Bidirectional Mining of Non-Redundant Recurrent Rules from a Sequence Database

David Lo[1], Bolin Ding[2], Lucia[1], Jiawei Han[2]
[1]School of Information Systems, Singapore Management University
[2]Department of Computer Science, University of Illinois at Urbana-Champaign
Emails: davidlo@smu.edu.sg,bding3@uiuc.edu,lucia.2009@phdis.smu.edu.sg,hanj@uiuc.edu

Abstract—**We are interested in scalable mining of a non-redundant set of significant recurrent rules from a sequence database. Recurrent rules have the form "whenever a series of precedent events occurs, eventually a series of consequent events occurs". They are intuitive and characterize behaviors in many domains. An example is the domain of software specification, in which the rules capture a family of properties beneficial to program verification and bug detection. We enhance a past work on mining recurrent rules by Lo, Khoo, and Liu to perform mining more scalably. We propose a new set of pruning properties embedded in a new mining algorithm. Performance and case studies on benchmark synthetic and real datasets show that our approach is much more efficient and outperforms the state-of-the-art approach in mining recurrent rules by up to two orders of magnitude.**

I. Introduction

Data mining has been shown useful in various domains including finance, marketing, bioinformatics and recently software engineering, e.g., [11], [17]. Many valuable data sources are in sequential formats ranging from logs, transaction histories, medical histories, program traces, and many more.

To analyze sequential data, Lo *et al.* propose recurrent rules stating: "Whenever a series of events occurs, eventually another series of events occurs". Recurrent rules capture temporal constraints that repeat a substantial number of times both *within a sequence* and *across multiple sequences* [18]. The rule format is general and is *not limited by look-ahead limit or window-width constraint*. This enables the rule to capture significant candidates for not only temporal short-distance but also long-distance cause-and-effect relationships in a dataset. There are many instance of recurrent rules in day-to-day settings:

1. Resource Locking Protocol: Whenever a lock is acquired, eventually it is released.
2. Internet Banking: Whenever a connection is made and an authentication is completed and fund transfer command is issued, eventually the fund is transferred.
3. Network Protocol: Whenever an HDLC connection is made and an acknowledgement is received, eventually a disconnection message is sent and an acknowledgement is received.

Zooming into the domain of software specification and verification, recurrent rules correspond to a family of program properties useful for program verification (*c.f.*, [9]). The first example given above is a program property. Research in software verification addresses approaches to check the correctness of a software with respect to a formal specification which often corresponds to a set of properties (*c.f.*, [6]). However, documented specifications might often be outdated or missing due to software evolution, reluctance in writing formal specifications and short-time-to-market cycle of software development (*c.f.*, [7], [5]).

There are past studies on mining association rules from sets of items [2]. These studies are later extended to mine for sequential patterns and episodes that consider ordering among events in sequences [3], [20]. Rules can be formed from both sequential patterns and episodes [24], [20]. Different from a pattern, a rule expresses a *constraint* involving its premise (*i.e.*, pre-condition) and consequent (*i.e.*, post-condition). These constraints are needed for their potential usages in filtering erroneous sequences, detecting outliers, etc.

However, rules generated from sequential patterns and episodes have different semantics from recurrent rules. A sequential rule $pre \rightarrow post$ states: "Whenever a sequence is a super-sequence of pre it will also be a super-sequence of pre concatenated with $post$". An episode rule $pre \rightarrow post$ states: "Whenever a window is a super-sequence of pre it will also be a super-sequence of pre concatenated with $post$".

To illustrate the differences, consider the following sequences:

Seq ID.	**Sequence**
$S1$	$\langle$`lock`, `use`, `use`, `use`, `use`, `unlock`$\rangle$
$S2$	$\langle$`lock`, `use`, `use`, `use`, `use`, `unlock`, `lock`, `lock`, `exit`$\rangle$

From sequence $S1$, episode rule mining with a window size of two is not able to mine the rule: "$lock$ must be followed by $unlock$" as the two events are *separated by more than two events*. From sequences $S1$ and $S2$, sequential rule mining would report "$lock$ must be followed by $unlock$ with a *perfect* confidence or likelihood (2 out of 2 cases)" despite the last two $lock$ operations in $S2$ are not paired with any subsequent $unlock$ operations. Considering both $S1$ and $S2$, recurrent rule mining would report "$lock$ is followed by $unlock$ in 2 out of 4 cases".

Recurrent rules generalize sequential rules where for each rule, *multiple* occurrences of the rule's premise and consequent both *within a sequence and across multiple sequences* are considered. Recurrent rules generalize episode rules by allowing precedent and consequent events to be separated by an *arbitrary* number of events in a *sequence database*. Also, a set of sequences rather than a single sequence is considered during mining. Furthermore, rather than mining all rules we mine a *non-redundant* set of rules.

Recurrent rules could be formalized in Linear Temporal Logic (LTL) [14]. LTL has been widely studied and many tools have been built on top of it, e.g., [6], [16]. Model checker is one of them [6]. It allows verifying correctness of safety critical systems based on the satisfaction of rules and properties formalized in LTL.

The new semantics of recurrent rules, as compared to association, sequential and episode rules, necessitates new pruning strategies and algorithms that utilize these strategies to efficiently mine for recurrent rules.

The pioneer work on recurrent rules is a past study by Lo, Khoo and Liu in [18]. However, with the growth in the size of data currently available, there is *a need for a more scalable algorithm*. On larger datasets or lower support thresholds, we find that there is a need to enhance the scalability of the algorithm in [18]. This work fills this gap by proposing a more scalable algorithm that embeds new pruning properties to mine recurrent rules more efficiently.

Our approach works in several steps: 1. Mining pruned pre-conditions, 2. Mining pruned post-conditions, 3. Rule formation, and 4. Redundant rules removal. A number of new pruning strategies and a new data structure are employed for efficient mining of a non-redundant set of recurrent rules. Under a condition which holds in many cases, the complexity of the proposed algorithm is smaller by an exponential factor than the complexity of the one proposed in [18] (see Section VI-D). Performance study conducted on benchmark datasets, both synthetic and real, shows that runtime is improved by up to 134 times. We also conducted a case study on a dataset of real software traces extracted from multiple user interactions with an instant messaging application.

The outline of this paper is as follows. In Section II, we discuss related work. Section III presents recurrent rule semantics in Linear Temporal Logic (LTL). Section IV describes some notations and definitions used in subsequent sections. Section V presents various theorems and properties used to prune search spaces. Section VI presents our algorithm and compares it with [18]. Section VII describes our performance study and case study. Finally, we conclude and describe future work in Section VIII.

II. Related Work

In this section, we discuss closely related past studies and compare them with our approach.

Past Studies on Temporal Logics. In this work we mine recurrent rules, which is under the family of Linear Temporal Logic properties [14]. Temporal logics itself has been widely used for various purposes ranging from specifying communication protocols among agents or models of agents' behaviors [15], modeling bio-molecular interactions [21], querying XML documents [4], supporting historical databases [28], verifying correctness of systems [6], etc. Many studies described above *use* temporal logics to accomplish a particular task. Different from these past studies, our goal is to *mine* temporal logic expressions automatically from datasets.

Mining Frequent Itemsets & Association Rules. Association rule mining is first proposed by Agrawal and Srikant in [2]. Association rule captures a relationship among items in a set, where *the ordering of items is irrelevant*. Association rules are generated by post-processing frequent itemsets. Notions of support and confidence are used as measures to distinguish significant rules. There are many work extending association rule mining, of special interest are work on closed frequent itemsets (*e.g.*, [22]) and non-redundant association rules (*e.g.*, [27]). Different from association rules, recurrent rules express *ordering constraints among events in a sequence database*. Due to this difference, pruning properties, suitable mining algorithms and the notion of rule redundancy are different.

Mining Patterns & Rules from Sequences. Sequential pattern mining [3], [23] discovers patterns that are supported by a *significant number of sequences*. A pattern is supported by a sequence if it is a sub-sequence of it. To remove redundant patterns, closed sequential pattern mining was proposed by Yan *et al.* [26] and improved by Wang and Han [25]. Spiliopoulou proposed the generation of sequential rules from sequential patterns [24]. Recurrent rules generalize sequential rules by considering *multiple occurrences* of the premise and consequent events within a sequence and across multiple sequences in the database. This generalization is significant since the precedent and consequent events of a rule can potentially appear repeatedly in a sequence. Considering program execution traces, due to loops and recursion, it is common to see program properties observed repeatedly in a trace.

Mannila *et al.* performed episode mining to discover *frequent episodes within a sequence of events* [20]. An episode is defined as a series of events occurring *relatively close* to one another (*i.e.* they occur at the same window). An episode is supported by a window if it is a sub-sequence of the series of events appearing in the window. There are many extensions of the work. Harms et al. mine for constrained episode rules where the distance between the precedent and consequent of a rule is further limited by a value smaller than the overall window size [12]. Garriga extends Mannila *et al.*'s work to replace a fixed-window size with a gap constraint between one event to the next in an episode [10]. Recurrent rules generalize rules formed from episodes by allowing precedent and consequent events to be separated by an arbitrary number of events. This generalization is significant since precedent and consequent events might possibly be separated by an arbitrary number of events in sequences. For example, useful program properties, such as: 'acquiring of a lock ($lock$) is eventually followed by its release ($unlock$)', or 'an opened file (*open*) is eventually closed ($closed$)' (*c.f* [17], [18]), often have their associated events occur at some arbitrary distance away from one another in an execution trace. Also, we analyze a *sequence database* and mine *non-redundant* rules.

In [17], Lo et al. proposed iterative patterns to discover

TABLE I
LTL EXPRESSIONS AND THEIR MEANINGS

$F(unlock)$ Meaning: Eventually *unlock* is called
$XF(unlock)$ Meaning: From the next event onwards, eventually *unlock* is called
$G(lock \rightarrow XF(unlock))$ Meaning: Globally whenever *lock* is called, then from the next event onwards, eventually *unlock* is called
$G(main \rightarrow XG(lock \rightarrow (\rightarrow XF(unlock \rightarrow XF(end)))))$ Meaning: Globally whenever *main* followed by *lock* are called, then from the next event onwards, eventually *unlock* followed by *end* are called

TABLE II
RULES AND THEIR LTL EQUIVALENCES

Notation	LTL Notation
$a \rightarrow b$	$G(a \rightarrow XFb)$
$\langle a,b\rangle \rightarrow c$	$G(a \rightarrow XG(b \rightarrow XFc))$
$a \rightarrow \langle b,c\rangle$	$G(a \rightarrow XF(b \wedge XFc))$
$\langle a,b\rangle \rightarrow \langle c,d\rangle$	$G(a \rightarrow XG(b \rightarrow XF(c \wedge XFd)))$

software specifications, which are defined based on the semantics of Message Sequence Charts (MSC). In [8], Ding et al. proposed an approach to mine for repetitive sub-sequences. Our work could be viewed as an extension of their approaches to mine for repetitive rules following the semantics of Linear Temporal Logics. In the software domain, LTL (but not MSC) is one of the most widely-used formalism for program verification (*i.e.*, ensuring correctness of a software system) [6]. Since the underlying target formalisms and semantics are different, both the search space pruning strategies and the mining algorithm needed to efficiently mine recurrent rules are very different from those used in the past studies in [17] and [8].

Mining recurrent rules is first proposed in [18]. In this work, we speed up the mining process further. We propose new pruning strategies and a new algorithm that embed the strategies into an effective approach to prune search space. Under a condition which holds in many cases, the complexity of our approach is exponentially better than the complexity of the approach in [18] (see Section VI-D). Furthermore, our empirical evaluation shows that our approach is able to outperform [18] by up to 2 orders of magnitude.

III. RULE SEMANTICS IN LTL

Our mined rules can be expressed in Linear Temporal Logic (LTL) [14]. LTL is a logic that specifies properties of a sequence (i.e., a series of events).

There are a number of LTL operators, among which we are only interested in the operators: 'G','F', and 'X'. The operator 'G' specifies that *globally* for every event in a sequence a certain property holds. The operator 'F' specifies that a property holds either at the current event or *finally* (or eventually) in one of the subsequent events in a sequence. The operator 'X' specifies that a property holds at the *next* event. Some examples are listed in Table I.

Our mined rules state whenever a series of precedent events occurs eventually another series of consequent events also occurs. A mined rule denoted as $pre \rightarrow post$, can be mapped to its corresponding LTL expression. Examples of such correspondences are shown in Table II. Note that although the operator 'X' might seem redundant, it is needed to specify rules such as $\langle a\rangle{\rightarrow}\langle b,b\rangle$ where the 'b's refer to *different occurrences of 'b'*.

The set of LTL expressions corresponding to the set of recurrent rules and are minable by our mining framework is shown in Backus-Naur Form (BNF) as follows:

$$\boxed{\begin{array}{rl} rules := & G(prepost) \\ prepost := & event \rightarrow post | event \rightarrow XG(prepost) \\ post := & XF(event) | XF(event \wedge XF(post)) \end{array}}$$

Example 1: To illustrate the semantics of recurrent rules, consider a sequence:

$$S = \langle main, lock, use, unlock, lock, end\rangle$$

The LTL property corresponding to the rule $\langle main, lock\rangle \rightarrow \langle unlock, end\rangle$ is the property $G(main \rightarrow XG(lock \rightarrow XF(unlock \wedge XF(end))))$. This property is violated by S as the second occurrence of $lock$ is not eventually followed by $unlock$ and end. Note however, the property corresponding to $\langle main, lock, use\rangle \rightarrow \langle unlock, end\rangle$, which is $G(main \rightarrow XG(lock \rightarrow XG(use \rightarrow XF(unlock \wedge XF(end)))))$ is satisfied. This is the case since the second occurrence of $lock$ immediately before end in S is not followed by a use.

IV. NOTATIONS & DEFINITIONS

This section presents some preliminary notations and definitions pertinent to mining recurrent rules. Many of these are taken from [18].

A. Basic Notations

Let I be a set of distinct *events* considered. The input to our mining problem is a *sequence database* denoted as $SeqDB = \{S_1, S_2, \ldots, S_{|SeqDB|}\}$. Each sequence is an ordered list of events, denoted as $\langle e_1, e_2, \ldots, e_{end}\rangle$ where $e_i \in I$.

We define a *pattern* P to be a series of events. We use *first*(P) and *last*(P) to denote the first and last event of P respectively. A pattern $P_1{+\!\!+}P_2$ denotes the concatenation of patterns P_1 and P_2, also said to be the *forward extension* of P_1 or *backward extension* of P_2. A pattern $P_1 = \langle e_1, e_2, \ldots, e_n\rangle$ is a *subsequence* (*sub-pattern*) of another pattern $P_2 = \langle e'_1, e'_2, \ldots, e'_m\rangle$ (or, P_2 is a *super-pattern* of P_1) if there exists integers $1 \leq i_1 < i_2 < \ldots < i_n \leq m$ such that $e_1 = e'_{i_1}$, $e_2 = e'_{i_2}$, $\cdots$, $e_n = e'_{i_n}$ (denoted as $P_1 \sqsubseteq P_2$, or $P_1 \sqsubset P_2$ if they are not equal).

Each *recurrent rule* R has the form $R_{pre} \rightarrow R_{post}$, where R_{pre} and R_{post} are two series of events (*i.e.*, two patterns). R_{pre} is referred to as the *premise* or *pre-condition* of the rule, while R_{post} is referred to as the *consequent* or *post-condition* of R.

B. Concepts and Problem Statement

In this subsection, we discuss various concepts and definitions and define our problem statement. We use the database in Table III as our running example throughout this paper.

Each recurrent rule we mine expresses:

TABLE III
EXAMPLE SEQUENCE DATABASE – *SeqDB*

Seq ID.	Sequence
S_1	$\langle \texttt{a}, \texttt{b}, \texttt{e}, \texttt{c}, \texttt{b}, \texttt{d}, \texttt{c}, \texttt{d} \rangle$
S_2	$\langle \texttt{a}, \texttt{c}, \texttt{b}, \texttt{e}, \texttt{a}, \texttt{e}, \texttt{d}, \texttt{c}, \texttt{b}, \texttt{d} \rangle$

"Whenever a series of events *has just occurred at a point in time (i.e. a temporal point)*, eventually another series of events occurs"

From the above semantics, to generate recurrent rules, we need to "peek" at interesting temporal points and "see" what series of events are likely to occur next. We will first formalize the notion of temporal points and occurrences.

*Definition 4.1 (***Temporal Points & Prefixes & Suffixes***):* Consider a sequence S of the form $\langle a_1, a_2, \ldots, a_n \rangle$. All events in S are indexed by their positions in S, starting from 1 (e.g., $S[2] = a_2$). These positions are called *temporal points in* S. For a temporal point j, the first j events $\langle a_1, \ldots, a_j \rangle$ is the j*-prefix* of S; the last j events $\langle a_{n-j+1}, \ldots, a_n \rangle$ is the j*-suffix* of S.

*Definition 4.2 (***Occurrences & Instances***):* Given a pattern P and a sequence S, an *occurrence* of P in S is defined to be a temporal point j, such that the j-prefix of S is a super-sequence of P and $last(P)$=$S[j]$.

Each of such j-prefixes is said to be an *instance* of the pattern P in S, *i.e.*, it is a super-sequence of P and $last(P)$ is indexed by j. An instance of P in S is *minimum* iff it is the shortest one, *i.e.*, there is no $k < j$ such that k-prefix of S is an instance of P.

Example 2: Consider a pattern P $\langle a, b \rangle$ and the sequence S_1 in the example database in Table III (*i.e.*, $\langle \texttt{a}, \texttt{b}, \texttt{e}, \texttt{c}, \texttt{b}, \texttt{d}, \texttt{c}, \texttt{d} \rangle$). Temporal points $\{2, 5\}$ are the *occurrences* of P in S_1, and the corresponding *instances* are $\langle \texttt{a}, \texttt{b} \rangle$ and $\langle \texttt{a}, \texttt{b}, \texttt{e}, \texttt{c}, \texttt{b} \rangle$, where the instance $\langle \texttt{a}, \texttt{b} \rangle$ is the minimum one.

Definition 4.3 is a standard database projection (*c.f.* [26], [25]) capturing series of events occurring after the first *temporal point*. Definition 4.4 is a new type of projection capturing series of events occurring after *each temporal point* introduced in [18].

*Definition 4.3 (***Projection & Sequence Support***):* A database *projected* on a pattern P is defined as:

$SeqDB_P$ = $\{(i, sx) \mid S_i \in SeqDB$, satisfying $S_i = px{+\!\!+}sx$, and px is the minimum instance of P in S_i, *i.e.*, px is the shortest prefix of S_i containing $P\}$.

For a pattern P, we define the *sequence support* $sup(P{,}SeqDB)$ to be the size of $SeqDB_P$ (or equivalently, the number of sequences in $SeqDB$ containing P). Reference to the database is omitted if it is clear from the context.

*Definition 4.4 (***All-Projection & Instance Support***):* A database *all-projected* on a pattern P is defined as:

$SeqDB_P^{all}$ = $\{(i, sx) \mid S_i \in SeqDB$, satisfying $S_i = px{+\!\!+}sx$, and px is an instance of P in $S_i\}$.

For a pattern P, we define the *instance support* $sup^{all}(P, SeqDB)$ to be the size of $SeqDB_P^{all}$ (or equivalently, the total number of instances of P in all sequences of $SeqDB$). Reference to the database is omitted if it is clear from the context.

Example 3: To illustrate the above concepts, we list the projected and all-projected database $SeqDB$ w.r.t. $\langle \texttt{a}, \texttt{b} \rangle$ in Table III. From the support notations, we have $sup(\langle \texttt{a}, \texttt{b} \rangle, SeqDB)$ $= |SeqDB_{\langle \texttt{a}, \texttt{b} \rangle}| = 2$, $sup^{all}(\langle \texttt{a}, \texttt{b} \rangle, SeqDB) = |SeqDB^{all}_{\langle \texttt{a}, \texttt{b} \rangle}|$ $= 4$.

TABLE IV
PROJECTION AND ALL-PROJECTION

Projection $SeqDB_{\langle \texttt{a}, \texttt{b} \rangle}$
$(1, \langle \texttt{e}, \texttt{c}, \texttt{b}, \texttt{d}, \texttt{c}, \texttt{d} \rangle)$
$(2, \langle \texttt{e}, \texttt{a}, \texttt{e}, \texttt{d}, \texttt{c}, \texttt{b}, \texttt{d} \rangle)$

All-Projection $SeqDB^{all}_{\langle \texttt{a}, \texttt{b} \rangle}$
$(1, \langle \texttt{e}, \texttt{c}, \texttt{b}, \texttt{d}, \texttt{c}, \texttt{d} \rangle)$
$(1, \langle \texttt{d}, \texttt{c}, \texttt{d} \rangle)$
$(2, \langle \texttt{e}, \texttt{a}, \texttt{e}, \texttt{d}, \texttt{c}, \texttt{b}, \texttt{d} \rangle)$
$(2, \langle \texttt{d} \rangle)$

The two projection methods associated notions of sup and sup^{all} are different. Specifically, sup^{all} reflects the number of occurrences of P in $SeqDB$ rather than the number of sequences in $SeqDB$ supporting P. They would differ a lot if sequences in $SeqDB$ are long, and P repeats multiple times in single sequences.

Each *recurrent rule* R, in the form of $R_{pre} \rightarrow R_{post}$, where R_{pre} (*pre-condition*) and R_{post} (*post-condition*) are two series of events (patterns), expresses that: whenever R_{pre} occurs at a temporal point, R_{post} likely occurs (after R_{pre}). From the above notions of temporal points, projected databases and supports of patterns, we can now define the *support* and *confidence* of R.

*Definition 4.5 (***Support & Confidence of Rule***):* Consider a recurrent rule R ($R_{pre}{\rightarrow}R_{post}$) in $SeqDB$. The *sequence support* (or *instance support*) of R, denoted as $sup(R, SeqDB)$ (or $sup^{all}(R, SeqDB)$), is defined to be the sequence support (or instance support) of $R_{pre}{+\!\!+}R_{post}$ in $SeqDB$[1]:

$$sup(R, SeqDB) = sup(R_{pre}{+\!\!+}R_{post}, SeqDB),$$
$$sup^{all}(R, SeqDB) = sup^{all}(R_{pre}{+\!\!+}R_{post}, SeqDB).$$

The *confidence* of R, denoted as $conf(R, SeqDB)$, is defined to be the ratio:

$$conf(R, SeqDB) = \frac{sup(R_{post}, SeqDB^{all}_{R_{pre}})}{sup^{all}(R_{pre}, SeqDB)}.$$

$SeqDB$ is omitted if it is clear from the context.

The confidence of R, $conf(R, SeqDB)$, can be interpreted as the likelihood of R_{post} happening after R_{pre}. It is defined to be the ratio of two quantities: $sup^{all}(R_{pre}, SeqDB)$– the number of times that R_{pre} occurs in $SeqDB$, and $sup(R_{post}, SeqDB^{all}_{R_{pre}})$–the number of times that R_{post} occurs after R_{pre} in $SeqDB$.

Example 4: Consider a sequence database $SeqDB$ in Table III and a recurrent rule R, $\langle \texttt{a}, \texttt{b} \rangle \rightarrow \langle \texttt{c}, \texttt{d} \rangle$. The sequence

[1]We standardize definitions of $sup(R, SeqDB)$ and $sup^{all}(R, SeqDB)$. In [18], $sup(R, SeqDB)$ is defined as $sup(R_{pre}, SeqDB)$ while $sup^{all}(R, SeqDB)$ is defined as $sup^{all}(R_{pre}{+\!\!+}R_{post}, SeqDB)$. The current definition of sup follows typical definitions of support in association rule mining and sequential rule mining.

support of R is the number of sequences in $SeqDB$ containing (or being a super-sequence of) the concatenation of pre-condition and post-condition, *i.e.*, $\langle \mathtt{a}, \mathtt{b}, \mathtt{c}, \mathtt{d} \rangle$. Both S_1 and S_2 (in Table III) contain it, so $sup(R) = 2$. The instance support of R is the number of *occurrences* of pattern $\langle \mathtt{a}, \mathtt{b}, \mathtt{c}, \mathtt{d} \rangle$ in $SeqDB$, *i.e.*, $sup^{all}(R) = sup^{all}(\langle \mathtt{a}, \mathtt{b}, \mathtt{c}, \mathtt{d} \rangle) = 3$.

The confidence of the rule R ($\langle \mathtt{a}, \mathtt{b} \rangle \rightarrow \langle \mathtt{c}, \mathtt{d} \rangle$) is the likelihood of $\langle \mathtt{c}, \mathtt{d} \rangle$ occurring after each *temporal point* of $\langle \mathtt{a}, \mathtt{b} \rangle$. Refer to $SeqDB^{all}_{\langle \mathtt{a},\mathtt{b} \rangle}$ in Table IV, there is an occurrence of $\langle \mathtt{c}, \mathtt{d} \rangle$ after each of the first three occurrences of $\langle \mathtt{a}, \mathtt{b} \rangle$ (*i.e.*, $sup^{all}(\langle \mathtt{a}, \mathtt{b} \rangle, SeqDB) = 4$ and $sup(\langle \mathtt{c}, \mathtt{d} \rangle, SeqDB^{all}_{\langle \mathtt{a},\mathtt{b} \rangle}) = 3$). So, $conf(R) = 3/4$.

Mining Recurrent Rules. We aim to mine *significant rules* (with sufficient supports and confidence). Formally, in a sequence database $SeqDB$, given threshold min_sup for sequence support, threshold min_sup^{all} for instance support, and threshold min_conf for confidence, our goal is to find all rules R in forms of $R_{pre} \rightarrow R_{post}$ s.t. $sup(R, SeqDB) \geq min\_sup$, $sup^{all}(R, SeqDB) \geq min\_sup^{all}$, and $conf(R, SeqDB) \geq min\_conf$ (*significant rule*).

To reduce the number of rules and improve efficiency (while not to lose much information), in [18], Lo *et al.* define a notion of rule redundancy based on *super-sequence relationship* among rules having the same support and confidence values. This is similar to the notion of *closed* patterns applied to sequential patterns [26], [25].

*Definition 4.6 (***Rule Redundancy***):* A rule $R = R_{pre} \rightarrow R_{post}$ is *redundant* if there is another rule $R' = R'_{pre} \rightarrow R'_{post}$ such that:
1) R is a sub-sequence of R' ($R_{pre}{+\!\!+}R_{post} \sqsubset R'_{pre}{+\!\!+}R'_{post}$);
2) both rules have the same support and confidence values;
3) in the case that the concatenations are the same ($R_{pre}{+\!\!+}R_{post} = R'_{pre}{+\!\!+}R'_{post}$) and 2) holds, to break the tie, we define the one with longer pre-condition as being redundant (*i.e.*, we wish to retain the rule with a shorter pre-condition and a longer post-condition).

Example 5: Following Example 4, a rule R': $\langle \mathtt{a}, \mathtt{b} \rangle \rightarrow \langle \mathtt{c} \rangle$ is redundant in $SeqDB$, because another rule R: $\langle \mathtt{a}, \mathtt{b} \rangle \rightarrow \langle \mathtt{c}, \mathtt{d} \rangle$ has the same (sequence & instance) supports and confidence as $R_{X'}$.

In this work, we mine non-redundant significant rules.

V. Pruning Properties for Scalable Mining of Recurrent Rules

The basic idea of our algorithm for mining non-redundant recurrent rules is: first, we find a candidate set of pre-conditions, and a candidate set of post-conditions; then, we pair two elements, one from each candidate set, to form a recurrent rules. For the scalable mining of rules, we study the anti-monotonicity property of confidence used to speed up the pairing procedure, and propose strategies for shrinking the two candidate sets and pruning redundant rules in advance.

This section is organized as follows. The anti-monotonicity property of confidence and pruning strategies rely on two other types of projections, called *prefix projection* and *suffix projection*, which are introduced in Section V-A. The anti-monotonicity property of confidence is presented in Section V-B. The strategies for shrinking candidate sets and pruning redundant rules are described in Section V-C. The description of the complete algorithm and its analysis are deferred to Section VI.

A. Suffix Projection and Prefix Projection

Recall *projection* (Definition 4.3) and *all-projection* (Definition 4.4) essentially project sequences to their suffixes after the instances of a pattern. Differently, here we introduce *suffix projection* (*suf-projection*) and *prefix projection* (*pre-projection*), which project sequences to the minimum suffixes containing a pattern and the complementary prefixes, respectively. Using the two new types of projections, we will introduce the anti-monotonicity property of confidence (Section V-B) and the strategies for pruning redundant rules (Section V-C).

*Definition 5.1 (***Suffix Projection and Prefix Projection***):* The j^{th} *suf-projection* of $SeqDB$ w.r.t. a pattern P is defined as:

$$SeqDB^{suf-j}_{P} = \{(i, sx) \mid S_i \in SeqDB, \text{ satisfying } S_i = px{+\!\!+}sx, \text{ and } sx \text{ is } \textit{the } j^{\text{th}} \textit{ minimum suffix} \text{ of } S_i \text{ containing } P\}.$$

The *pre-projection* of $SeqDB$ w.r.t. a pattern P is defined as:

$$SeqDB^{pre}_{P} = \{(i, px) \mid S_i \in SeqDB, \text{ satisfying } S_i = px{+\!\!+}sx, \text{ and } sx \text{ is } \textit{the minimum suffix} \text{ of } S_i \text{ containing } P\}.$$

The (1st) minimum suffix of S_i containing P is sx iff no suffix shorter than sx contains P. The j^{th} minimum suffix of S_i containing P is sx iff no suffix, starting with $first(P)$, shorter than sx, and longer than the $(j-1)^{\text{th}}$ minimum suffix containing P, contains P.

Example 6: We still use $SeqDB$ in Table III to illustrate the above concepts. Consider pattern $\langle \mathtt{c}, \mathtt{d} \rangle$ here. The 1st suf-projection, the 2nd suf-projection, and the pre-projection of $SeqDB$ w.r.t. $\langle \mathtt{c}, \mathtt{d} \rangle$ are shown in Table V.

TABLE V
Suf-Projection and Pre-Projection

Suf-Projection $SeqDB^{suf-1}_{\langle \mathtt{c},\mathtt{d} \rangle}$
$(1, \langle \mathtt{c}, \mathtt{d} \rangle)$
$(2, \langle \mathtt{c}, \mathtt{b}, \mathtt{d} \rangle)$

Suf-Projection $SeqDB^{suf-2}_{\langle \mathtt{c},\mathtt{d} \rangle}$
$(1, \langle \mathtt{c}, \mathtt{b}, \mathtt{d}, \mathtt{c}, \mathtt{d} \rangle)$
$(2, \langle \mathtt{c}, \mathtt{b}, \mathtt{e}, \mathtt{a}, \mathtt{e}, \mathtt{d}, \mathtt{c}, \mathtt{b}, \mathtt{d} \rangle)$

Pre-Projection $SeqDB^{pre}_{\langle \mathtt{c},\mathtt{d} \rangle}$
$(1, \langle \mathtt{a}, \mathtt{b}, \mathtt{e}, \mathtt{c}, \mathtt{b}, \mathtt{d} \rangle)$
$(2, \langle \mathtt{a}, \mathtt{c}, \mathtt{b}, \mathtt{e}, \mathtt{a}, \mathtt{e}, \mathtt{d} \rangle)$

B. Anti-Monotonicity Property of Confidence

Consider a recurrent rule R in the form $R_{pre} \rightarrow R_{post}$ in $SeqDB$. In this subsection, we discuss how to utilize $SeqDB^{pre}_{R_{post}}$ to calculate $conf(R)$. We also present the anti-monotonicity property of confidence.

Suppose we have the confidence $conf(R) = \alpha$. This rule implies that in the database, the consequence R_{post} of the rule appears after a fraction α of the occurrences of the premise R_{pre} of the rule. Note both R_{pre} and R_{post} can appear one or more times in a sequence $S_i \in SeqDB$. Occurrences of premise R_{pre} which are not followed by an occurrence of R_{post} contribute negatively to the confidence of the rule. On the other hand, those followed by one or more occurrence of R_{post} contribute positively to the confidence.

Of importance is the portion of each sequence before the last occurrences of the consequent R_{post}–this portion of sequences is exactly formulated as the prefix projection $SeqDB^{pre}_{R_{post}}$. For each S_i in $SeqDB$, occurrences of R_{pre} appearing *before the last occurrence* of R_{post} contribute positively to the confidence, because there is at least one instance of R_{post} afterwards. There are totally $sup^{all}(R_{pre}, SeqDB^{pre}_{R_{post}})$ such occurrences of R_{pre}. On the other hand, those appearing *after the last occurrence* of R_{post} contribute negatively to the confidence.

Formally, we can prove that $sup(R_{post}, SeqDB^{all}_{R_{pre}})$ is equal to $sup^{all}(R_{pre}, SeqDB^{pre}_{R_{post}})$, and thus the confidence of a rule can be also defined using pre-projection via R_{post}, as stated in the following proposition. We omit its proof due to the space limit.

Proposition 1: Consider a rule R, in the form of $R_{pre} \to R_{post}$, and a sequence database $SeqDB$. We have

$$conf(R, SeqDB) = \frac{sup^{all}(R_{pre}, SeqDB^{pre}_{R_{post}})}{sup^{all}(R_{pre}, SeqDB)}. \quad (1)$$

Example 7: Using the same setting as that of Example 4, now we compute the confidence of recurrent rule R, $\langle \mathtt{a}, \mathtt{b} \rangle \to \langle \mathtt{c}, \mathtt{d} \rangle$, in $SeqDB$ (Table III) in the way suggested by the above proposition. $SeqDB^{pre}_{\langle \mathtt{c},\mathtt{d} \rangle}$ is listed in Table V. It is easy to see there are three instances of $\langle \mathtt{a}, \mathtt{b} \rangle$ in $SeqDB^{pre}_{\langle \mathtt{c},\mathtt{d} \rangle}$ (two in S_1 and one in S_2). Therefore, we have $sup^{all}(\langle \mathtt{a}, \mathtt{b} \rangle, SeqDB^{pre}_{\langle \mathtt{c},\mathtt{d} \rangle}) = 3$. According to Equation 1, we have $conf(R) = 3/4$, obtaining the same result as in Example 4.

Proposition 1 also suggests the anti-monotonicity property of $conf(R)$ w.r.t. R_{post}: as R_{post} grows (*i.e.*, $e{+\!\!+}R_{post}$), $SeqDB^{pre}_{R_{post}}$ shrinks (the border shifts to the left, *i.e.*, for each $(i, px) \in SeqDB^{pre}_{R_{post}}$, the prefix px becomes shorter), and thus $sup^{all}(R_{pre}, SeqDB^{pre}_{R_{post}})$ and $conf(R)$ decrease. We formulate this intuition as Proposition 2, and illustrate it in Example 8.

Proposition 2: Consider two rules R and R' in a sequence database $SeqDB$, with $R'_{pre} = R_{pre}$ and $R'_{post} = e{+\!\!+}R_{post}$ for some event $e \in I$. Then we have $conf(R) \geq conf(R')$.

Proof: Consider any two prefixes of the same sequence S_i in $SeqDB$ from the prefix projections w.r.t. R and R', respectively. Suppose they are: $(i, px) \in SeqDB^{pre}_{R_{post}}$ and $(i, px') \in SeqDB^{pre}_{R'_{post}}$. Since $R'_{post} = e{+\!\!+}R_{post}$, we must have px is a super-sequence of px'. Therefore, there are at least as many occurrences of R_{pre} in px as the ones of R'_{pre} in px'. This implies

$$sup^{all}(R_{pre}, SeqDB^{pre}_{R_{post}}) \geq sup^{all}(R'_{pre}, SeqDB^{pre}_{R'_{post}}),$$

which completes the proof. ■

Example 8: Consider two rules $R = \langle a, b \rangle \to \langle c, d \rangle$ and $R' = \langle a, b \rangle \to \langle e, c, d \rangle$ in the sequence database $SeqDB$ (Table III). Recall $SeqDB^{pre}_{\langle \mathtt{c},\mathtt{d} \rangle}$ shown in Table V. $SeqDB^{pre}_{\langle \mathtt{e},\mathtt{c},\mathtt{d} \rangle}$ contains two elements: $(1, \langle \mathtt{a}, \mathtt{b} \rangle)$ and $(2, \langle \mathtt{a}, \mathtt{c}, \mathtt{b}, \mathtt{e}, \mathtt{a} \rangle)$. As can be verified, each sequence in $SeqDB^{pre}_{\langle \mathtt{c},\mathtt{d} \rangle}$ is a super-sequence of the corresponding sequence in $SeqDB^{pre}_{\langle \mathtt{e},\mathtt{c},\mathtt{d} \rangle}$; and, we can calculate $conf(R') = 2/4 \leq 3/4 = conf(R)$ (since $sup^{all}(\langle a, b \rangle, SeqDB^{pre}_{\langle e,c,d \rangle}) = 2$).

*Theorem 1 (***Anti-Monotonicity Property of Confidence***):* Consider two rules R and R' in a sequence database $SeqDB$, with $R'_{pre} = R_{pre}$ and $R_{post} = evs{+\!\!+}R'_{post}$, where evs is an arbitrary series of events. Then we have $conf(R) \geq conf(R')$. In other words, if R is not confident enough ($conf(R) < min\_conf$), R' is not either.

Proposition 2 directly leads to Theorem 1. Theorem 1 can be utilized to speed up the pairing of pre-condition candidates and post-condition candidates by avoiding generating rules with insufficient confidence. The basic idea is that, when we try to pair a pre-condition candidate with a post-condition candidate to form a rule, if the confidence of the resulting rule is lower than the threshold, we no longer try to pair the same pre-condition with any backward extension of the post-condition. Details are deferred to Section VI.

C. Strategies for Eliminating Redundant Rules

Aside from early detection of rules with low confidence, we also desire to detect redundant rules (see Definition 4.6) early. A rule is redundant if there exists another rule, which is a super-sequence and has the same support and confidence. Our goal is to detect the redundancy early in the pre-condition and post-condition candidates, so as to prune the pre-conditions and post-conditions which definitely lead to the formation of redundant rules. We still need to test whether a rule generated from pre/post-condition candidates is redundant afterwards, but our pruning strategies can reduce the number of pre/post-condition candidates by a lot.

Two strategies are introduced in Theorem 2 and Theorem 3, for pruning redundant pre-conditions and post-conditions, respectively. Theorem 2 is borrowed from [18].[2]

*Theorem 2 (***Pruning Redundant Pre-Conds***):* In a sequence database $SeqDB$, consider a pre-condition candidate R_{pre}. If there is a pre-condition candidate $R'_{pre} \sqsupset R_{pre}$ s.t. (i) $R'_{pre} = P_1{+\!\!+}e{+\!\!+}P_2$ while $R_{pre} = P_1{+\!\!+}P_2$, for some event e and nonempty P_1, P_2, and (ii) $SeqDB_{R_{pre}} = SeqDB_{R'_{pre}}$, then *for any* post-condition candidate $post$ and *any* forward extension $R_{pre}{+\!\!+}P$, the rule $(R_{pre}{+\!\!+}P) \to post$ is redundant.

Theorem 2 implies that, there is no need to put R_{pre} or its forward extension $(R_{pre}{+\!\!+}P)$ into the pre-condition candidate set, if R_{pre} satisfies the stated properties (i)-(ii). The intuition is that: $(R_{pre}{+\!\!+}P) \to post$ is redundant because there is a longer rule $(R'_{pre}{+\!\!+}P) \to post$ with the same support and confidence.

As Theorem 2 has been presented and proved in [18], thus in the following part of this subsection, we focus on how to prune post-conditions (Theorem 3). In [18], another theorem to prune redundant post-condition is given; however, applying

[2]Theorem 1 and 3 are new.

that theorem requires performing one mining operation on the projected database for every single pre-condition candidate, which is not efficient enough for large datasets.

We first discuss when a post-condition candidate is redundant w.r.t. confidence (Lemma 1), and then discuss when it is redundant w.r.t. support (Lemma 2). Finally, we combine them as our pruning strategy for post-conditions (Theorem 3).

Lemma 1 and 2 state that, if for a post-condition candidate, inserting some event in the middle does not change the pre-projection and suf-projection of $SeqDB$ w.r.t. it, then any backward extension of this post-condition can only generate redundant rules.

Lemma 1: In a sequence database $SeqDB$, consider a post-condition candidate R_{post}. If there is a post-condition candidate $R'_{post} \sqsupset R_{post}$ s.t.
(i) $R'_{post} = P_1{+\!\!+}e{+\!\!+}P_2$ while $R_{post} = P_1{+\!\!+}P_2$, for some event e, subsequences P_1, and (nonempty) P_2, and
(ii) $SeqDB^{pre}_{R_{post}} = SeqDB^{pre}_{R'_{post}}$,
then *for any* pre-condition candidate pre and *any* backward extension $P{+\!\!+}R_{post}$ of R_{post}, the rule $R = pre \to (P{+\!\!+}R_{post})$ is not *confidence-closed* (*i.e.*, there exists another rule $R' \sqsupset R$ s.t. $conf(R) = conf(R')$).

Proof: To prove this lemma, it suffices to prove the existence of R'. We construct $R' = pre \to (P{+\!\!+}R'_{post})$. In fact, from (ii) $SeqDB^{pre}_{R_{post}} = SeqDB^{pre}_{R'_{post}}$ and Equation (1) in Proposition 1, we directly get $conf(R) = conf(R')$, which completes the proof. ■

Lemma 2: In a sequence database $SeqDB$, consider a post-condition candidate R_{post}. If there is a post-condition candidate $R'_{post} \sqsupset R_{post}$ s.t.
(i) $R'_{post} = P_1{+\!\!+}e{+\!\!+}P_2$ while $R_{post} = P_1{+\!\!+}P_2$, for some event e, and subsequences P_1 and (nonempty) P_2,
(iii) $\forall j$: $SeqDB^{suf-j}_{R_{post}} = SeqDB^{suf-j}_{R'_{post}}$, and
(iv) $\forall j$: $\left|\left(SeqDB^{suf-j}_{R_{post}}\right)^{all}_{R_{post}}\right| = \left|\left(SeqDB^{suf-j}_{R'_{post}}\right)^{all}_{R'_{post}}\right|$,
then *for any* pre-condition candidate pre and *any* backward extension $P{+\!\!+}R_{post}$ of R_{post}, the rule $R = pre \to (P{+\!\!+}R_{post})$ is not *support-closed* (*i.e.*, there exists another rule $R' \sqsupset R$ s.t. $sup(R) = sup(R')$ and $sup^{all}(R) = sup^{all}(R')$).

Proof: Similar to the proof of Lemma 2, we prove the existence of R'. And again, we construct $R' = pre \to (P{+\!\!+}R'_{post})$ and only need to prove $sup(R) = sup(R')$ and $sup^{all}(R) = sup^{all}(R')$ to complete the proof.

To prove $sup(R) = sup(R')$, we only need to prove, for each sequence $S_i \in SeqDB$, $R \sqsubseteq S_i$ (*i.e.*, $pre{+\!\!+}P{+\!\!+}R_{post} \sqsubseteq S_i$) if and only if $R' \sqsubseteq S_i$ (*i.e.*, $pre{+\!\!+}P{+\!\!+}R'_{post} \sqsubseteq S_i$): From (iii), for each $(i, sx) \in SeqDB^{suf-1}_{R_{post}} = SeqDB^{suf-1}_{R'_{post}}$, we know $R_{post}, R'_{post} \sqsubseteq sx$. Suppose $S_i = px{+\!\!+}sx$, if $pre{+\!\!+}P \sqsubseteq px$, then we have both $R \sqsubseteq S_i$ and $R' \sqsubseteq S_i$, otherwise, neither is true.

To prove $sup^{all}(R) = sup^{all}(R')$, we only need to show, for each sequence $S_i \in SeqDB$, R ($pre{+\!\!+}P{+\!\!+}R_{post}$) and R' ($pre{+\!\!+}$ $P{+\!\!+}R'_{post}$) have the same number of instances in S_i: Suppose the shortest prefix of S_i containing $pre{+\!\!+}P$ is px_0. Consider $(i, sx) \in SeqDB^{suf-j}_{R_{post}} = SeqDB^{suf-j}_{R'_{post}}$ and let $S_i = px{+\!\!+}sx$. Let j' be the *minimum* j that satisfies $px_0 \sqsubseteq px$. Let $(i, sx') \in SeqDB^{suf-j'}_{R_{post}} = SeqDB^{suf-j'}_{R'_{post}}$ and $S_i = px'{+\!\!+}sx'$. From (iv), R_{post} has the same number of instances in sx' as R'_{post} has in sx'. So, R has the same number of instances in S_i as R' does. ■

Theorem 3 (**Pruning Redundant Post-Conds):** In a sequence database $SeqDB$, consider a post-condition candidate R_{post}. If the properties (i)-(iv) in Lemma 1 and 2 are satisfied, then *for any* pre-condition candidate pre and *any* backward extension $P{+\!\!+}R_{post}$, the rule $pre \to (P{+\!\!+}R_{post})$ is redundant.

Theorem 3 directly follows from Lemma 1 and 2, and it implies that, there is no need to put R_{post} or its backward extension $P{+\!\!+}R_{post}$ into the post-candidate set, if (i)-(iv) in Lemma 1 and 2 are satisfied.

Example 9: Consider a post-condition candidate $R_{post} = \langle c \rangle$ in the sequence database $SeqDB$ (Table III). Recall the suf-projection and pre-projection of $SeqDB$ w.r.t. $\langle c, d \rangle$ is shown in Table V. Choosing $R'_{post} = \langle c, d \rangle$, we can easily verify that R_{post} here satisfies properties (i)-(iv) in Lemma 1 and 2. So from Theorem 3, any backward extension of $\langle c \rangle$ can only generate redundant rules. Take the backward extension $\langle b, c \rangle$ for example: consider the rule $R = \langle a \rangle \to \langle b, c \rangle$ and $R' = \langle a \rangle \to \langle b, c, d \rangle$; R is redundant because we have $conf(R) = conf(R') = 2/3$, $sup(R) = sup(R') = 2$, and $sup^{all}(R) = sup^{all}(R') = 3$.

Theorem 2 and 3 throw away those pre- and post- condition candidates that are generating redundant rules, before we try to pair them. The two theorems can serve as early-stop conditions in the pattern-growth generation of pre- and post- condition candidates.

In the next section, we will show how to utilize the pruning strategies, namely Theorem 1, 2, and 3, in a holistic mining algorithm.

VI. Algorithm

In this section, we first introduce our framework to mine (non-redundant) rules in Section VI-A. Two stages of our mining framework are detailed in Section VI-B and VI-C. We analyze our algorithm in Section VI-D. We call our framework <u>B</u>idirecti<u>O</u>nal pruning-<u>B</u>ased recurrent rule mining algorithm (BOB).

A. Algorithm Framework

Recall the goal of our mining algorithm is, given a sequence database $SeqDB$ and three thresholds min_sup, min_sup^{all}, and min_conf, we want to find all the rules Rs in $SeqDB$ with support $sup(R)$ no less than min_sup, instance support $sup^{all}(R)$ no less than min_sup^{all}, and confidence $conf(R)$ no less than min_conf.

Our mining algorithm consists of two major stages. In the first stage, we use a pattern-growth algorithm to find a candidate set of pre-conditions and a candidate set of post-conditions–strategies introduced by Theorem 2 and 3 are used to prune candidates for mining non-redundant rules. In the second stage, pre-conditions and post-conditions are paired to form rules, with the *significant* (*sufficient supports and confidence*) and non-redundant ones output–Theorem 1 is used to speed up the pairing procedure. We outline the algorithm as follows:

1) (Mining Pruned Pre-Conditions) Mine a set of pre-condition candidates PRE satisfying min_sup. Candidates are pruned based on Theorem 2.
2) (Mining Pruned Post-Conditions) Mine a set of post-condition candidates $POST$ satisfying min_sup. Candidates are pruned based on Theorem 3.
3) (Forming Rules) For each $pre \in PRE$, pair it with each $post \in POST$ to form a rule R (in some order). Test whether a rule is significant (sufficient supports and confidence), and keep all significant rules in a set $RULES$. Theorem 1 is used to speed up the pairing here.
4) (Removing Redundant Rules) Remove the remaining redundant rules (Definition 4.6) in $RULES$, and output the rest.

Step 1 and 2 above belong to the first stage, which will be elaborated in Section VI-B. Step 3 and 4 belong to the second stage, which will be described in more detail in Section VI-C.

B. Stage I: Mining Pruned Candidates

We propose a new pattern-growth algorithm to find pre-condition (PRE) and post-condition ($POST$) candidates.

The algorithm for mining pruned candidates is outlined in Algorithm 1. It is important to note that both the mining of pre-condition candidates and the mining of post-condition candidates are done using *pattern-growth (i.e., depth-first search of the pattern space)*. The difference is: to mine pre-conditions, patterns are grown forward (P is grown to $P{+\!+}e$), and thus we utilize the projections of $SeqDB$ (projected to the suffixes); on the other hand, to mine post-condition candidates, patterns are grown backward ($e{+\!+}P$), and thus we utilized the pre-projections of $SeqDB$ (projected to the prefixes). The subroutine BackwardGrow and ForwardGrow implement the forward and backward pattern growth strategy respectively. The reason for differentiating mining pre-conditions from mining post-conditions is to facilitate the pruning strategies presented in Theorem 2 and 3.

Line 6 and 11 utilize the anti-monotonicity property of support (longer patterns cannot have higher sequence supports).

Line 7 and 12 apply the pruning strategies in Theorem 2 and 3, respectively. It should be noted that, once a pattern P is pruned, we stop growing from P, because Theorem 2(3) states that any of its forward(backward) extension can only lead to redundant rules.

Representation of Projected Databases. The remaining question is how to represent projected databases $SeqDB_P$, $SeqDB_P^{all}$, $SeqDB_P^{pre}$, and $SeqDB_P^{suf-j}$ (used in Theorem 2 and 3), which are referred to in Algorithm 1. It is not space-efficient to store them explicitly. Interestingly, all the projected databases are either prefixes or suffixes of the original sequences. Therefore, we only need to store the ending/starting positions of prefixes/suffixes in the original sequences, instead of explicitly storing the prefixes/suffixes.

Algorithm 1 Mining (Pruned) Candidates

Input: sequence database $SeqDB = \{S_1, S_2, \ldots, S_N\}$; threshold min_sup
Output: pre-/post-condition candidates PRE/$POST$

```
1:  $I \leftarrow$ all events appearing in $SeqDB$; $PRE, POST \leftarrow \varnothing$;
2:  for each $e \in I$ do
3:      ForwardGrow($e, SeqDB_P$);
4:      BackwardGrow($e, SeqDB_P^{pre}$);
5:  return $PRE$ and $POST$;
```

Subroutine ForwardGrow($P, SeqDB_P$)
Input: pattern P and projected database $SeqDB_P$ w.r.t. P
Objective: add pre-condition candidates with prefix P into PRE

```
6:  if $|SeqDB_P| \geq min\_sup$ then
7:      if $P$ satisfies (i)-(ii) in Theorem 2 then prune $P$ else
8:          $Pre \leftarrow Pre \cup \{P\}$;
9:          for each event $e$ appearing in $SeqDB_P$ do
10:             ForwardGrow($P{+\!+}e, SeqDB_{P{+\!+}e}$);
```

Subroutine BackwardGrow($P, SeqDB_P^{pre}$)
Input: pattern P and pre-projected database $SeqDB_P^{pre}$ w.r.t. P
Objective: add post-condition candidates with suffix P into $POST$

```
11: if $|SeqDB_P| \geq min\_sup$ then
12:     if $P$ satisfies (i)-(iv) in Theorem 3 then prune $P$ else
13:         $Post \leftarrow Post \cup \{P\}$;
14:         for each event $e$ appearing in $SeqDB_P^{pre}$ do
15:             BackGrow($e{+\!+}P, SeqDB_{e{+\!+}P}^{pre}$);
```

C. Stage II: Forming Rules

This subsection describes Step 3-4 of our algorithm framework, *i.e.*, how to form significant non-redundant rules from the two candidate sets PRE and $POST$ obtained in Step 1-2.

The basic idea is to pair each pre-condition from PRE with each post-condition from $POST$ to form a rule, and then test whether its supports and confidence are high enough and whether it is redundant. Note the strategies introduced in Theorem 2 and 3 (line 7 and 12 in Algorithm 1) can prune some redundant rules but not all of them. So the redundancy test here is still needed. The main purpose of Theorem 2 and 3 is to shrink the candidate sets, and thus speed up the mining and pairing of candidates.

This stage is outlined in Algorithm 2. There are four remaining questions: (i) how to organize PRE and $POST$ to facilitate the pairing procedure; (ii) in what order to pair each $pre \in PRE$ with each $post \in POST$; (iii) how to compute the supports and confidence of a rule; (iv) how to test whether a rule is redundant.

Prefix Hash Tree. We use a prefix hash-tree (PHT) data

Algorithm 2 Forming Rules from Candidates

Input: pre-/post-condition candidates $PRE/POST$ in $SeqDB$; thresholds min_sup, min_sup^{all}, and min_conf
Output: Significant and non-redundant rules

```
1: Store $PRE$ in a Prefix Hash Tree $PHT_{PRE}$;
2: Store $POST$ in a Prefix Hash Tree $PHT_{POST}$;
3: for each $pre \in PRE$ do
4:    for each $post \in PHT_{POST}$ (in DFS order) do
5:       Let $R$ be the rule $pre \rightarrow post$;
6:       if $conf(R) < min\_conf$ or
               $sup(R) < min\_sup$ then
            skip the subtree below $post$;
7:       Add $R$ into $RULES$;
8: Eliminate redundant rules in $RULES$;
9: return $RULES$
```

structure to organize the set of candidates PRE and $POST$. For $POST$, all patterns are stored in the PHT in reverse order. Each node in this tree represents a pattern (obtained by following the path from root to this node). Each node is also associated with its corresponding projected databases: $SeqDB_P$, $SeqDB_P^{all}$, and $SeqDB_P^{pre}$. As is discussed in Section VI-B, each projected database is stored implicitly. Each node has a hash table to quickly locate one of its child in a constant lookup operation given an event.

An example of a PHT is shown in Figure 1.

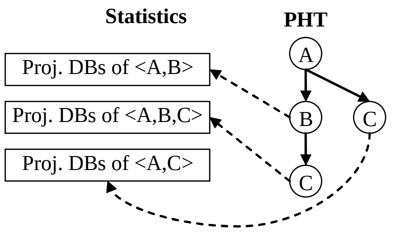

Fig. 1. **Prefix Hash-Tree (PHT) Data Structure**

Embedding the Anti-Monotonicity Property. We store $POST$ in a PHT in reverse order and scan $R_{post} \in POST$ in DFS order. By following the DFS order to visit nodes in the PHT, a post-condition is always scanned earlier than its backward extensions. This feature enables us to embed the anti-monotonicity properties of confidence (Theorem 1) and sequence support into the pairing algorithm (*i.e.*, line 6 in Algorithm 2): for any $R_{pre} \in PRE$, once a rule $R_{pre} \rightarrow R_{post}$ has a low support or low confidence, we can skip scanning the whole subtree below R_{post}. For example, in Figure 1 (suppose it is the PHT of post-condition candidates), for some pre-condition candidate R_{pre}, if we find $R_{pre} \rightarrow \langle \mathtt{a}, \mathtt{b} \rangle$ has a low support or confidence, we can skip scanning $\langle \mathtt{a}, \mathtt{b}, \mathtt{c} \rangle$, and continue to $\langle \mathtt{a}, \mathtt{c} \rangle$.

Computing Supports and Confidence. We store both PRE and $POST$ in PHTs. For a rule $R = R_{pre} \rightarrow R_{post}$, its supports and confidence can be computed from the projected databases w.r.t. R_{pre} and R_{post} stored in the PHTs. To compute $sup(R)$ we compare $SeqDB_{R_{pre}}$ with $SeqDB_{R_{post}}^{pre}$ and look for common sequences where the premise R_{pre} occurs first before R_{post}. For computing $conf(R)$, we use the formula defined in Proposition 1 and use the projected database $SeqDB_{R_{pre}}^{all}$ and $SeqDB_{R_{post}}^{pre}$. The ratio of the occurrences of R_{pre} in $SeqDB_{R_{post}}^{pre}$ to all occurrences of R_{pre} is $conf(R)$. $Sup^{all}(R)$ is the size of $SeqDB_{pre+\!+post}^{all}$ which is usually in the PHTs. However, there are cases where $R_{pre} \rightarrow R_{post}$ is a significant rule while $R_{pre}+\!+R_{post}$ is neither in PHT_{PRE} nor PHT_{POST}. For these cases, we need to re-scan the database to find the instance support of R.

Eliminating Redundant Rules. Next, in line 8 of Algorithm 2 we want to remove redundant rules. Some redundant rules have been pruned early from the candidate sets in stage I. But to eliminate all redundant rules, this step is needed. To illustrate the need for removing redundant rules even after applying Theorem 2 and 3, consider the following database containing two sequences:

Seq ID.	**Sequence**
$S1$	$\langle \mathtt{a}, \mathtt{b}, \mathtt{b}, \mathtt{c} \rangle$
$S2$	$\langle \mathtt{a}, \mathtt{b}, \mathtt{b}, \mathtt{c} \rangle$

The rule $R1 = \langle b \rangle \rightarrow \langle c \rangle$ has a sequence support, instance support, and confidence of 2, 2, and 100%. These are the same as those of rule $R2 = \langle a \rangle \rightarrow \langle b, c \rangle$. Although $R2$ subsumes $R1$, yet, the pre-condition $\langle b \rangle$ is not considered redundant due to $\langle a \rangle$ by Theorem 2. Similarly $\langle c \rangle$ is not considered redundant due to $\langle b, c \rangle$ by Theorem 3.

We map rules R's with the same supports and confidence, *i.e.*, the same triple ($sup(R)$, $sup^{all}(R)$, $conf(R)$), into a bucket (using, *e.g.*, a hash table). Then, for each bucket, we want to remove the redundant rules (*i.e.*, the ones with their super-pattern in the same bucket).

D. Algorithm Comparative Analysis

Consider a database containing a set of sequences with events coming from an alphabet A. The worst case complexity of mining frequent patterns of length at most k from such a database is O($|A|^k$) database scan operations. Consider a set of patterns P, the worst case complexity of constructing PHT from patterns in P is O($|P|$) database scan operations. In our analysis, we use database scan as the unit of operation. We ignore the time needed to eliminate redundant rules since no database scan operation is involved.

In BOB, we perform: 1. two mining operations on the original sequence database to obtain the set of PRE and $POST$, 2. construction of PHT_{PRE} and PHT_{POST}, and 3. Composition of premises and consequents to form non-redundant rules. Let us also assume that both the premises and consequents have a maximum length of k. The complexity of our approach is at most O($|A|^k$ + $|A|^k$ + $|A|^k$ + $|A|^k$ + $|RULES\_REQ\_SCAN|$) = O($|A|^k$ + $|RULES\_REQ\_SCAN|$). The first term of the first formula (i.e., $|A|^k$) is due to the mining of the premises, the second is due to the mining of the consequents, the third and fourth are due to the construction of PHT_{PRE} and PHT_{POST}, and the last is due to the composition of premises and consequents to form rules. Remember many rules can be constructed without requiring any additional database scan operation. Some however require the re-scanning of the database to compute their instance support values (see Section VI-C).

The algorithm in [18] (LKL08) takes as input a sequence database $SeqDB$ and works in two steps: 1. Mine a pruned set of pre-conditions obeying minimum sequence support threshold and Theorem 2 from $SeqDB$, 2. For each pre-condition, mine a set of post-conditions obeying the minimum confidence and minimum instance support thresholds. At the end of the first step, for each pre-condition pre, LKL08 constructs a projected database $SeqDB_{pre}^{all}$. Another mining operation is then performed on this projected database. The complexity is thus O($|A|^k \times |A|^k$). Notice that if $|RULES\_REQ\_SCAN|$ is not large, the complexity of our approach is smaller by an exponential factor than that of LKL08. In the worst case, $|RULES\_REQ\_SCAN|$ = $|A|^k \times |A|^k$ though.

However, in frequent pattern mining, worst case analysis is often not interesting. This is so as in the *worst case*, all pruning properties do not work. In the worst case, all possible rules up to a particular length are significant and none of them is redundant. The following points summarize reasons behind the superiority of our approach as compared to [18]:

1) We employ two new pruning strategies described by Theorem 1 and 3. These strategies are embedded into our new mining algorithm to remove search space not pruned before by the approach in [18].
2) The projected database created in [18] could be very large especially if patterns *recur* in a sequence many times. For a premise pre, the projected database $SeqDB_{pre}^{all}$ could be larger than the original $SeqDB$ especially in cases where the number of repetitions of pre within each sequence containing it is high. Also, projecting with respect to a premise can produce a very localized dataset (i.e., a single sequence is split into its many suffixes), resulting in hard-to-mine dense dataset with a large number of frequent patterns even at a high support threshold.
3) During the mining of premises and consequents from $SeqDB$, for each pattern P, we store summary information, e.g., $SeqDB_P^{pre}$ and $SeqDB_P^{all}$. This information is used to immediately prune insignificant rules not satisfying minimum sequence support and confidence thresholds (see Section VI-C). For this pruning, we do not need to re-scan the database, rather only the summary information needs to be analyzed. The algorithm in [18] can only perform database scan operations to prune candidate rules.

To compare effectiveness of various pruning strategies, experiments on various datasets are needed. We perform this empirical evaluation in Section VII.

VII. Empirical Evaluation

Experiments have been performed to evaluate the scalability of our approach. A case study on analyzing traces from an instant messaging application has also been conducted.

Environment and Pattern Miners. All experiments are performed on a Pentium Core 2 Duo 3.17GHz PC with 3GB main memory running Windows XP Professional. Algorithms are written in Visual C#.Net We compare the approach presented in [18] with our approach. We refer to the two approaches as LKL08 and BOB respectively.

Datasets. To reduce the threat of external validity (i.e., the generalizability of our result), we investigate a variety of datasets. Four datasets, two synthetic and two real, are studied.

IBM synthetic data generator is used [3]. It is modified to produce sequences of events, rather than sequences of sets of events. The generator accepts a set of parameters. We focus on four parameters: D, C, N, and R. They correspond to the number of sequences (in 1000s), the average number of events per sequence, the number of different events (in 1000s), and the repetition level (range: 0 to 1) respectively. All other parameters of the synthetic data generator are set to their default values. We experiment with two synthetic datasets: D5C20N10R0.5 and D10C10N10R0.5. Dataset D5C20N10R0.5 contains sequences with an average length of 64.4 and a maximum length of 275. Dataset D10C10N10R0.5 contains sequences with an average length of 31.2 and a maximum length of 133. D5C20N10R0.5 has less sequences of longer lengths. On the other hand, D10C10N10R0.5 has more sequences of shorter lengths.

We also experiment on a click stream dataset (*i.e.*, Gazelle dataset) from KDDCup 2000 which has also been used to evaluate frequent sequential pattern miners, i.e., CloSpan [26] and BIDE [25]. The dataset contains 29,369 sequences with an average length of 3 and a maximum length of 651. Compared to the two synthetic datasets, this real data has a lower average length but contains sequences of longer lengths. The gap in the lengths of long and short sequences is also wider.

To evaluate our algorithm performance on mining from program traces, we generate traces from TotInfo program in the Siemens Test Suite [13]. The test suite comes with 893 correct test cases. We run these test cases to obtain 893 traces. Each trace is a sequence of events where every event is a method invocation. We refer this dataset as the TotInfo dataset. The TotInfo dataset contains sequences with an average length of 12.1 and a maximum length of 136.

Results. In Figure 2 and 3, we plot the runtime needed and the number of rules mined from D5C20N10R0.5 when varying the minimum sequence support threshold and the minimum confidence threshold respectively. In this study, for simplicity sake, we set the minimum instance support threshold to be equal to the minimum sequence support threshold. Each of the runtime graphs has two lines corresponding to the two algorithms' runtime at various thresholds.

We could note that BOB is faster than LKL08 by up to one order of magnitude (i.e., 10x faster). Also, we note that BOB pruning strategy is effective in reducing required runtime when the minimum confidence threshold is raised from 50% to 90%. On the other hand, no significant performance change could be noticed in LKL08 case.

In Figure 4 and 5, we plot the runtime needed and the

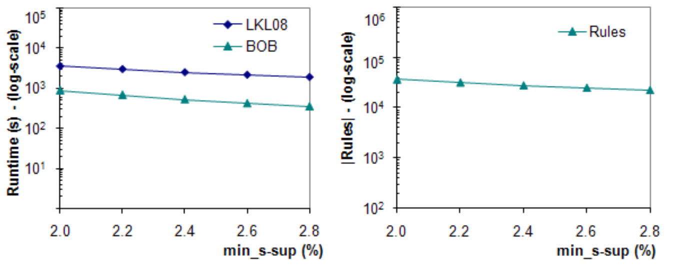
Fig. 2. Runtime (left) & |Rules| (right) for D5C20N10R0.5 dataset when varying min_s-sup (at min_conf=50%)

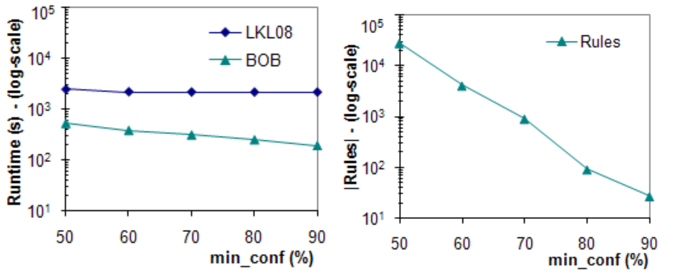
Fig. 3. Runtime (left) & |Rules| (right) for D5C20N10R0.5 dataset when varying min_conf (at min_s-sup =2.4%)

number of rules mined from the D10C10N10R0.5 dataset when varying the minimum sequence support threshold and the minimum confidence threshold respectively. Again, we notice that BOB is up to 10 times faster than LKL08. Also, we note that the gap between the performance of BOB and LKL08 is increased when the minimum confidence threshold is raised from 50% to 90%. We do not experiment with mining at lower confidence threshold levels as low confidence rules have very little use and are likely to only capture noises.

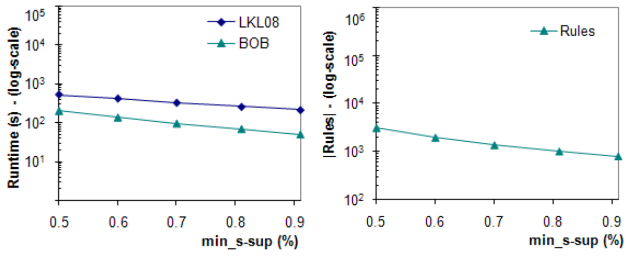
Fig. 4. Runtime (left) & |Rules| (right) for D10C10N10R0.5 dataset when varying min_s-sup (at min_conf=50%)

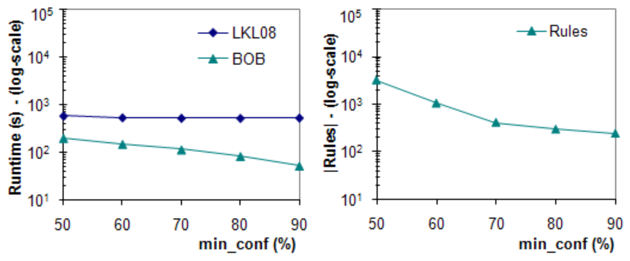
Fig. 5. Runtime (left) & |Rules| (right) for D10C10N10R0.5 dataset when varying min_conf (at min_s-sup =0.5%)

In Figure 6 and 7, we plot the runtime needed and the number of rules mined from the Gazelle dataset when varying the minimum sequence support threshold and the minimum confidence threshold respectively. We notice that BOB improves the performance of LKL08 by two orders of magnitude (i.e., more than 100x faster). At support level 0.020%, LKL08 is not able to finish within 8 hours. Thus BOB can successfully mine rules at a lower support threshold that is not minable by LKL08 in a reasonable amount of time.

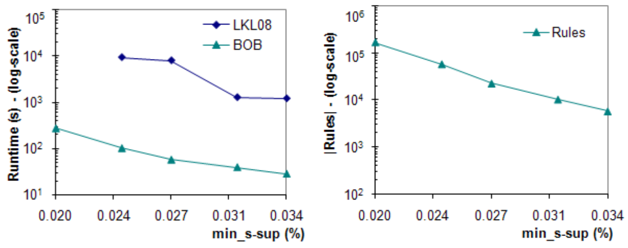
Fig. 6. Runtime (left) & |Rules| (right) for Gazelle dataset when varying min_s-sup (at min_conf=50%)

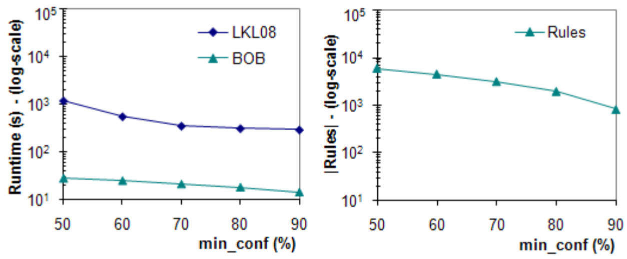
Fig. 7. Runtime (left) & |Rules| (right) for for Gazelle dataset when varying min_conf(at min_sup=0.034%)

In Figure 8 and 9, we plot the runtime needed and the number of rules mined from the TotInfo dataset when varying the minimum sequence support threshold and the minimum confidence threshold respectively. At all support levels shown in Figure 8 and 50% minimum confidence threshold, LKL08 is not able to run due to an out of memory exception while BOB is able to finish in less than 2 minutes. When we decrease the minimum confidence threshold, we see an exponential increase in the runtime of LKL08. BOB runtime on the other hand remains the same. LKL08 performs a pattern mining operation for every projected database of each mined pre-conditions, this causes the high runtime values.

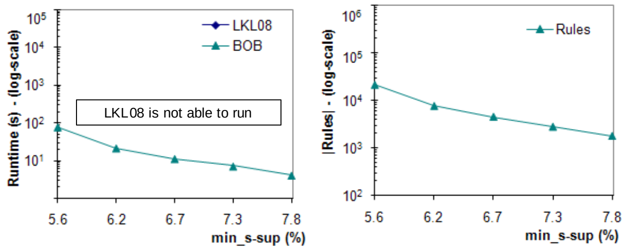

Fig. 8. Runtime (left) & |Rules| (right) for TotInfo dataset when varying min_s-sup (at min_conf=50%)

Case Study. In past studies, recurrent rules, either in full or restricted form, have been mined from various software datasets [18], [19]. Mined rules correspond to interesting temporal properties extracted from execution traces of programs. In [18], Lo et al. mined rules from execution traces generated by running test cases of JBoss Application Server. In [19], in a study within Microsoft, Lo et al. mined a restricted form of two-event recurrent rules with quantification from Windows device drivers and other Windows applications.

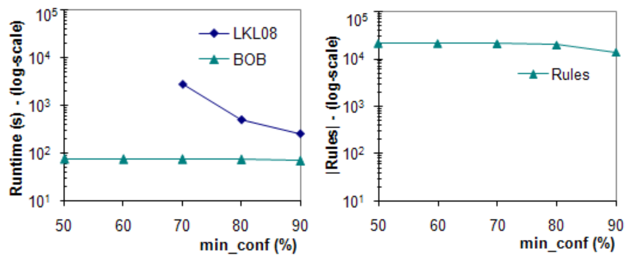

Fig. 9. Runtime (left) & |Rules| (right) for TotInfo dataset when varying min_conf (at min_s-sup =5.6%)

In this study, we consider another trace data obtained from user interactions with a drawing utility of an instant messaging application. Many systems like the one that we analyze do not come with sufficient test cases. We ask a student who is not involved in this study to interact with the system.

Premise →	**Consequent**
Map PC.createWritersMap() void PC.showWindow() void PC.unselect() void PC.showWindow() JID PC.getMyJID() void PC.draw(Shape)	PH.addShapeDrawnByMe(…)
Legend	
PC- nu.fw.jeti.plugins.drawing.shapes.PictureChat PH - nu.fw.jeti.plugins.drawing.shapes.PictureHistory	

Fig. 10. **Jeti Instant Messaging Application: Drawing Scenario**

We use Jeti [1], a popular instant messaging application which supports many features. We record 30 interactions with the drawing tool of Jeti application and collect 30 traces. The traces have an average length of 1,430 and a maximum length of 11,838 events. Each event is a method call. The purpose of this case study is to show the usefulness of the mined rules by discovering frequent and significant rules describing behaviors of the drawing sub-component of Jeti. Using minimum sequence support and instance support of 25 traces/instances and a minimum confidence threshold of 90%, BOB could complete in 57 seconds while LKL08 is only able to complete in 2844 seconds. A total of 19 rules are collected after applying the following post-processing steps:

1. Density. Only report a mined rule iff the number of its unique events is more than 80% of its length.
2. Ranking. Order mined rules according to their lengths and support values.

A sample mined rule is shown in Figure 10. The rule captures the scenario when a user draws an object (e.g., a rectangle, a line, etc) to a canvas. First, a resource, i.e., a Map object, is created by a `PictureChat` object. Next, multiple invocations of `showWindow(...)` method are made by different callers. When the application starts, an empty window is first shown or displayed. After an object is drawn, the drawn object (i.e, rectangle, line, etc.) would request the canvas (i.e., `PictureChat` object) to "unselect" and redraw itself. The system next retrieves the identifier of the user that draws the object by the invocation of `getMyJID(...)` method. This identifier is later affixed to the object drawn. Finally, the canvas records the operation in a `PictureHistory` object.

VIII. Conclusion

This work proposes a new approach to mine recurrent rules in the form of "Whenever a series of events occurs, another series of events also occurs". The proposed approach is more scalable than the previous approach in [18]. Rather than performing a mining operation for each non-redundant pre-condition as proposed in [18], our new approach employs a number of new pruning strategies embedded in a new mining algorithm that requires only two mining and some additional database scanning operations. Under a condition which holds in many cases, the complexity of the proposed algorithm is smaller by an exponential factor than the complexity of the one proposed in [18]. We have experimented on various datasets: synthetic and real. Experiments have shown that the new algorithm improves the runtime of the previous algorithm by up to two orders of magnitude. In the future, we are looking into more applications of the mining algorithm and opportunities to further speed up the mining process.

References

[1] "Jeti. Version 0.7.6 (Oct. 2006)." http://jeti.sourceforge.net/.
[2] R. Agrawal and R. Srikant, "Fast algorithms for mining association rules," in *Int. Conf. on Very Large Data Bases*, 1994.
[3] ——, "Mining sequential patterns." in *Int. Conf. on Data Eng.*, 1995.
[4] M. Arenas, P. Barcelo, and L. Libkin, "Combining temporal logics for querying xml documents," in *Int. Conf. on Database Theory*, 2007.
[5] R. Capilla and J. Duenas, "Light-weight product-lines for evolution and maintenance of web sites," in *Euro. Conf. on Soft. Maintenance and Re-Eng.*, 2003.
[6] E. Clarke, O. Grumberg, and D. Peled, *Model Checking*. MIT Press, 1999.
[7] S. Deelstra, M. Sinnema, and J. Bosch, "Experiences in software product families: Problems and issues during product derivation," in *Soft. Product Line Conf.*, 2004.
[8] B. Ding, D. Lo, J. Han, and S.-C. Khoo, "Efficient mining of closed repetitive gapped subsequences from a sequence database." in *Int. Conf. on Data Eng.*, 2009.
[9] M. Dwyer, G. Avrunin, and J. Corbett, "Patterns in property specifications for finite-state verification." in *Int. Conf. on Soft. Eng.*, 1999.
[10] G. Garriga, "Discovering unbounded episodes in sequential data." in *Euro. Conf. on Prin. and Prac. of Knowledge Disco. in Databases*, 2003.
[11] M. Gupta and S. Ray, "Sequence pattern discovery with applications to understanding gene regulation and vaccine design." in *Handbook of Statistics*, 2006.
[12] S. Harms, J. Deogun, and T. Tadesse, "Discovering sequential association rules with constraints and time lags in multiple sequences." in *Int. Symp. on Intel. Sys.*, 2002.
[13] M. Hutchins, H. Foster, T. Goradia, and T. Ostrand, "Experiments on the effectiveness of dataflow- and control-flow-based test adequacy criteria," in *Int. Conf. on Soft. Eng.*, 1994.
[14] M. Huth and M. Ryan, *Logic in Computer Science*. Cambridge, 2004.
[15] W. Jamroga, "A temporal logic for stochastic multi-agent systems," in *Pacific Rim Int. Conf. on Multi-Agents*, 2008.
[16] H. Kugler, D. Harel, A. Pnueli, Y. Lu, and Y. Bontemps, "Temporal logic for scenario-based specifications," in *Int. Conf. on Tools and Algo. for the Constr. and Ana. of Sys.*, 2005.
[17] D. Lo, S.-C. Khoo, and C. Liu, "Efficient mining of iterative patterns for software specification discovery." in *Int. Conf. on Knowledge Disco. and Data Mining*, 2007.
[18] ——, "Efficient mining of recurrent rules from a sequence database." in *Int. Conf. on Database Sys. for Adv. App.*, 2008.
[19] D. Lo, G. Ramalingam, V.-P. Ranganath, and K. Vaswani, "Mining quantified temporal rules: Formalism, algorithms, and evaluation." in *Working Conf. on Reverse Eng.*, 2009.
[20] H. Mannila, H. Toivonen, and A. Verkamo, "Discovery of frequent episodes in event sequences," *Data Mining and Knowledge Disco.*, vol. 1, pp. 259–289, 1997.
[21] P. Monteiro, D. Ropers, R. Mateescu, A. Freitas, and H. de Jong, "Temporal logic patterns for querying dynamic models of cellular interaction networks," *Bioinformatics*, vol. 24, pp. 227–233, 2008.
[22] N. Pasquier, Y. Bastide, R. Taouil, and L. Lakhal, "Discovering frequent closed itemsets for association rules," in *Int. Conf. on Database Theory*, 1999.
[23] J. Pei, J. Han, B. Mortazavi-Asl, H. Pinto, Q. Chen, U. Dayal, and M.-C. Hsu, "Prefixspan: Mining sequential patterns efficiently by prefix-projected pattern growth." in *Int. Conf. on Data Eng.*, 2001.
[24] M. Spiliopoulou, "Managing interesting rules in sequence mining." in *Euro. Conf. on Prin. and Prac. of Knowledge Disco. in Databases*, 1999.
[25] J. Wang and J. Han, "BIDE: Efficient mining of frequent closed sequences." in *Int. Conf. on Data Eng.*, 2004.
[26] X. Yan, J. Han, and R. Afhar, "CloSpan: Mining closed sequential patterns in large datasets." in *SIAM Int. Conf. on Data Mining*, 2003.
[27] M. Zaki, "Mining non-redundant association rules." *Data Mining and Knowledge Disco.*, 2004.
[28] S. Zhang, "A temporal logic for supporting historical databases." *Knowledge and Info. Sys.*, 2000.