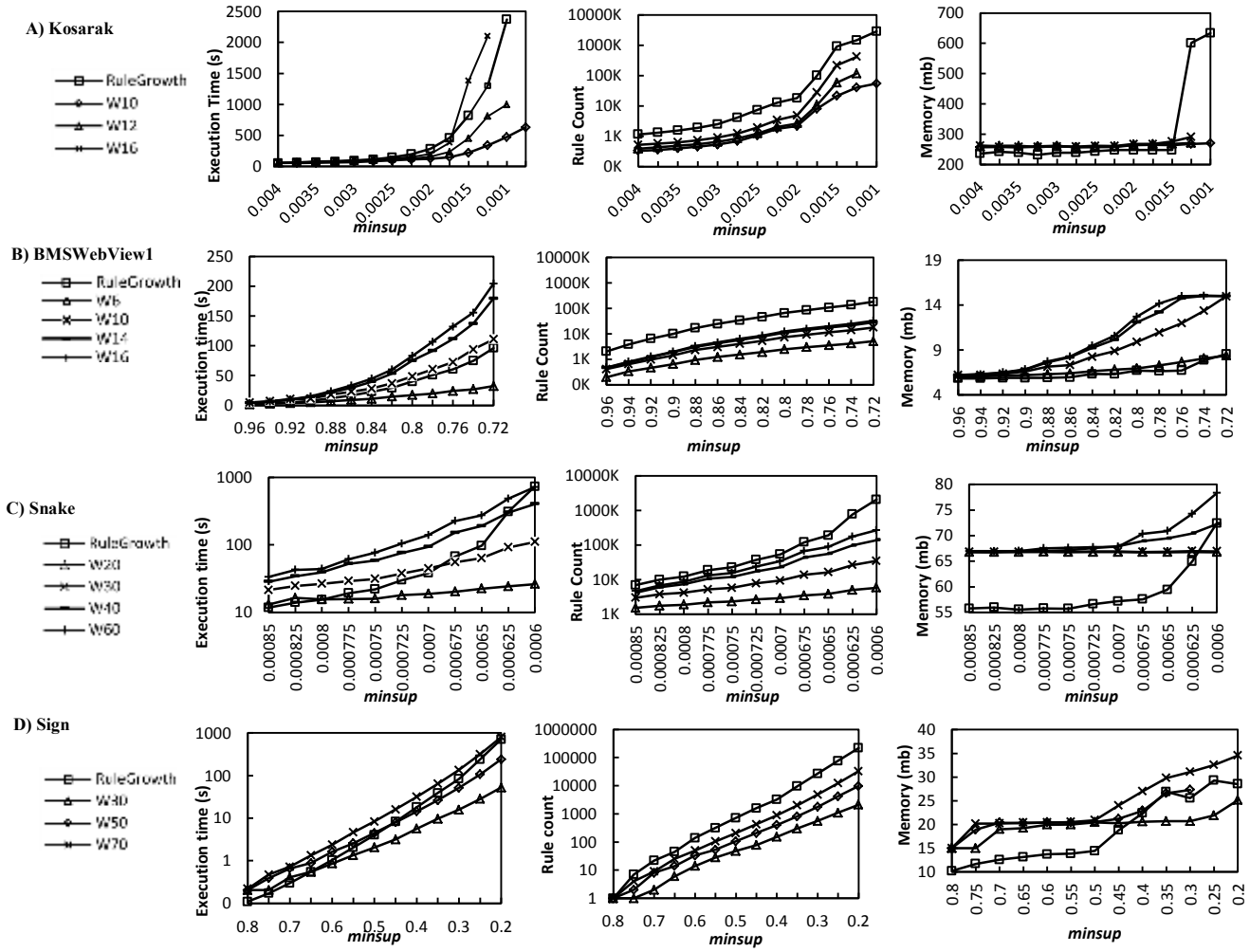

Figure 10. Influence of *window_size*

7.5 Experiment to assess the scalability of the algorithms

The fourth experiment assesses the scalability of CMRules, CMDeo, RuleGrowth and TRuleGrowth with respect to the number of sequences |S|. For this experiment, the original Kosarak dataset was used because it is a very large dataset containing 700,000 sequences, which is convenient for varying the size of the dataset easily. Snake, BMS1 and Sign were not used because they are much smaller than Kosarak. For the experiment, algorithms were run with *minsup* = 0.003 and *minconf* = 0.5, while |S| was varied from 10,000 to 200,000 with an increment of 10,000. For TRuleGrowth, *window_size* was set to 10. As for previous experiments, a maximum memory usage of 1 GB was set. Moreover, a time limit of 1,000 seconds was used. Results of the experiment are shown in Fig. 11. As it can be seen, CMRules, RuleGrowth and TRuleGrowth's execution time and maximum memory usage grow linearly with the size of |S|. CMDeo also shows a similar trend. However, as it approaches the memory limit of 1 GB, its performance is negatively affected by the Java garbage collector No results are available for |S|>100,000 for CMDeo as it exceeded the memory limit.

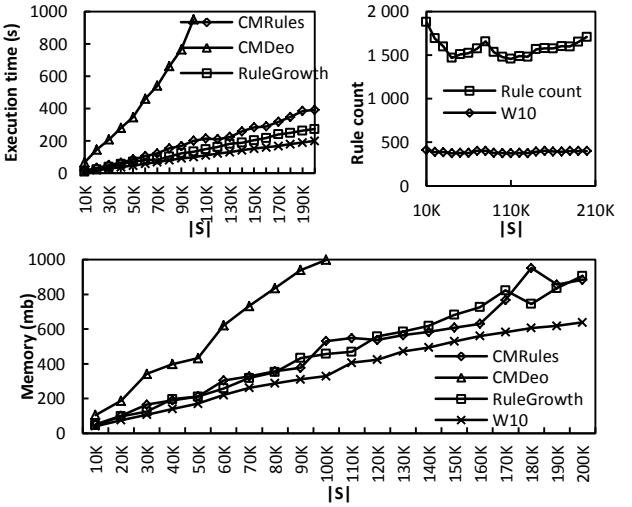

Fig. 11. Result of the scalability experiment with Kosarak

7.6 Performance analysis

The efficiency of RuleGrowth/TRuleGrowth can be analyzed as follows.

No candidate generation. RuleGrowth/TRuleGrowth discover rules by scanning sequences from the database

to grow rules. The algorithms do not test candidates not occurring in the database unlike CMRules and CMDeo.

Sid sets keep shrinking. It is easy to see that as rules grow, sid sets become smaller and less sequences need to be scanned. Sid sets generally reduce substantially as rules grow.

Complexity. RuleGrowth/TRuleGrowth are pseudo-polynomial algorithms. Their complexity is linear with respect to the number of sequential rules in a database, either one or two recursive calls are performed to EXPANDLEFT/EXPANDRIGHT for each sequential rule. The cost of each call is upper bounded by the time of scanning the database once (in the worst case), and counting the frequency of items.

7.7 Experiment to Assess Prediction Accuracy

The RuleGrowth/TRuleGrowth/CMRules algorithms have been applied successfully in e-learning [6, 7], manufacturing simulation [31], quality control [27], web page click-stream analysis [30] and anti-pattern detection in service based systems [32]. In the next paragraphs, we present results from the application of TRuleGrowth to webpage prefetching. In this application, we have compared prediction accuracy using (1) sequential rules (SR) mined by RuleGen [17] enhanced with a *window_size* constraint and (2) partially-ordered sequential rules (POSR) mined by TRuleGrowth. These experiments were carried with the Kosarak and BMS1 datasets, which are click-stream datasets. Note that we here only give a summary of the results. Full results about this experiment can be found in a dedicated publication [30]. For this experiment, each dataset was split in a training set and a testing set based on a *training_ratio* parameter. The training set was used to generate SR and POSR, respectively. Then, the testing set was used to test prediction accuracy using the rules. Each sequence from the test set was split into prefix and suffix parts based on some parameters named *prefix_size* and *suffix_size*. The task of prediction for a sequence was to predict the first item from the suffix using the information from the prefix. We measured the accuracy (number of good predictions divided by the size of the test set) and the coverage (number of sequences where it was possible to make a prediction). In this experiment, we tuned RuleGen and TRuleGrowth with the *minsup* and *minconf* values that provided the best results. We varied (1) *prefix_size*, (2) *suffix_size*, (3) *training_ratio* and (4) *window_size* to perform measurements.

Overall, we have observed that using POSR always provide a considerably higher accuracy and coverage (up to 30 % higher accuracy and up to 60% higher coverage), depending on the scenario. For example, when setting *minconf* = 0.5, *training_ratio* = 50%, *minsup* = 0.00055 (BMS1) and *minsup* = 0.002 (*Kosarak*), *prefix_size* = 3, *suffix_size* = *3* and *window_size* = 5, results were as follows. For BMS1, POSR provided about 25% accuracy / 95% coverage, while SR provided about 10% accuracy / 50% coverage. For Kosarak, POSR provided about 12 % accuracy / 50 % coverage and SR provided about 5% accuracy / 10% coverage. The reason why SR have poor coverage is that rules are too specific as highlighted in Section 1.

The experiment has also shown that using the *window_size constraint* is beneficial. For POSR, the best values of *window_size* were between 5 and 7 (BMS1) and 7 (Kosarak). For SR, the best values were 5 (BMS1) and 7 (Kosarak). Increasing *window_size* above these values did not improve accuracy but increased execution times.

Lastly, another interesting result is that using approximately 1,000 to 10,000 rules was enough to provide the best accuracy for both POSR and SR.

With this experiment, we have presented a real application where POSR provides a clear benefit over the use of SR, and where the *window_size* constraint is important.

7 Conclusion

This paper presented two algorithms. RuleGrowth is a novel algorithm for mining sequential rules common to multiple sequences. Unlike previous algorithms, it uses a pattern-growth approach for discovering valid rules such that it avoids considering rules not appearing in the database. The second algorithm (TRuleGrowth) allows the user to specify a sliding-window constraint on rules to be mined. To evaluate RuleGrowth and TRuleGrowth, we performed several experiments on four real-life datasets having different characteristics. First, the performance of RuleGrowth was compared with CMRules and CMDeo while varying the *minsup* and *minconf* parameters, to assess their influence on the performance of each algorithm. Second, RuleGrowth was compared to TRuleGrowth for different *window_size* values to evaluate the benefits of using the window size constraint. Experimental results show that RuleGrowth is up to several of magnitude faster and uses up to an order of magnitude less memory than CMRules and CMDeo. Moreover experiment show that the execution time and the number of valid rules found can be reduced by several orders of magnitude when the window size constraint is used. Lastly, we have reported results from a real application where using partially-ordered sequential rules and the window constraint greatly improves accuracy over sequential rules.

Acknowledgements

The authors thank the Fonds Québécois de la Recherche sur la Nature et les Technologies for its financial support.

References

[1] R. Agrawal, T. Imielminski and A. Swami, "Mining Association Rules Between Sets of Items in Large Databases," *Proc. 13th ACM SIGMOD Intern. Conf. on Management of Data*, pp. 207-216, 1993.

[2] R. Agrawal, R. Srikant, "Mining Sequential Patterns," *Proc. 11th Intern. Conf. on Data Eng.*, pp. 3-14, 1995.

[3] D.W. Cheung, J. Han, V. Ng. and Y. Wong, "Maintenance of discovered association rules in large databases: An incremental updating technique," *Proc. 12th Intern. Conf. on Data Eng.*, pp. 106-114, 1996.

[4] G. Das, K.-I. Lin, H. Mannila, G. Renganathan and P. Smyth, "Rule Discovery from Time Series," *Proc. 4th ACM Intern. Conf. Know. Discovery and Data Mining*, pp. 16-22, 1998.

[5] J.S. Deogun and L. Jiang, "Prediction Mining – An Approach to Mining Association Rules for Prediction," *Proc. 10th Intern. Conf. Rough Sets,*

Fuzzy Sets, Data Mining, and Granular Comp., pp. 98-108, 2005.

[6] U. Faghihi, P. Fournier-Viger and R. Nkambou, "A Computational Model for Causal Learning in Cognitive Agents," *Knowledge Based Systems*, vol. 30, pp. 48-56, 2012.

[7] P. Fournier-Viger, U. Faghihi, R. Nkambou and E. Mephu Nguifo, "CMRules: An Efficient Algorithm for Mining Sequential Rules Common to Several Sequences," *Knowledge Based Systems*, vol. 25, no. 1, pp. 63-76, 2012.

[8] J.H. Hamilton and K. Karimi, "The TIMERS II Algorithm for the Discovery of Causality," *Proc. 9th Pacific-Asia Conference on Knowledge Discovery and Data Mining*, pp. 744-750, 2005.

[9] S.K. Harms, J. Deogun and T. Tadesse, "Discovering Sequential Association Rules with Constraints and Time Lags in Multiple Sequences," *Proc. 13th Intern. Symp. Method. Intell. Systems*, pp. 373-376., 2002.

[10] I. Jonassen, J.F. Collins and D.G. Higgin, "Finding flexible patterns in unaligned protein sequences," *Protein Science*, vol. 4, no. 8, pp. 1587-1595, 1995.

[11] S. Laxman and P. Sastry, "A survey of temporal data mining," *Sadhana*, vol. 3, pp. 173-198, 2006.

[12] D. Lo, S.-C. Khoo and L. Wong, "Non-redundant sequential rules – Theory and algorithm," *Inform. Syst.*, vol. 34, no. 4-5, pp. 438-453, 2009.

[13] H. Mannila, H. Toivonen and A.I. Verkano, "Discovery of frequent episodes in event sequences," *Data Mining and Knowledge Discovery*, vol. 1, no. 3, pp. 259-289, 1999.

[14] J. Pei, J. Han et al., "Mining Sequential Patterns by Pattern-Growth: The PrefixSpan Approach," *IEEE Trans. Knowledge and Data Eng.*, vol. 16, no. 10, pp. 1-17, 2004.

[15] P. Fournier-Viger, Knowledge discovery in problem-solving learning activities, Ph.D. Thesis, Univ. Quebec in Montreal, Montreal, 2010.

[16] Y.L. Hsieh, D.-L. Yang and J. Wu, "Using Data Mining to Study Upstream and Downstream Causal Relationship in Stock Market," *Proc. 2006 Joint Conf. Inf. Sc.*, 2006.

[17] M. J. Zaki, "SPADE: An Efficient Algorithm for Mining Frequent Sequences," *Machine Learning*, vol. 42, no.1-2, pp. 31-60, 2001.

[18] P. Fournier-Viger, A. Gomariz, M. Campos and R. Thomas, "Fast Vertical Sequential Pattern Mining Using Co-occurrence Information," *Proc. 18th Pacific-Asia Conference on Knowledge Discovery and Data Mining*, Springer, pp. 40-52, 2014.

[19] Y. Zhao, H. Zhang, L. Cao, C. Zhang and H. Bohlscheid, "Mining Both Positive and Negative Impact-Oriented Sequential Rules From Transactional Data," *Proc. 13th Pacific-Asia Conference on Knowledge Discovery and Data Mining*, Springer, pp. 656-663, 2009.

[20] P. Fournier-Viger and V.S. Tseng, "TNS: Mining Top-K Non-Redundant Sequential Rules," *Proc. 28th Symposium on Applied Computing*, ACM Press, pp. 164-166, 2013.

[21] P. Fournier-Viger, R. Nkambou and V. S. Tseng, "RuleGrowth: Mining Sequential Rules Common to Several Sequences by Pattern-Growth," *Proc. 26th ACM Symp. Applied Computing*, pp. 954-959, 2011.

[22] A. Pitman and M. Zanker, "An Empirical Study of Extracting Multidimensional Sequential Rules for Personalization and Recommendation in Online Commerce," *Proc. Wirtschaftinformatik 2011*, pp. 180-189, 2011.

[23] P. Papapetrou, G. Kollios, S. Sclaroff and D. Gunopulos, "Discovering Frequent Arrangements of Temporal Intervals," *Proc. of 5th IEEE International Conference on Data Mining*, pp. 354-361, 2005.

[24] J. Ayres, J. Flannick, J. Gehrke and T. Yiu, "Sequential PAttern mining using a bitmap representation," *Proc. 8th ACM Intern. Conf. Know. Discovery and Data Mining* (*KDD'02*), pp. 429-435, 2002.

[25] H. Minqing and B. Liu, "Opinion Feature Extraction Using Class Sequential Rules," *Proc. AAAI Spring Symp. on Computational Approaches to Analyzing Weblogs*, Palo Alto, USA, March 2006.

[26] J.E. McDunn, K.D. Husain, A.D. Polpitiya, A. Burykin and J. Huan, "Plasticity of the systemic inflammatory response to acute infection during critical illness: development of the riboleukogram," *PLoS One*, vol. 13, no. 2, e1564, 2008.

[27] T. Bogon, I.J. Timm, A.D. Lattner, D. Paraskevopoulos, U. Jessen, M. Schmitz, S. Wenzel, S. Spieckermann, "Towards Assisted Input and Output Data Analysis in Manufacturing Simulation: The EDASIM Approach," *Proc. 2012 Winter Simulation Conference*, pp. 257-269, 2012.

[28] M.A. Sartor, V. Mahavisno, V. G. Keshamouni, J. Cavalcoli et al., "ConceptGen: a gene set enrichment and gene set relation mapping tool," *Bioinformatics*, vol. 26, no. 4, pp. 456-463, 2010.

[29] D. Lo, G. Ramalingam, V. P. Ranganath and K. Vaswani, "Mining Quantified Temporal Rules: Formalism, Algorithms, and Evaluation," *Proc. 16th Working Conference on Reverse Engineering*, pp. 62-71, 2009.

[30] P. Fournier-Viger, T. Gueniche and V.S. Tseng, "Using Partially-Ordered Sequential Rules to Generate More Accurate Sequence Prediction," *Proc. 8th International Conference on Advanced Data Mining and Applications*, Springer, pp. 431-442, 2012.

[31] B. Kamsu-Foguem, F. Rigal and F. Mauget, "Mining association rules for the quality improvement of the production process," *Expert Systems and Applications*, vol. 40, no. 4, pp. 1034-1045, 2013.

[32] M. Nayrolles, N. Moha, P. Valtchev, "Improving SOA antipatterns detection in Service Based Systems by mining execution traces," *Proc. 20th IEEE Working Conference on Reverse Engineering*, pp. 321-330, 2013.

[33] P. Fournier-Viger, A. Gomariz, A. Soltani, T. Gueniche, C.W. Wu. and V.S. Tseng, "SPMF: a Java Open-Source Pattern Mining Library," *Journal of Machine Learning Research*, vol. 15, pp. 3389-3393, 2014.

Philippe Fournier-Viger (Ph.D.) is an assistant-professor at University of Moncton, Canada. He received a Ph.D. in Cognitive Computer Science at the University of Quebec in Montreal (2010). He has published more than 70 research papers in refereed international conferences and journals. His research interests include data mining, pattern mining, text mining, intelligent tutoring systems, knowledge representation and cognitive modeling. He is the founder of the popular SPMF open-source data mining library.

Cheng Wei-Wu received the M.Ss. degree in computer science and information engineering from Ming Chuan University, Taiwan, R.O.C., in 2009. He is currently pursuing the Ph.D. degree in the Department of Computer Science and Information Engineering, National Cheng Kung University, Taiwan, R.O.C.

Vincent S. Tseng (Ph.D) is a professor at National Cheng Kung University (NCKU), Taiwan, ROC. Dr. Tseng received his Ph.D. degree from National Chiao Tung University (1997). He has a wide variety of research interests covering data mining, biomedical informatics, multimedia databases, mobile and Web technologies. He has published more than 200 research papers in referred journals and international conferences and also held (or filed), more than 15 patents.

Longbing Cao (Ph.D) is a Professor at the University of Technology Sydney, and the Data Mining Research Leader of the Australian Capital Markets Cooperative Research Centre. He got one PhD in Intelligent Sciences and another in Computing Sciences. His research interests include data mining and machine learning and their applications, behavior informatics, multi-agent technology, open complex intelligent systems, and agent mining.

Roger Nkambou (Ph.D) is a Professor of Computer Science at the University of Quebec at Montreal, and Director of the Graduate Program in Cognitive Computing (http://dic.uqam.ca). He received his Ph.D. (1996) in Computer Science from the University of Montreal. His research interests include knowledge representation, intelligent tutoring systems, intelligent software agents, ontology engineering, student modeling and affective computing.