Efficient Mining of Closed Repetitive Gapped Subsequences from a Sequence Database

Bolin Ding [#1], David Lo [&*‡2], Jiawei Han [#3], and Siau-Cheng Khoo [*4]

[#]*Department of Computer Science, University of Illinois at Urbana-Champaign*
[1 3]`{bding3,hanj}@uiuc.edu`
[&]*School of Information Systems, Singapore Management University*
[2]`davidlo@smu.edu.sg`
[*]*Department of Computer Science, National University of Singapore*
[4]`khoosc@comp.nus.edu.sg`

Abstract— There is a huge wealth of sequence data available, for example, customer purchase histories, program execution traces, DNA, and protein sequences. Analyzing this wealth of data to mine important knowledge is certainly a worthwhile goal.

In this paper, as a step forward to analyzing patterns in sequences, we introduce the problem of mining closed repetitive gapped subsequences and propose efficient solutions. Given a database of sequences where each sequence is an ordered list of events, the pattern we would like to mine is called *repetitive gapped subsequence*, which is a subsequence (possibly with gaps between two successive events within it) of some sequences in the database. We introduce the concept of *repetitive support* to measure how frequently a pattern repeats in the database. Different from the sequential pattern mining problem, *repetitive support* captures not only repetitions of a pattern in different sequences but also the repetitions within a sequence. Given a user-specified support threshold min_sup, we study finding the set of all patterns with repetitive support no less than min_sup. To obtain a compact yet complete result set and improve the efficiency, we also study finding closed patterns. Efficient mining algorithms to find the complete set of desired patterns are proposed based on the idea of *instance growth*. Our performance study on various datasets shows the efficiency of our approach. A case study is also performed to show the utility of our approach.

I. INTRODUCTION

A huge wealth of sequence data is available from wide-range of applications, where sequences of events or transactions correspond to important sources of information including customer purchasing lists, credit card usage histories, program execution traces, sequences of words in a text, DNA, and protein sequences. The task of discovering frequent subsequences as patterns in a sequence database has become an important topic in data mining. A rich literature contributes to it, such as [1], [2], [3], [4], [5], [6], and [7]. As a step forward in this research direction, we propose the problem of mining closed repetitive gapped subsequences from a sequence database.

By *gapped subsequence*, we mean a subsequence, which appears in a sequence in the database, possibly with gaps between two successive events. For brevity, in this paper, we use the term *pattern* or *subsequence* for gapped subsequence.

In this paper, we study finding *frequent repetitive patterns*, by capturing not only pattern *instances* (*occurrences*) *repeating in different sequences* but also those *repeating within each*
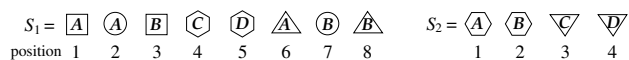
‡Work done while the second author was with National Univ. of Singapore

Fig. 1. Pattern AB and CD in a Database of Two Sequences

sequence. As in other frequent pattern mining problems, we measure how frequently a pattern repeats by "*support*". Following is a motivating example about our support definition.

Example 1.1: Figure 1 shows two sequences, which might be generated when a trading company are handling the requests of customers. Let the symbols represent: 'A' - *request placed*, 'B' - *request in-process*, 'C' - *request cancelled*, and 'D' - *product delivered*. Suppose $S_1 = AABCDABB$ and $S_2 = ABCD$ are two sequences from different customers.

Consider pattern AB ("*process a request*" after "*the customer places one*"). We mark its instances in S_1 and S_2 with different shapes. We have totally 4 instances of AB, and among them, squares, circles, and triangles, are the ones repeating within S_1. Consider CD ("*deliver the product*" after "*the customer cancels a request*"). It has 2 instances, each repeating only once in each sequence. In our definition, the support of AB, $\mathsf{sup}(AB) = 4$, and $\mathsf{sup}(CD) = 2$. It can be seen that although both AB and CD appear in two sequences, AB repeats more "frequently" than CD does in S_1. This information is useful to differentiate the two customers. ∎

Before defining the support of a pattern formally, there are two issues to be clarified:

1) We only capture the *non-overlapping* instances of a pattern. For example in Figure 1, once the pair of squares is counted in the support of pattern AB, 'A' in square with 'B' in circle should not be counted. This non-overlapping requirement prevents over-counting the support of long patterns, and makes the set of instances counted in the support informative to users.

2) We use the *maximum* number of non-overlapping instances to measure how frequently a pattern appears in a database (capturing as many non-overlapping instances as possible). We emphasize this issue, also because when 1) is obeyed, there are still different ways to capture the non-overlapping instances. In Figure 1, if alternatively, in S_1, we pair 'A' in square with 'B' in circle, and 'A' in circle with 'B' in triangle, as two instances of AB, there is no more non-overlapping instance in S_1, and we get only 3 instances of AB in total, rather than 4.

TABLE I

DIFFERENT TYPES OF RELATED WORK

	Input	Apriori Property	Output Patterns	Repetitions of Patterns in Each Sequence	Constraint of Instances (Occurrences) Counted in the Support
Agrawal and Srikant [1]	Multiple sequences	Yes	All/Closed/Maximal	Ignore	Subsequences
Manilla et al. [2]	One sequence	Yes	All	Capture	Fixed-width windows or minimal windows
Zhang et al. [6]	One sequence	No	All	Capture	Subsequences satisfying "gap requirement"
El-Ramly et al. [4]	Multiple sequences	No	All/Maximal	Capture	Substrings with first / last event matched
Lo et al. [7]	Multiple sequences	Yes (Weak)	All/Closed	Capture	Subsequences following MSC/LSC semantics
This paper	Multiple sequences	Yes	All/Closed	Capture	Non-overlapping subsequences

Our support definition, *repetitive support*, and its semantic property will be elaborated in Section II-A based on 1) and 2) above. It will be shown that our support definition preserves the *Apriori property*, "*any super-pattern of a nonfrequent pattern cannot be frequent* [8]," which is essential for designing efficient mining algorithms and defining *closed patterns*.

Problem Statement. The problem of *mining (closed) repetitive gapped subsequences*: Given a sequence database SeqDB and a support threshold min_sup, find the complete set of (closed) patterns with *repetitive support* no less than min_sup.

Our repetitive pattern mining problem defined above is crucial in the scenarios where the repetition of a pattern within each sequence is important. Following are some examples.

Repetitive subsequences may correspond to frequent customer behaviors over a set of long historical purchase records. In Example 1.1, given historical purchase records S_1 and S_2, some patterns (behaviors), like CD, might appear in every sequence, but only once in each sequence; some others, like AB, might not only appear in every sequence, but also repeat frequently within some sequences. Sequential pattern mining [1] cannot differentiate these two kinds of patterns. The difference between them can be found by our repetitive pattern mining, and used to guide a marketing strategy.

Repetitive subsequences can represent frequent software behavioral patterns. There are recent interests in analyzing program execution traces for behavioral models [9], [10], [11], [7], [12], and [13]. Since a program is composed of different paths depending on the input, a set of program traces each corresponding to potentially different sequences should be analyzed. Also, because of the existence of loops, patterns of interests can repeat multiple times within each sequence, and the corresponding instances may contain arbitrarily large gaps. Frequent usage patterns may either appear in many different traces or repeat frequently in only a few traces. In Example 1.1, AB is considered more frequent than CD, because AB appears 3 times in S_1. Given program execution traces, these patterns can aid users to understand existing programs [7], verify a system [14], [15], re-engineer a system [4], prioritize tests for regression tests [16], and is potentially useful for finding bugs and anomalies in a program [17].

Related Work. Our work is a variant of sequential pattern mining, first introduced by Agrawal and Srikant [1], and further studied by many, with different methods proposed, such as PrefixSpan by Pei *et al.* [3] and SPAM by Ayres *et al.* [18]. Recently, there are studies on mining only representative patterns, such as closed sequential patterns by Yan *et al.* [5] and Wang and Han [19]. However, different from ours, sequential pattern mining ignores the (possibly frequent) repetitions of a patterns in a particular sequence. The support of a pattern is the number of sequences containing this pattern. In Example 1.1, both patterns AB and CD have support 2.

Consider a larger example: In a database of 100 sequences, $S_1 = \ldots = S_{50} = CABABABABABD$ and $S_{51} = \ldots = S_{100} = ABCD$. In sequential pattern mining, both AB and CD have support 100. It is undesirable to consider AB and CD equally frequent for two reasons: 1) AB appears more frequently than CD does in the whole database, because it repeats more frequently in $S_1, ..., S_{50}$; 2) mining AB can help users notice and understand the difference between $S_1, ..., S_{50}$ and $S_{51}, ..., S_{100}$, which is useful in the applications mentioned above. In our repetitive support definition, we differentiate AB from CD: $\mathsf{sup}(AB) = 5 \cdot 50 + 50 = 300$ and $\mathsf{sup}(CD) = 100$.

There are also studies ([2], [4], [6], [7]) on mining the repetition of patterns within sequences.

In episode mining by Manilla *et al.* [2], a single sequence is input, and a pattern is called a (*parallel*, *serial*, or *composite*) *episode*. There are two definitions of the support of an episode ep: (i) the number of width-w windows (substrings) which contain ep as subsequences; and (ii) the number of minimal windows which contain ep as subsequences. In both cases, the occurrences of a pattern, as a series of events occurring close together, are captured as substrings, and they may overlap. In Example 1.1, in definition (i), for $w = 4$, serial episode AB has support 4 in S_1 (because 4 width-4 windows $[1, 4]$, $[2, 5]$, $[4, 7]$, and $[5, 8]$ contain AB), but some occurrence of AB, like 'A' in circle with 'B' in circle, are not captured because of the width constraint; in definition (ii), the support of AB is 2, and only two occurrences ('A' in circle with 'B' in square and 'A' in triangle with 'B' in circle) are captured. Casas-Garriga replaces the fixed-width constraint with a gap constraint [20].

In DNA sequence mining, Zhang *et al.* [6] introduce *gap requirement* in mining periodic patterns from sequences. In particular, all the occurrences (both overlapping ones and non-overlapping ones) of a pattern in a sequence satisfying the gap requirement are captured, and the support is the total number of such occurrences. The support divided by N_l is a normalized value, *support ratio*, within interval $[0, 1]$, where N_l is the maximum possible support given the gap requirement. In Example 1.1, given requirement "gap ≥ 0 and ≤ 3", pattern AB has support 4 in S_1 ('A' and 'B' can have 0-3 symbols between them), and its support ratio is $4/22$.

El-Ramly *et al.* [4] study mining user-usage scenarios of GUI-based program composed of screens. These scenarios are termed as interaction patterns. The support of such a

pattern is defined as the number of substrings, where (i) the pattern is contained as subsequences, and (ii) the first/last event of each substring matches the first/last event of the pattern, respectively. In Example 1.1, AB has support 9, with 8 substrings in S_1, $(1,3)$, $(1,7)$, ..., $(6,7)$, and $(6,8)$, captured.

Lo *et al.* [7] propose iterative pattern mining, which captures occurrences in the semantics of Message Sequence Chart/Live Sequence Chart, a standard in software modeling. Specifically, an occurrence of a pattern $e_1 e_2 \ldots e_n$ is captured in a substring obeying QRE (e_1 $\mathcal{G}^*$ e_2 $\mathcal{G}^*$ $\ldots$ $\mathcal{G}^*$ e_n), where $\mathcal{G}$ is the set of all events except $\{e_1, \ldots, e_n\}$, and $*$ is Kleene star. The support of a pattern is the number of all such occurrences. In Example 1.1, pattern AB has support 3: 'A' in circle with 'B' in square and 'A' in triangle with 'B' in circle are captured in S_1; 'A' in hexagon with 'B' in hexagon is captured in S_2.

In Table I, some important features of our work are compared with the ones of different types of related work.

Contributions. We propose and study the problem of mining repetitive gapped subsequences. Our work complements existing work on sequential pattern mining. Our definition of instance/support takes both the occurrences of a pattern repeating in different sequences and those repeating within each sequence into consideration, which captures interesting repetitive patterns in various domains with long sequences, such as customer purchase histories and software traces. For low support threshold in large datasets, the amount of frequent patterns could be too large for users to browse and to understand. So we also study finding closed patterns. A performance study on various datasets shows the efficiency of our mining algorithms. A case study has also been conducted to show the utility of our approach in extracting behaviors from software traces of an industrial system; and the result shows that our repetitive patterns can provide additional information that complements the result found by a past study on mining iterative patterns from software traces [7].

Different from the projected database operation used by PrefixSpan [3], CloSpan [5], and BIDE [19], we propose a different operation to grow patterns, which we refer to as *instance growth*. Instance growth is designed to handle repetitions of a pattern within each sequence, and to facilitate computing the maximum number of non-overlapping instances.

For mining all frequent patterns, instance growth is embedded into the depth-first pattern growth framework. For mining closed patterns, we propose *closure checking* to rule out non-closed ones on-the-fly without referring to previously generated patterns, and propose *landmark border checking* to prune the search space. Experiments show the number of closed frequent patterns is much less than the number of all frequent ones, and our closed-pattern mining algorithm is sped up significantly with these two checking strategies.

Organization. Section II gives the problem definition formally and preliminary analysis. Section III describes the *instance growth* operation, followed by the design and analysis of our two algorithms: GSgrow for mining all frequent patterns and CloGSgrow for mining closed ones. Section IV presents the results of our experimental study performed on both synthetic and real datasets, as well as a case study to show the power of our approach. Finally, Section V concludes this paper.

II. Repetitive Gapped Subsequences

In this section, we formally define the problem of mining repetitive gapped subsequences.

Let $\mathcal{E}$ be a set of distinct *events*. A *sequence* S is an ordered list of events, denoted by $S = \langle e_1, e_2, \ldots, e_{\text{length}} \rangle$, where $e_i \in \mathcal{E}$ is an event. For brevity, a sequence is also written as $S = e_1 e_2 \ldots e_{\text{length}}$. We refer to the ith event e_i in the sequence S as $S[i]$. An input *sequence database* is a set of sequences, denoted by $\mathsf{SeqDB} = \{S_1, S_2, \ldots, S_N\}$.

Definition 2.1 (Subsequence and Landmark): Sequence $S = e_1 e_2 \ldots e_m$ is a subsequence of another sequence $S' = e'_1 e'_2 \ldots e'_n$ ($m \leq n$), denoted by $S \sqsubseteq S'$ (or S' is a supersequence of S), if there exists a sequence of integers (positions) $1 \leq l_1 < l_2 < \ldots < l_m \leq n$ s.t. $S[i] = S'[l_i]$ (i.e., $e_i = e'_{l_i}$) for $i = 1, 2, \ldots, m$. Such a sequence of integers $\langle l_1, \ldots, l_m \rangle$ is called a landmark of S in S'. Note: there may be more than one landmark of S in S'. ∎

A *pattern* $P = e_1 e_2 \ldots e_m$ is also a sequence. For two patterns P and P', if P is a subsequence of P', then P is said to be a *sub-pattern* of P', and P' a *super-pattern* of P.

A. Semantics of Repetitive Gapped Subsequences

Definition 2.2 (Instances of Pattern): For a pattern P in a sequence database $\mathsf{SeqDB} = \{S_1, S_2, \ldots, S_N\}$, if $\langle l_1, \ldots, l_m \rangle$ is a landmark of pattern $P = e_1 e_2 \ldots e_m$ in $S_i \in \mathsf{SeqDB}$, pair $(i, \langle l_1, \ldots, l_m \rangle)$ is said to be an instance of P in SeqDB, and in particular, an instance of P in sequence S_i. ∎

We use $S_i(P)$ to denote *the set of instances of P in S_i*, and use $\mathsf{SeqDB}(P)$ to denote *the set of instances of P in SeqDB*. Moreover, for an *instance set* $I \subseteq \mathsf{SeqDB}(P)$, let $I_i = I \cap S_i(P) = \{(i, \langle l_1^{(k)}, \ldots, l_m^{(k)} \rangle), 1 \leq k \leq n_i\}$ be the subset of I containing the instances in S_i.

By defining "support", we aim to capture both the occurrences of a pattern repeating in different sequences and those repeating within each sequence. A naive approach is to define the support of P, $\mathsf{sup}_{\mathsf{all}}(P)$, to be the total number of instances of P in SeqDB; i.e. $\mathsf{sup}_{\mathsf{all}}(P) = |\mathsf{SeqDB}(P)|$. However, there are two problems with $\mathsf{sup}_{\mathsf{all}}(P)$: (i) We over-count the support of a long pattern because a lot of its instances overlap with each other at a large portion of positions. For example, in $\mathsf{SeqDB} = \{AABBCC \ldots ZZ\}$, pattern $ABC \ldots Z$ has support 2^{26}, but pattern AB only has support $2^2 = 4$. (ii) The violation of the *Apriori property* ($\mathsf{sup}_{\mathsf{all}}(P) < \mathsf{sup}_{\mathsf{all}}(P')$ for some P and its super-pattern P') makes it hard to define *closed patterns*, and to design efficient mining algorithm.

In our definition of *repetitive support*, we aim to avoid counting *overlapping instances* multiple times in the support value. So we first formally define overlapping instances.

Definition 2.3 (Overlapping Instances): Two instances of a pattern $P = e_1 e_2 \ldots e_m$ in $\mathsf{SeqDB} = \{S_1, S_2, \ldots, S_N\}$, $(i, \langle l_1, \ldots, l_m \rangle)$ and $(i', \langle l'_1, \ldots, l'_m \rangle)$, are overlapping if (i)

TABLE II

SIMPLE SEQUENCE DATABASE

Sequence	e_1	e_2	e_3	e_4	e_5	e_6	e_7
S_1	A	B	C	A	B	C	A
S_2	A	A	B	B	C	C	C

$i = i'$, AND (ii) $\exists 1 \leq j \leq m : l_j = l'_j$. Equivalently, $(i, \langle l_1, \ldots, l_m \rangle)$ and $(i', \langle l'_1, \ldots, l'_m \rangle)$ are non-overlapping if (i') $i \neq i'$, OR (ii') $\forall 1 \leq j \leq m : l_j \neq l'_j$. ∎

Definition 2.4 (Non-redundant Instance Set): A set of instances, $I \subseteq$ SeqDB(P), of pattern P in SeqDB is non-redundant if any two instances in I are non-overlapping. ∎

It is important to note that from (ii') in Definition 2.3, for two NON-overlapping instances $(i, \langle l_1, \ldots, l_m \rangle)$ and $(i', \langle l'_1, \ldots, l'_m \rangle)$ of the pattern $P = e_1 e_2 \ldots e_m$ in SeqDB with $i = i'$, we must have $l_j \neq l'_j$ for every $1 \leq j \leq m$, but it is possible that $l_j = l'_{j'}$ for some $j \neq j'$. We will clarify this point in the following example with pattern ABA.

Example 2.1: Table II shows a sequence database SeqDB = $\{S_1, S_2\}$. Pattern AB has 3 landmarks in S_1 and 4 landmarks in S_2. Accordingly, there are 3 instances of AB in S_1: $S_1(AB) = \{(1, \langle 1, 2 \rangle), (1, \langle 1, 5 \rangle), (1, \langle 4, 5 \rangle)\}$, and 4 instances of AB in S_2: $S_2(AB) = \{(2, \langle 1, 3 \rangle), (2, \langle 2, 3 \rangle), (2, \langle 1, 4 \rangle), (2, \langle 2, 4 \rangle)\}$. The set of instances in SeqDB: SeqDB$(AB) = S_1(AB) \cup S_2(AB)$. Instances $(i, \langle l_1, l_2 \rangle) = (1, \langle 1, 2 \rangle)$ and $(i', \langle l'_1, l'_2 \rangle) = (1, \langle 1, 5 \rangle)$ are overlapping, because $i = i'$ and $l_1 = l'_1$, i.e., they overlap at the first event, 'A' ($S_1[1] = A$). Instances $(i, \langle l_1, l_2 \rangle) = (1, \langle 1, 2 \rangle)$ and $(i', \langle l'_1, l'_2 \rangle) = (1, \langle 4, 5 \rangle)$ are non-overlapping, because $l_1 \neq l'_1$ and $l_2 \neq l'_2$. Instance sets $I^{AB} = \{(1, \langle 1, 2 \rangle), (1, \langle 4, 5 \rangle), (2, \langle 1, 3 \rangle), (2, \langle 2, 4 \rangle)\}$ and $I'^{AB} = \{(1, \langle 1, 5 \rangle), (2, \langle 2, 3 \rangle), (2, \langle 1, 4 \rangle)\}$ are both non-redundant.

Now consider pattern ABA in SeqDB. It has 3 instances in S_1: $S_1(ABA) = \{(1, \langle 1, 2, 4 \rangle), (1, \langle 1, 2, 7 \rangle), (1, \langle 4, 5, 7 \rangle)\}$, and no instance in S_2. Instances $(i, \langle l_1, l_2, l_3 \rangle) = (1, \langle 1, 2, 7 \rangle)$ and $(i', \langle l'_1, l'_2, l'_3 \rangle) = (1, \langle 4, 5, 7 \rangle)$ are overlapping, because $i = i'$ and $l_3 = l'_3$. Instances $(i, \langle l_1, l_2, l_3 \rangle) = (1, \langle 1, 2, 4 \rangle)$ and $(i', \langle l'_1, l'_2, l'_3 \rangle) = (1, \langle 4, 5, 7 \rangle)$ are non-overlapping (although $l_3 = l'_1 = 4$), because $l_1 \neq l'_1$, $l_2 \neq l'_2$, and $l_3 \neq l'_3$. So instance set $I^{ABA} = \{(1, \langle 1, 2, 4 \rangle), (1, \langle 4, 5, 7 \rangle)\}$ is non-redundant. ∎

A non-redundant instance set $I \subseteq$ SeqDB(P) is *maximal* if there is no non-redundant instance set I' of pattern P s.t. $I' \supseteq I$. To *avoid counting overlapping instances multiple times* and to *capture as many non-overlapping instances as possible*, the *support* of pattern P could be naturally defined as the size of a maximal non-redundant instance set $I \subseteq$ SeqDB(P). However, maximal non-redundant instance sets might be of different sizes. For example, in Example 2.1, for pattern AB, both non-redundant instance sets I^{AB} and I'^{AB} are maximal, but $|I^{AB}| = 4$ while $|I'^{AB}| = 3$. Therefore, our *repetitive support* is defined to be the *maximum* size of all possible non-redundant instance sets of a pattern, as the measure of how frequently the pattern occurs in a sequence database.

Definition 2.5 (Repetitive Support and Support Set): The repetitive support of a pattern P in SeqDB is defined to be

$$\text{sup}(P) = \max\{|I| \mid I \subseteq \text{SeqDB}(P) \text{ is non-redundant}\}. \quad (1)$$

The non-redundant instance set I with $|I| = $ sup(P) is called a support set of P in SeqDB. ∎

Example 2.2: Recall Example 2.1, I^{AB} and I'^{AB} are two non-redundant instance sets of pattern AB in SeqDB. It can be verified that $|I^{AB}| = 4$ is the maximum size of all possible non-redundant instance sets. Therefore, sup$(AB) = 4$, and I^{AB} is a support set. We may have more than one support set of a pattern. Another possible support set of AB is $I''^{AB} = \{(1, \langle 1, 2 \rangle), (1, \langle 4, 5 \rangle), (2, \langle 2, 3 \rangle), (2, \langle 1, 4 \rangle)\}$.

Similarly, sup$(ABA) = 2$ and I^{ABA} is a support set. ∎

To design efficient mining algorithms, it is necessary that repetitive support sup(P) defined in (1) is *polynomial-time computable*. We will show how to use our *instance grow* operation to compute sup(P) in polynomial time (w.r.t. the total length of sequences in SeqDB) in Section III-A. Note: two instances in a support set must be non-overlapping; if we replace Definition 2.3 (about "overlapping") with a stronger version, computing sup(P) will become NP-complete. [1]

Mining (Closed) Repetitive Gapped Subsequences. Based on Definition 2.5, a (closed) pattern P is said to be *frequent* if sup$(P) \geq$ min_sup, where min_sup is a specified by users. Our goal of mining repetitive gapped subsequences is to find all the frequent (closed) patterns given SeqDB and min_sup.

Considering our definition of repetitive support, our mining problem is needed in applications where the repetition of a pattern within each sequence is important.

B. Apriori Property and Closed Pattern

Repetitive support satisfies the following *Apriori property*.

Lemma 1 (Monotonicity of Support): Given two patterns P and P' in a sequence database SeqDB, if P' is a super-pattern of P ($P \sqsubseteq P'$), then sup$(P) \geq$ sup(P').

Proof: We claim: if $P \sqsubseteq P'$, for a support set I^* of P' (i.e., $I^* \subseteq$ SeqDB(P') is non-redundant and $|I^*| = $ sup(P')), we can construct a non-redundant instance set $\hat{I} \subseteq$ SeqDB(P) s.t. $|\hat{I}| = |I^*|$. Then it suffices to show

$$\text{sup}(P) = \max\{|I| \mid I \subseteq \text{SeqDB}(P) \text{ is non-redundant}\}$$
$$\geq |\hat{I}| = |I^*| = \text{sup}(P').$$

---

[1] A stronger version of Definition 2.3: changing (ii) into "$\exists 1 \leq j \leq m$ and $1 \leq j' \leq m : l_j = l'_{j'}$," and (ii') into "$\forall 1 \leq j \leq m$ and $1 \leq j' \leq m : l_j \neq l'_{j'}$." Based on this stronger version, re-examine pattern ABA in Example 2.1 and 2.2: its instances $(i, \langle l_1, l_2, l_3 \rangle) = (1, \langle 1, 2, 4 \rangle)$ and $(i', \langle l'_1, l'_2, l'_3 \rangle) = (1, \langle 4, 5, 7 \rangle)$ will be overlapping (because $l_3 = l'_1$), and thus sup$(ABA) = 1$ rather than 2 (because I^{ABA} is no longer a feasible support set).

With this stronger version of Definition 2.3, computing sup(P) becomes NP-complete, which can be proved by the reduction of the *iterated shuffle problem*. The iterated shuffle problem is proved to be NP-complete in [21].

Given an alphabet $\mathcal{E}$ and two strings $v, w \in \mathcal{E}^*$, the *shuffle* of v and w is defined as $v \odot w = \{v_1 w_1 v_2 w_2 \ldots v_k w_k \mid v_i, w_i \in \mathcal{E}^*$ for $1 \leq i \leq k, v = v_1 \ldots v_k$, and $w = w_1 \ldots w_k\}$. The *iterated shuffle* of v is $\{\lambda\} \cup \{v\} \cup (v \odot v) \cup (v \odot v \odot v) \cup (v \odot v \odot v \odot v) \cup \ldots$, where λ is an empty string. For example, $w = AABBAB$ is in the iterated shuffle of $v = AB$, because $w \in (v \odot v \odot v)$; but $w = ABBA$ is not in the iterated shuffle of v. Given two strings w and v, the iterated shuffle problem is to determine whether w is in the iterated shuffle of v.

The idea of the reduction from the iterated shuffle problem to the problem of computing sup(P) (under the stronger version of Definition 2.3) is: given strings w and v (with string length $|w| = k|v|$), let pattern $P = v$ and database SeqDB = $\{w\}$; w is in the iterated shuffle of $v \Leftrightarrow$ sup$(P) = k$.

To prove the above claim, w.l.o.g., let $P' = e_1 \ldots e_{j-1} e_j e_{j+1} \ldots e_m$ and $P = e_1 \ldots e_{j-1} e_{j+1} \ldots e_m$, i.e., P' is obtained by inserting e_j into P. Given a support set I^* of P', for each instance ins $= (i, \langle l_1, \ldots, l_{j-1}, l_j, l_{j+1}, \ldots, l_m \rangle) \in I^*$, we delete l_j from the landmark to construct ins$_{-j} = (i, \langle l_1, \ldots, l_{j-1}, l_{j+1}, \ldots, l_m \rangle)$, and add ins$_{-j}$ into $\hat{I}$.

Obviously, ins$_{-j}$ constructed above is an instance of P. For any two instances ins and ins$'$ in I^* s.t. ins $\neq$ ins$'$, we have ins$_{-j} \neq$ ins$'_{-j}$ and they are non-overlapping. Therefore, the instance set $\hat{I}$ constructed above is non-redundant, and $|\hat{I}| = |I^*|$, which completes our proof. ∎

Theorem 1 is an immediate corollary of Lemma 1.

Theorem 1 (Apriori Property): If P is not frequent, any of its super-patterns is not frequent either. Or equivalently, if P is frequent, all of its sub-patterns are frequent. ∎

Definition 2.6 (Closed Pattern): A pattern P is closed in a sequence database SeqDB if there exists NO super-pattern P' ($P \sqsubseteq P'$) s.t. $\mathsf{sup}(P) = \mathsf{sup}(P')$. P is non-closed if there exists a super-pattern P' s.t. $\mathsf{sup}(P) = \mathsf{sup}(P')$. ∎

Lemma 2 (Closed Pattern and Support Set): In a sequence database SeqDB, consider a pattern P and its super-pattern P': $\mathsf{sup}(P') = \mathsf{sup}(P)$ if and only if for any support set I' of P', there exists a support set I of P, s.t.

(i) for each instance $(i', \langle l'_1, \ldots, l'_{|P'|} \rangle) \in I'$, there exists a unique instance $(i, \langle l_1, \ldots, l_{|P|} \rangle) \in I$, where $i' = i$ and landmark $\langle l_1, \ldots, l_{|P|} \rangle$ is a subsequence of landmark $\langle l'_1, \ldots, l'_{|P'|} \rangle$; and

(ii) for each instance $(i, \langle l_1, \ldots, l_{|P|} \rangle) \in I$, there exists a unique instance $(i', \langle l'_1, \ldots, l'_{|P'|} \rangle) \in I'$, where $i' = i$ and landmark $\langle l'_1, \ldots, l'_{|P'|} \rangle$ is a supersequence of landmark $\langle l_1, \ldots, l_{|P|} \rangle$.

Proof: Direction "$\Leftarrow$" is trivial. To prove direction "$\Rightarrow$", let $I^* = I'$: such an I can be constructed in the same way as the construction of $\hat{I}$ in the proof of Lemma 1, and the proof can be completed because $\mathsf{sup}(P') = \mathsf{sup}(P)$. ∎

For a pattern P and its super-pattern P' with $\mathsf{sup}(P) = \mathsf{sup}(P')$, conditions (i)-(ii) in Lemma 2 imply a one-to-one correspondence between instances of pattern P in a support set I and those of P' in a support set I'. In particular, instances of P' in I' can be obtained by extending instances of P in I (since landmark $\langle l_1, \ldots, l_{|P|} \rangle$ is a subsequence of landmark $\langle l'_1, \ldots, l'_{|P'|} \rangle$). So it is redundant to store both patterns P' and P, and we define the closed patterns. Moreover, because of the equivalence between "$\mathsf{sup}(P) = \mathsf{sup}(P')$" and conditions (i)-(ii) in Lemma 2, we can define closed patterns merely based on the repetitive support values (as in Definition 2.6).

Example 2.3: Consider SeqDB in Table II. It is shown in Example 2.2 that $\mathsf{sup}(AB) = 4$. We also have $\mathsf{sup}(ABC) = 4$, and a support set of ABC is $I^{ABC} = \{(1, \langle 1, 2, 3 \rangle),$ $(1, \langle 4, 5, 6 \rangle), (2, \langle 1, 3, 5 \rangle), (2, \langle 2, 4, 6 \rangle)\}$. Since $\mathsf{sup}(AB) = \mathsf{sup}(ABC)$, AB is not a closed pattern, and direction "$\Rightarrow$" of Lemma 2 can be verified here: for support set I^{ABC} of ABC, there exists a support set of AB, $I^{AB} = \{(1, \langle 1, 2 \rangle),$ $(1, \langle 4, 5 \rangle), (2, \langle 1, 3 \rangle), (2, \langle 2, 4 \rangle)\}$, s.t. landmarks $\langle 1, 2 \rangle$, $\langle 4, 5 \rangle$,

$\langle 1, 3 \rangle$, and $\langle 2, 4 \rangle$ are subsequences of landmarks $\langle 1, 2, 3 \rangle$, $\langle 4, 5, 6 \rangle$, $\langle 1, 3, 5 \rangle$, and $\langle 2, 4, 6 \rangle$, respectively. ∎

III. EFFICIENT MINING ALGORITHMS

In this section, given a sequence database SeqDB and a support threshold min_sup, we introduce algorithms GS-grow for mining frequent repetitive **g**apped **s**ubsequences and CloGSgrow for mining **clo**sed frequent **g**apped **s**ubsequences. We start with introducing an operation, *instance growth*, used to compute the repetitive support $\mathsf{sup}(P)$ of a pattern P in Section III-A. We then show how to embed this operation into depth-first *pattern growth* procedure with the Apriori property for mining all frequent patterns in Section III-B. The algorithm with effective pruning strategy for mining closed frequent patterns is presented in Section III-C. Finally, we analyze the complexity of our algorithms in Section III-D.

Different from the projected database operation used in sequential pattern mining (like [3], [5], and [19]), our instance growth operation is designed to avoid overlaps of the repetitions of a pattern within each sequence in the pattern growth procedure. It keeps track of a set of non-overlapping instances of a pattern to facilitate computing its repetitive support (i.e., the maximum number of non-overlapping instances).

A. Computing Repetitive Support using Instance Growth

The maximization operator in Equation (1) makes it non-trivial to compute the repetitive support $\mathsf{sup}(P)$ of a given pattern P. We introduce a greedy algorithm to find $\mathsf{sup}(P)$ in this subsection. This algorithm, based on instance growth, can be naturally extended for depth-first pattern growth, to mine frequent patterns utilizing the Apriori property.

We define the *right-shift order of instances*, which is used in our **ins**tance **grow**th operation INSgrow (Algorithm 2).

Definition 3.1 (Right-Shift Order of Instances): Given two instances $(i, \langle l_1, \ldots, l_m \rangle)$ and $(i', \langle l'_1, \ldots, l'_m \rangle)$ of a pattern P in a sequence database SeqDB, $(i, \langle l_1, \ldots, l_m \rangle)$ is said to come before $(i', \langle l'_1, \ldots, l'_m \rangle)$ in the right-shift order if $(i < i') \vee (i = i' \wedge l_m < l'_m)$. ∎

The following example is used to illustrate the intuition of instance growth operation INSgrow for computing $\mathsf{sup}(P)$.

Example 3.1: Table III shows a more involved sequence database SeqDB. We compute $\mathsf{sup}(ACB)$ in the way illustrated in Table IV. We explain the three steps as follows:

1) Find a support set I^A of A (the 1st column). Since there is only one event, I^A is simply the set of all instances.
2) Find a support set I^{AC} of AC (the 2nd column). Extend each instance in I^A in the right-shift order (recall Definition 3.1), adding the next available event 'C' on the right to its landmark. There is no event 'C' left for extending $(2, \langle 7 \rangle)$, so we stop at $(2, \langle 5 \rangle)$.
3) Find a support set I^{ACB} of ACB (the 3rd column). Similar to step 2, for there is no 'B' left for $(2, \langle 5, 6 \rangle)$, we stop at $(2, \langle 1, 2 \rangle)$. Note $(1, \langle 4, 5 \rangle)$ cannot be extended as $(1, \langle 4, 5, 6 \rangle)$ (instances $(1, \langle 1, 3, 6 \rangle)$ and $(1, \langle 4, 5, 6 \rangle)$ are overlapping). We get $\mathsf{sup}(ACB) = 3$.

Sequence	e_1	e_2	e_3	e_4	e_5	e_6	e_7	e_8	e_9
S_1	A	B	C	A	C	B	D	D	B
S_2	A	C	D	B	A	C	A	D	D

Support set I^A	Support set I^{AC}	Support set I^{ACB}
$(1, \langle 1 \rangle) \rightarrow$	$(1, \langle 1, 3 \rangle) \rightarrow$	$(1, \langle 1, 3, 6 \rangle)$
$(1, \langle 4 \rangle) \rightarrow$	$(1, \langle 4, 5 \rangle) \rightarrow$	$(1, \langle 4, 5, 9 \rangle)$
$(2, \langle 1 \rangle) \rightarrow$	$(2, \langle 1, 2 \rangle) \rightarrow$	$(2, \langle 1, 2, 4 \rangle)$
$(2, \langle 5 \rangle) \rightarrow$	$(2, \langle 5, 6 \rangle) \rightarrow$	
$(2, \langle 7 \rangle) \rightarrow$		
$\mathsf{sup}(A) = 5$	$\mathsf{sup}(AC) = 4$	$\mathsf{sup}(ACB) = 3$

3') To compute $\mathsf{sup}(ACA)$, we start from step 2 and change step 3. To get a support set I^{ACA} of ACA, similarly, extend instances in I^{AC} in the right-shift order: $(1, \langle 1, 3 \rangle) \rightarrow (1, \langle 1, 3, 4 \rangle)$, $(2, \langle 1, 2 \rangle) \rightarrow (2, \langle 1, 2, 5 \rangle)$, and $(2, \langle 5, 6 \rangle) \rightarrow (2, \langle 5, 6, 7 \rangle)$. There is no '$A$' left for $(1, \langle 4, 5 \rangle)$. We get $I^{ACA} = \{(1, \langle 1, 3, 4 \rangle), (2, \langle 1, 2, 5 \rangle), (2, \langle 5, 6, 7 \rangle)\}$ and $\mathsf{sup}(ACA) = 3$. Note: $(2, \langle 1, 2, 5 \rangle)$ and $(2, \langle 5, 6, 7 \rangle)\}$ are non-overlapping ($e_5 = A$ in S_2 appears twice but as different 'A's in pattern ACA; recall Definition 2.3 and pattern ABA in Example 2.1).■

We formalize the method we used to compute $\mathsf{sup}(P)$ in Example 3.1 as Algorithm 1, called $\mathsf{supComp}$. Given a sequence database SeqDB and a pattern P, it outputs a support set I of P in SeqDB. The main idea is to couple *pattern growth* with *instance growth*. Initially, let I be a support set of size-1 pattern e_1; in each of the following iterations, we extend I from a support set of $e_1 \ldots e_{j-1}$ to a support set of $e_1 \ldots e_{j-1} e_j$ by calling $\mathsf{INSgrow}(\mathsf{SeqDB}, e_1 \ldots e_{j-1}, I, e_j)$. It is important to maintain I to be *leftmost* (Definition 3.2), so as to ensure the output of $\mathsf{INSgrow}(\mathsf{SeqDB}, e_1 \ldots e_{j-1}, I, e_j)$ in line 3 is a support set of $e_1 \ldots e_{j-1} e_j$ as a loop invariant. So, finally, a support set of P is returned.

To prove the correctness of $\mathsf{supComp}$, we formally define leftmost support sets and analyze subroutine $\mathsf{INSgrow}$.

Definition 3.2 (Leftmost Support Set): A support set I of pattern P in SeqDB is said to be leftmost, if: let $I = \{(i^{(k)}, \langle l_1^{(k)}, \ldots, l_m^{(k)} \rangle), 1 \leq k \leq \mathsf{sup}(P)\}$ (sorted in the right-shift order for $k = 1, 2, \ldots, \mathsf{sup}(P)$); for any other support set I' of P, $I' = \{(i'^{(k)}, \langle l_1'^{(k)}, \ldots, l_m'^{(k)} \rangle), 1 \leq k \leq \mathsf{sup}(P)\}$ (also sorted in the right-shift order, and thus $i^{(k)} = i'^{(k)}$), we have $l_j^{(k)} \leq l_j'^{(k)}$ for all $1 \leq k \leq \mathsf{sup}(P)$ and $1 \leq j \leq m$. ■

Example 3.2: Consider SeqDB in Table III. $I = \{(1, \langle 1, 2 \rangle), (1, \langle 4, 9 \rangle), (2, \langle 1, 4 \rangle)\}$ is a support set of AB, but I is NOT leftmost, because there is another support set $I' = \{(1, \langle 1, 2 \rangle), (1, \langle 4, 6 \rangle), (2, \langle 1, 4 \rangle)\}$ s.t. $l_2^{(2)} = 9 > l_2'^{(2)} = 6$. ■

Definition 3.3 (Pattern Growth '∘'): For a pattern $P = e_1 e_2 \ldots e_m$, pattern $e_1 e_2 \ldots e_m e$ is said to be a growth of P with event e, denoted by $P \circ e$. Given another pattern

---

Algorithm 1 $\mathsf{supComp}(\mathsf{SeqDB}, P)$: Compute Support (Set)

Input: sequence database $\mathsf{SeqDB} = \{S_1, S_2, \ldots, S_N\}$; pattern $P = e_1 e_2 \ldots e_m$.
Output: a leftmost support set I of pattern P in SeqDB.

1: $I \leftarrow \{(i, \langle l_1 \rangle) \mid \text{for some } i, S_i[l_1] = e_1\}$;
2: **for** $j = 2$ to m **do**
3:    $I \leftarrow \mathsf{INSgrow}(\mathsf{SeqDB}, e_1 \ldots e_{j-1}, I, e_j)$;
4: **return** I ($|I| = \mathsf{sup}(P)$);

Algorithm 2 $\mathsf{INSgrow}(\mathsf{SeqDB}, P, I, e)$: Instance Growth

Input: sequence database $\mathsf{SeqDB} = \{S_1, S_2, \ldots, S_N\}$; pattern $P = e_1 e_2 \ldots e_{j-1}$; leftmost support set I of P; event e.
Output: a leftmost support set I^+ of pattern $P \circ e$ in SeqDB.

1: **for each** $S_i \in \mathsf{SeqDB}$ **s.t.** $I_i = I \cap S_i(P) \neq \varnothing$ (P has instances in S_i) **in the ascending order of** i **do**
2:    $\mathsf{last\_position} \leftarrow 0$, $I_i^+ \leftarrow \varnothing$;
3:    **for each** $(i, \langle l_1, \ldots, l_{j-1} \rangle) \in I_i = I \cap S_i(P)$
   in the right-shift order (ascending order of l_{j-1}**) do**
4:       $l_j \leftarrow \mathsf{next}(S_i, e, \max\{\mathsf{last\_position}, l_{j-1}\})$;
5:       **if** $l_j = \infty$ **then break**;
6:       $\mathsf{last\_position} \leftarrow l_j$;
7:       $I_i^+ \leftarrow I_i^+ \cup \{(i, \langle l_1, \ldots, l_{j-1}, l_j \rangle)\}$;
8: **return** $I^+ = \cup_{1 \leq i \leq N} I_i^+$;

Subroutine $\mathsf{next}(S, e, \mathsf{lowest})$
Input: sequence S; item e; integer lowest.
Output: minimum l s.t. $l > \mathsf{lowest}$ and $S[l] = e$.

9: **return** $\min\{l \mid S[l] = e \text{ and } l > \mathsf{lowest}\}$;

$Q = e_1' e_2' \ldots e_n'$, pattern $e_1 \ldots e_m e_1' \ldots e_n'$ is said to be a growth of P with Q, denoted by $P \circ Q$. ■

Instance Growth (Algorithm 2): Instance growth operation $\mathsf{INSgrow}(\mathsf{SeqDB}, P, I, e)$, is an important routine for computing repetitive support, as well as mining (closed) frequent patterns. Given a leftmost support set I of pattern P in SeqDB and an event e, it extends I to a leftmost support set I^+ of $P \circ e$. To achieve this, for each instance $(i, \langle l_1, \ldots, l_{j-1} \rangle) \in I_i = I \cap S_i(P)$ (lines 3-7), we find the minimum l_j, s.t. $l_j > \max\{\mathsf{last\_position}, l_{j-1}\}$ and $S_i[l_j] = e$, by calling $\mathsf{next}(S_i, e, \max\{\mathsf{last\_position}, l_{j-1}\})$ (line 4). When such l_j cannot be found ($l_j = \infty$), stop scanning I_i. Because $S_i[l_j] = e$ and $l_j > l_{j-1}$, we have $(i, \langle l_1, \ldots, l_{j-1}, l_j \rangle)$ is an instance of $P \circ e$, and it should be added into I_i^+ (line 7). What's more, since $l_j > \mathsf{last\_position}$ and $\mathsf{last\_position}$ is equal to l_j found in the last iteration (line 6), it follows that I_i^+ is non-redundant (no two instances in I_i^+ are overlapping), and instances are added into I_i^+ in the right-shift order.

Lemma 3 (Non-Redundant/Right-Shift in Instance Growth): In $\mathsf{INSgrow}(\mathsf{SeqDB}, P, I, e)$ (Algorithm 2), $I^+ = \cup_{1 \leq i \leq N} I_i^+$ is finally a non-redundant instance set of pattern $P \circ e$, and these instances are inserted into I^+ in the right-shift order.

Proof: Directly from the analysis above. ■

We then show I^+ is actually a leftmost support set of $P \circ e$.

Lemma 4 (Correctness of Instance Growth): Given a leftmost support set I of pattern $P = e_1 \ldots e_{j-1}$ in SeqDB and an event e, INSgrow(SeqDB, P, I, e) (Algorithm 2) correctly computes a leftmost support set I^+ of pattern $P \circ e$.

Proof: For each $S_i \in$ SeqDB and instances in $I_i = I \cap S_i(P) = \{(i, \langle l_1^{(k)}, \ldots, l_{j-1}^{(k)} \rangle), 1 \leq k \leq n_i\}$ sorted in the right-shift order, INSgrow gets $I_i^+ = \{(i, \langle l_1^{(k)}, \ldots, l_{j-1}^{(k)}, l_j^{(k)} \rangle), 1 \leq k \leq n_i^+\}$ also in the right-shift order (ascending order of $l_j^{(k)}$).

(i) We prove $I^+ = \cup_{1 \leq i \leq N} I_i^+$ is a support set of $P \circ e$. From Lemma 3, I^+ is a non-redundant instance set of pattern $P \circ e$, so we only need to prove $|I^+| = \sup(P \circ e)$. For the purpose of contradiction, if $|I^+| < \sup(P \circ e)$, then for some S_i, there exists a non-redundant instance set I_i^* of $P \circ e$ in S_i s.t. $|I_i^*| > |I_i^+|$. Let $I_i^* = \{(i, \langle l_1'^{(k)}, \ldots, l_{j-1}'^{(k)}, l_j'^{(k)} \rangle), 1 \leq k \leq n_i^*\}$, where $n_i^+ < |I_i^*| = n_i^* \leq n_i$. Suppose I_i^* is sorted in the ascending order of $l_{j-1}'^{(k)}$, without loss of generality, we can assume $l_j'^{(k)}$'s are also in the ascending order for $k = 1, 2, \ldots, n_i^*$. Otherwise, if for some k, $l_j'^{(k-1)} > l_j'^{(k)}$, then we can safely swap $l_j'^{(k-1)}$ and $l_j'^{(k)}$. I_i^* is still a non-redundant instance set after swapping. For I is a leftmost support set, we have $l_{j-1}^{(k)} \leq l_{j-1}'^{(k)} < l_j'^{(k)}$ for $1 \leq k \leq n_i^*$. From $l_{j-1}^{(1)} < l_j'^{(1)}$ and the choice of $l_j^{(1)}$ (in line 4), we have $l_j^{(1)} \leq l_j'^{(1)}$. From $l_{j-1}^{(2)} < l_j'^{(2)}$ and last_position $= l_j^{(1)} \leq l_j'^{(1)}$, we have $l_j^{(2)} \leq l_j'^{(2)}$. By induction, we have $l_j^{(n_i^+)} \leq l_j'^{(n_i^+)}$. Consider $l_j'^{(k_0)}$ for $k_0 = n_i^+ + 1 \leq n_i^*$, we have $l_j^{(n_i^+)} \leq l_j'^{(n_i^+)} < l_j'^{(k_0)}$ and $l_{j-1}^{(k_0)} \leq l_{j-1}'^{(k_0)} < l_j'^{(k_0)}$. Therefore, $(i, \langle l_1^{(k_0)}, \ldots, l_{j-1}^{(k_0)} \rangle)$ can be extended as $(i, \langle l_1^{(k_0)}, \ldots, l_{j-1}^{(k_0)}, l_j'^{(k_0)} \rangle)$ to be an instance of $P \circ e$, and this contradicts with the fact that $|I_i^+| = n_i^+ < k_0$ (INSgrow gets $l_j^{(k_0)} = \infty$ in lines 4-5). So we have $I^+ = \cup_{1 \leq i \leq N} I_i^+$ is a support set of $P \circ e$.

(ii) We prove the support set I^+ is leftmost. For any support set I^* of $P \circ e$, consider each I_i^+ and $I_i^* = I^* \cap S_i(P \circ e) = \{(i, \langle l_1'^{(k)}, \ldots, l_{j-1}'^{(k)}, l_j'^{(k)} \rangle), 1 \leq k \leq n_i^+\}$. With the inductive argument used in (i), similarly, we can show that $l_j^{(1)} \leq l_j'^{(1)}, l_j^{(2)} \leq l_j'^{(2)}, \ldots, l_j^{(n_i^+)} \leq l_j'^{(n_i^+)}$. Since I_i is leftmost, I_i^+ (gotten by adding $l_j^{(k)}$ into I_i) is also leftmost. Therefore, the support set I^+ is leftmost.

With (i) and (ii), we complete our proof. ∎

Although it is not obvious, the existence of leftmost support sets (Definition 3.2) is implied by (ii) in the above proof. Specifically, the leftmost support set of a size-1 pattern is simply the set of all the instances. The leftmost support set of a size-j pattern can be constructed from the one of its prefix pattern, a size-$(j-1)$ pattern (as in INSgrow). From Lemma 4, the support sets found by our mining algorithms (GSgrow and CloGSgrow introduced later) are leftmost.

Theorem 2 (Correctness of supComp): Algorithm 1 can compute the leftmost support set I of pattern P in SeqDB.

Proof: Initially, in line 1, I is the leftmost support set

of size-1 pattern e_1. By repeatedly applying Lemma 4 for the iterations of line 2-3, we complete our proof. ∎

In Section III-D, we will show INSgrow (Algorithm 2) runs in polynomial time (Lemma 5). Since INSgrow is called m times in supComp (given $P = e_1 e_2 \ldots e_m$), computing $\sup(P) = |I|$ with supComp only requires polynomial time (nearly linear w.r.t. the total length of sequences in SeqDB). For the space limit, we omit the detailed analysis here.

Example 3.3: Recall how we compute $\sup(ACB)$ in three steps in Example 3.1. In algorithm supComp, I is initialized as I^A in line 1 (step 1). In each of the following two iterations of lines 2-3, INSgrow(SeqDB, A, I, C) and INSgrow(SeqDB, AC, I, B) return I^{AC}, i.e., Step 2), and I^{ACB}, i.e., Step 3), respectively. Finally, I^{ACB} is returned in line 4.

Subroutine next($S, e,$ lowest) returns the next position l after position lowest in S s.t. $S[l] = e$. For example, in Step 3) of Example 3.1 (i.e., INSgrow(SeqDB, AC, I, B)), when $S_i = S_1$ and $(i, \langle l_1, \ldots, l_{j-1} \rangle) = (1, \langle 4, 5 \rangle)$, we have last_position $= 6$ (for we had $(1, \langle 1, 3 \rangle) \rightarrow (1, \langle 1, 3, 6 \rangle)$ in the previous iteration). Therefore, in line 4, we get $l_j = $ next($S_1, B, \max\{6, 5\}) = 9$, and add $(1, \langle 4, 5, 9 \rangle)$ into I_1^+ ($(1, \langle 4, 5 \rangle) \rightarrow (1, \langle 4, 5, 9 \rangle)$). ∎

B. GSgrow: *Mining All Frequent Patterns*

In this subsection, we discuss how to extend supComp (Algorithm 1) with the Apriori property (Theorem 1) and the depth-first pattern growth procedure to find all frequent patterns, which is formalized as GSgrow (Algorithm 3).

GSgrow shares similarity with other pattern-growth based algorithms, like PrefixSpan [3], in the sense that both of them traverse the pattern space in a depth-first way. However, rather than using the *projected database*, we embed the *instance growth* operation INSgrow (Algorithm 2) into the depth-first pattern growth procedure. Initially, all size-1 patterns with their support sets are found (line 3), and for each one ($P = e$), mineFre(SeqDB, P, I) is called (line 4) to find all frequent patterns (kept in Fre) with P as their prefixes.

Subroutine mineFre(SeqDB, P, I) is a DFS of the pattern space starting from P, to find all frequent patterns with P as prefixes and put them into set Fre (line 7). In each iteration of lines 8-10, a support set I^+ of pattern $P \circ e$ is found based on the support set I of P, by calling INSgrow(SeqDB, P, I, e) (line 9) and mineFre(SeqDB, $P \circ e, I^+$) is called recursively (line 10). The Apriori property (Theorem 1) can be applied to prune the pattern space for the given threshold min_sup (line 6). Finally, all frequent patterns are in the set Fre.

Theorem 3 (Correctness of GSgrow): Given a sequence database SeqDB and a threshold min_sup, Algorithm 3 can find all patterns with repetitive support no less than min_sup.

Proof: GSgrow is an (DFS) extension from supComp (Algorithm 1). Its correctness is due to Theorems 1 and 2. ∎

Example 3.4: Given SeqDB shown in Table III and min_sup $= 3$, we start with each single event e ($A, B, C,$ or D) as a size-1 pattern. For size-1 pattern A, its leftmost support set is $I = I^A$ (as in Table IV), and mineFre(SeqDB, A, I) is

Input: sequence database $\mathsf{SeqDB} = \{S_1, S_2, \ldots, S_N\}$; threshold $\mathsf{min\_sup}$.

Output: $\{P \mid \mathsf{sup}(P) \geq \mathsf{min\_sup}\}$.

1: $\mathcal{E} \leftarrow$ all events appearing in SeqDB; $\mathsf{Fre} \leftarrow \varnothing$;
2: **for each** $e \in \mathcal{E}$ **do**
3:   $P \leftarrow e$; $I \leftarrow \{(i, \langle l \rangle) \mid$ for some $i, S_i[l] = e\}$;
4:   $\mathsf{mineFre}(\mathsf{SeqDB}, P, I)$;
5: **return** Fre;

Subroutine $\mathsf{mineFre}(\mathsf{SeqDB}, P, I)$

Input: sequence database $\mathsf{SeqDB} = \{S_1, S_2, \ldots, S_N\}$; pattern $P = e_1 e_2 \ldots e_{j-1}$; support set I of pattern P in SeqDB.

Objective: add all frequent patterns with prefix P into Fre.

6: **if** $|I| \geq \mathsf{min\_sup}$ **then**
7:   $\mathsf{Fre} \leftarrow \mathsf{Fre} \cup \{P\}$;
8:   **for each** $e \in \mathcal{E}$ **do**
9:     $I^+ \leftarrow \mathsf{INSgrow}(\mathsf{SeqDB}, P, I, e)$;
10:    $\mathsf{mineFre}(\mathsf{SeqDB}, P \circ e, I^+)$;

Algorithm 4 CloGSgrow: Mining Closed Frequent Patterns

... ...
6: **if** $|I| \geq \mathsf{min\_sup}$ and $\neg(\mathsf{LBCheck}(P) = \mathsf{prune})$ **then**
7:   **if** $\mathsf{CCheck}(P) = \mathsf{closed}$ **then** $\mathsf{Fre} \leftarrow \mathsf{Fre} \cup \{P\}$;
... ...

---

called. Then in each iteration of lines 8-10, support set $I^+ = I^{AA}, \ldots, I^{AD}$ of pattern $AA, \ldots, AD$ is found (line 9), and $\mathsf{mineFre}(\mathsf{SeqDB}, A \circ e, I^+)$ is called recursively (line 10), for $e = A, \ldots, D$. Similarly, when $\mathsf{mineFre}(\mathsf{SeqDB}, P, I)$ is called for some size-2 pattern P, like $P = AA$, then support set $I^+ = I^{AAA}, \ldots, I^{AAD}$ is found, and $\mathsf{mineFre}(\mathsf{SeqDB}, P \circ e, I^+)$ is called, for $e = A, \ldots, D$. Note: if $\mathsf{sup}(P) < \mathsf{min\_sup}$ (i.e., $|I| < \mathsf{min\_sup}$), like $|I^{AAA}| = 1 < 3$, we stop growing pattern AAA because of the Apriori property (line 6). ∎

C. CloGSgrow: *Mining Closed Frequent Patterns*

From Definition 2.6 and Lemma 2, a non-closed pattern P is "redundant" in the sense that there exits a super-pattern P' of pattern P with the same repetitive support, and P''s support sets can be extended from P's support sets. In this subsection, we focus on generating the set of closed frequent patterns. Besides proposing the *closure checking* strategy to rule out non-closed patterns on-the-fly, we propose the *landmark border checking* strategy to prune the search space.

Definition 3.4 (Pattern Extension): For a pattern $P = e_1 e_2 \ldots e_m$ and one of its super-patterns P' with size $m + 1$, there are three cases: for some event e', (1) $P' = e_1 e_2 \ldots e_m e'$; (2) $\exists 1 \leq j < m : P' = e_1 \ldots e_j e' e_{j+1} \ldots e_m$; and (3) $P' = e' e_1 e_2 \ldots e_m$. In any of the three cases, P' is said to be an extension to P w.r.t. e'. ∎

Theorem 4 (Closure Checking): In SeqDB, pattern P is NOT closed iff for some event e', the extension to P w.r.t.

e', denoted by P', has support $\mathsf{sup}(P') = \mathsf{sup}(P)$.

Proof: Directly from the definition of closed patterns (Definition 2.6) and the Apriori property (Theorem 1). ∎

The above theorem shows that, to check whether a pattern P is closed, we only need to check whether there exists an extension P' to P w.r.t. some event e', s.t. $\mathsf{sup}(P) = \mathsf{sup}(P')$. This strategy can be simply embedded into GSgrow to rule out non-closed patterns from the output. But, unfortunately, we cannot prune the search space using this closure checking strategy. That means, even if we find that pattern P is NOT closed, we cannot stop growing P in line 8-10 of GSgrow (Algorithm 3). Therefore, using this strategy only, when mining closed frequent patterns, we cannot expect any efficiency improvement to GSgrow. Following is such an example.

Example 3.5: Consider SeqDB shown in Table III. Given $\mathsf{min\_sup} = 3$, AB is a frequent pattern because $\mathsf{sup}(AB) = 3$, with a leftmost support set $\{(1, \langle 1, 2 \rangle), (1, \langle 4, 6 \rangle), (2, \langle 1, 4 \rangle)\}$. AB is non-closed because pattern ACB, an extension to AB, has the same support, $\mathsf{sup}(ACB) = \mathsf{sup}(AB) = 3$ (Theorem 4). A leftmost support set of ACB is $\{(1, \langle 1, 3, 6 \rangle), (1, \langle 4, 5, 9 \rangle), (2, \langle 1, 2, 4 \rangle)\}$. Although AB is non-closed, we still need to grow AB to ABA, $\ldots$, ABD, because there may be some closed frequent pattern with AB as its prefix, like pattern ABD ($\mathsf{sup}(ABD) = 3$). ∎

The following theorem is used to prune the search space.

Theorem 5 (Landmark Border Checking): For pattern $P = e_1 e_2 \ldots e_m$ in SeqDB and an extension to P w.r.t. some event e', denoted by P', let $I = \{(i^{(k)}, \langle l_1^{(k)}, \ldots, l_m^{(k)} \rangle), 1 \leq k \leq \mathsf{sup}(P)\}$ (sorted in the right-shift order) be a leftmost support set of P, and $I' = \{(i'^{(k)}, \langle l_1'^{(k)}, \ldots, l_m'^{(k)}, l_{m+1}'^{(k)} \rangle), 1 \leq k \leq \mathsf{sup}(P')\}$ (sorted in the right-shift order) be a leftmost support set of P'. If there exists P' s.t. (i) $\mathsf{sup}(P) = \mathsf{sup}(P')$ and (ii) $l_{m+1}'^{(k)} \leq l_m^{(k)}$ for all $k = 1, 2, \ldots, \mathsf{sup}(P) = \mathsf{sup}(P')$, then there is no closed pattern with P as its prefix.

Proof: Because of (i), we have $i^{(k)} = i'^{(k)}$. The main idea of our proof is: for any pattern $P \circ Q$ with *such* P as its prefix, we replace P with P', and get pattern $P' \circ Q$; if $\mathsf{sup}(P \circ Q) = \mathsf{sup}(P' \circ Q)$, $P \circ Q$ is non-closed. In the following, we prove $\mathsf{sup}(P \circ Q) = \mathsf{sup}(P' \circ Q)$ to complete our proof.

Let $P \circ Q$ be a size-n pattern, and I'' its leftmost support set. For each instance $(i^{(k)}, \langle l_1^{(k)}, \ldots, l_m^{(k)}, l_{m+1}^{(k)}, \ldots, l_n^{(k)} \rangle) \in I''$, we have $(i^{(k)}, \langle l_1^{(k)}, \ldots, l_m^{(k)} \rangle) \in I$ is an instance of P. Replacing the prefix $\langle l_1^{(k)}, \ldots, l_m^{(k)} \rangle$ with $\langle l_1'^{(k)}, \ldots, l_m'^{(k)}, l_{m+1}'^{(k)} \rangle$ (a landmark of P'), since $l_{m+1}'^{(k)} \leq l_m^{(k)} < l_{m+1}^{(k)}$, we get an instance $(i^{(k)}, \langle l_1'^{(k)}, \ldots, l_m'^{(k)}, l_{m+1}'^{(k)}, l_{m+1}^{(k)}, \ldots, l_n^{(k)} \rangle)$ of $P' \circ Q$. It can be shown that the instances of $P' \circ Q$ constructed in this way are not overlapping. Therefore, $\mathsf{sup}(P \circ Q) \leq \mathsf{sup}(P' \circ Q)$. Because $P \circ Q$ is a sub-pattern of $P' \circ Q$, we have $\mathsf{sup}(P \circ Q) = \mathsf{sup}(P' \circ Q)$. This completes our proof. ∎

The above theorem means, if for pattern P, there exists an extension P' s.t. conditions (i) and (ii) are satisfied, then we can stop growing P in the DFS. Because there is no closed pattern with P as its prefix, growing P will not generate any

closed pattern. Although it introduces some additional cost for checking "landmark borders" $l_m^{(k)}$'s and $l'^{(k)}_{m+1}$'s, this strategy is effective for pruning the search space, and can improve the efficiency of our closed-pattern mining algorithm significantly. The improvement will be demonstrated by the experiments conducted on various datasets in Section IV.

Formally, our closed-pattern mining algorithm, CloGSgrow (Algorithm 4), is similar to GSgrow (Algorithm 3), but replaces line 6 and line 7 in GSgrow with line 6 and line 7 in CloGSgrow, respectively. Notation-wise, CCheck(P) = closed iff the closure checking (Theorem 4) implies P is closed. LBCheck(P) = prune iff P satisfies conditions (i) and (ii) in the landmark border checking (Theorem 5), which implies P is not only non-closed but also prunable.

The correctness of CloGSgrow is directly from the correctness of GSgrow (Theorem 3), and Theorems 4 and 5 above.

Example 3.6: Consider SeqDB shown in Table III, we verify Theorem 4 and 5 here. Let $P = AA$ and $e' = C$. Given min_sup = 3, AA is a frequent pattern because sup(AA) = 3. The leftmost support set of AA is $I = \{(1, \langle 1, 4\rangle), (2, \langle 1, 5\rangle), (2, \langle 5, 7\rangle)\}$. By Theorem 4, AA is not closed because pattern $P' = ACA$, an extension to $P = AA$ w.r.t. $e' = C$, has the same support, sup(ACA) = 3. The leftmost support set of ACA is $I' = \{(1, \langle 1, 3, 4\rangle), (2, \langle 1, 2, 5\rangle), (2, \langle 5, 6, 7\rangle)\}$. By Theorem 5, AA can be pruned from further growing, because any pattern with AA as its prefix is not closed (its prefix $P = AA$ can be replaced with $P' = ACA$, and the support is unchanged). We examine such a pattern, AAD. We have sup(AAD) = 3 and the leftmost support set $I'' = \{(1, \langle 1, 4, 7\rangle), (2, \langle 1, 5, 8\rangle), (2, \langle 5, 7, 9\rangle)\}$. As in the proof of Theorem 5, in I'', we can replace $\langle 1, 4\rangle$, the prefix of a landmark in I'', with $\langle 1, 3, 4\rangle$, a landmark in I'; replace $\langle 1, 5\rangle$ with $\langle 1, 2, 5\rangle$; replace $\langle 5, 7\rangle$ with $\langle 5, 6, 7\rangle$. Then we get a support set $\{(1, \langle 1, 3, 4, 7\rangle), (2, \langle 1, 2, 5, 8\rangle), (2, \langle 5, 6, 7, 9\rangle)\}$ of $ACAD$. So sup($ACAD$) = 3, and AAD is not closed.

Recall AB and its extension ACB in Example 3.5, although sup(AB) = sup(ACB), AB cannot be safely pruned because the leftmost support set of ACB has "shifted right" from the leftmost support set of AB, which violates (ii) in Theorem 5 ($6 > 2$ in the first instance, and $9 > 6$ in the second one). There are closed patterns with the prefix AB, like ABD. ∎

D. Complexity Analysis

In this subsection, we analyze the time/space complexity of our mining algorithms GSgrow and CloGSgrow. Before that, we need to introduce how the subroutine next in INSgrow (Algorithm 2) is implemented, and how instances are stored.

Inverted Event Index. Inspired by the *inverted index* used in search engine indexing algorithms, *inverted event index* is used in subroutine next. Simply put, for each event $e \in \mathcal{E}$ and $S_i \in$ SeqDB, create an ordered list $\mathcal{L}_{e,S_i} = \{j | S_i[j] = e\}$. When subroutine next($S, e$, lowest) is called, we can simply place a query, "what is the smallest element that is larger than lowest in $\mathcal{L}_{e,S}$?" If the main memory is large enough for the index structure $\mathcal{L}_{e,S_i}$'s, we can use arrays to implement them,

and apply a binary search to handle this query. Otherwise, B-trees can be employed to index $\mathcal{L}_{e,S_i}$'s. We have *the time complexity of subroutine* next(S, e, lowest) *is* $O(\log L)$, where $L = \max\{|\mathcal{L}_{e,S_i}|\} = O(\max\{|S_1|, \ldots, |S_N|\})$.

Compressed Storage of Instances. For an instance of a size-n pattern P, $(i, \langle l_1, l_2, \ldots, l_n\rangle)$, we only need to store triple (i, l_1, l_n), and keep all instances sorted in the right-shift order (ascending order of l_n). In this way, all operations related to instances in our algorithms can be done with (i, l_1, l_n). If required, the leftmost support set of P can be constructed from these triples. Details are omitted here. So in our algorithms, *we only need constant space $O(1)$ to store an instance.*

Time Complexity. We first analyze the time complexity of the instance growth operation INSgrow, and then analyze the complexity of mining all frequent patterns with GSgrow.

Lemma 5 (Time Complexity of Instance Growth INSgrow*):* Algorithm 2's time complexity is $O(\text{sup}(P) \cdot \log L)$.

Proof: Given event e, pattern P, and its leftmost support set I in SeqDB, INSgrow computes the leftmost support set I^+ of $P \circ e$. Subroutine next is called only once for each instance in I, and S_i is skipped if $S_i(P) \cap I = \varnothing$ (line 1). So the total cost is $O(|I| \cdot \log L) = O(\text{sup}(P) \cdot \log L)$. ∎

Recall Fre is the set of all frequent patterns, found by our mining algorithm GSgrow, given support threshold min_sup. Let $E = |\mathcal{E}|$ be the number of distinct events. We have the following complexity result for GSgrow (Algorithm 3).

Theorem 6 (Time Complexity of Mining All Patterns): Algorithm 3's time complexity is $O(\sum_{P \in \text{Fre}} \text{sup}(P) \cdot E \log L)$.

Proof: For each $P \in$ Fre and $e \in \mathcal{E}$, instance growth operation INSgrow to grow P to $P \circ e$ and compute its support set I^+ (line 9) is the dominating factor in the time complexity of Algorithm 3. From Lemma 5, this step uses $O(\text{sup}(P) \cdot \log L)$ time. From the Apriori property (Theorem 1) and line 6 of Algorithm 3, we know INSgrow is executed only for patterns in Fre. So the total time is $O\left(\sum_{P \in \text{Fre}} \sum_{e \in \mathcal{E}} \text{sup}(P) \cdot \log L\right) = O\left(\sum_{P \in \text{Fre}} \text{sup}(P) \cdot E \log L\right)$. ∎

The time complexity of GSgrow is nearly optimal in the sense that even if we are given the set Fre, it will take $\Omega(\sum_{P \in \text{Fre}} \text{sup}(P))$ time to compute the supports of patterns in Fre and output their support sets. For each pattern $P \in$ Fre, the additional factor, E, in the complexity of GSgrow is the time needed to enumerate possible events e's to check whether $P \circ e$ is a frequent pattern. In practice, this factor is usually not as large as $E = |\mathcal{E}|$ because we can maintain a list of possible events which are much fewer than those in $\mathcal{E}$.

It is difficult to analyze the time complexity of mining closed patterns, i.e., CloGSgrow (Algorithm 4), quantitatively, since its running time is largely determined by not only the number of closed patterns but also the structure of them. Its scalability will be evaluated experimentally in Section IV.

Space Complexity. Let sup$_{\text{max}}$ be the maximum support of (size-1) patterns in SeqDB, and len$_{\text{max}}$ the maximum length of a frequent pattern. The following theorem shows the running-

time space (not including the space consumed by the inverted event index) used in our two mining algorithm is small.

Theorem 7 (Space Complexity of Two Mining Algorithms): Besides the inverted event index $\mathcal{L}_{e,S_i}$'s, the space consumed by Algorithms 3 (i.e., GSgrow) and 4 (i.e., CloGSgrow) is $O(\text{sup}_{max} \cdot \text{len}_{max})$.

Proof: The depth of the DFS pattern growth procedure mineFre of both Algorithm 3 and 4 is bounded by len_{max}. Using the compressed storage of instances, in mineFre, for each depth, we need only $O(|I|) = O(\text{sup}(P))$ space. So the total space required is $O(\text{sup}_{max} \cdot \text{len}_{max})$. ∎

IV. PERFORMANCE AND CASE STUDY

We evaluate the scalability of our approach and conduct a case study to show its utility. All experiments were performed on an IBM X41 Intel Pentium M 1.6GHz Tablet PC with 1.5GB of RAM running Windows XP. Algorithms were written in C++. Datasets and binary codes used in our experiments are available in the first author's homepage.

A. Performance Study

In Figure 2-6, we test our two algorithms, GSgrow (mining all frequent patterns, labeled as '**All**') and CloGSgrow (mining closed patterns, labeled as '**Closed**'), to demonstrate the scalability of our approaches and the effectiveness of our search space pruning strategy (Theorem 5) when the support threshold min_sup and the size of database are varied.

Datasets. To evaluate scalability, we use three datasets: one synthetic and two real datasets. The first data set, a synthetic data generator provided by IBM (the one used in [1]), is used with modification to generate sequences of events. The data generator accepts a set of parameters, D, C, N, and S, corresponding to the *number of sequences* |SeqDB| (in 1000s), the *average number of events per sequence*, the *number of different events* (in 1000s), and the *average number of events in the maximal sequences*, respectively. The second one is a click stream dataset (*Gazelle dataset*) in KDD Cup 2000, which has been a benchmark dataset used by past studies on mining sequences, like [5], [19], and [7]. The Gazelle dataset contains 29369 sequences and 1423 distinct events. Although the average sequence length is only 3, there are a number of long sequences (the maximum length is 651), where a pattern may repeat many times. The third one is a set of software traces collected from Traffic alert and Collision Avoidance System (*TCAS dataset*) described in [7]. The TCAS dataset contains 1578 sequences and 75 distinct events. The average sequence length is 36 and the maximum length is 70.

Experiment-1 (Support Threshold). We vary support threshold min_sup on three datasets D5C20N10S20 (gotten from the data generator by setting D=5, C=20, N=10, and S=20), Gazelle, and TCAS. The results are shown in Figures 2-4. We report (a) the running time (in seconds) and (b) the number of patterns found by GSgrow and CloGSgrow.

Similar to other works on closed sequential pattern mining [5], [19], low support thresholds are used to test the scalability of CloGSgrow (mining closed patterns). In Figures 2, 3,
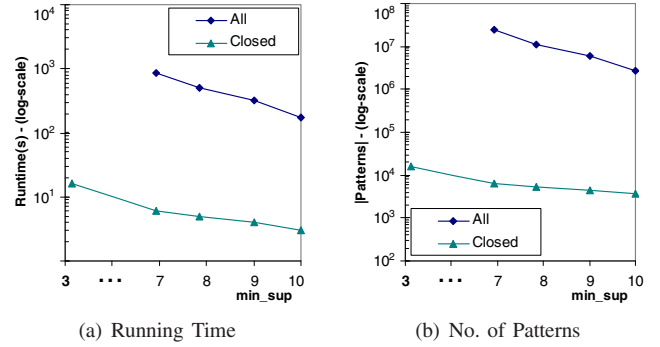

(a) Running Time (b) No. of Patterns

Fig. 2. Varying Support Threshold min_sup for D5C20N10S20 Dataset

(a) Running Time (b) No. of Patterns
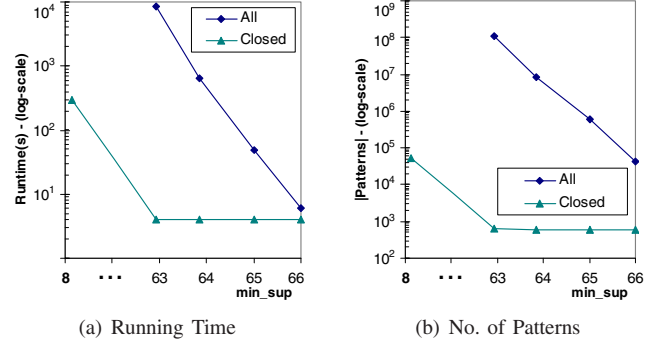
Fig. 3. Varying Support Threshold min_sup for Gazelle Dataset

(a) Running Time (b) No. of Patterns
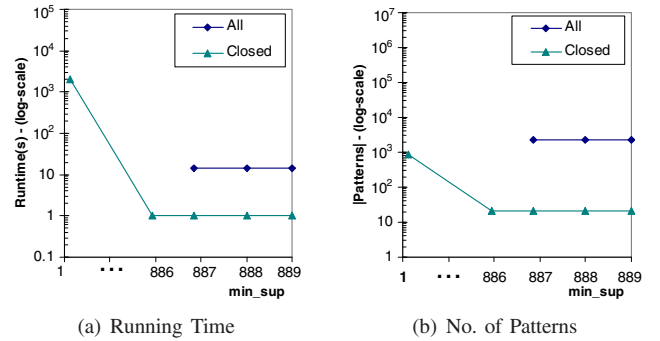
Fig. 4. Varying Support Threshold min_sup for TCAS Dataset

and 4, the points directly after "..." in the X-axis correspond to the "cut-off" points, where GSgrow (mining all patterns) takes too long to complete the computation. Only thresholds larger than these cut-off points are used in GSgrow.

For all datasets, even at very low support, CloGSgrow is able to complete within 34 minutes. TCAS dataset especially highlights performance benefit of our pruning strategy: CloGSgrow completes with the *lowest possible support threshold*, 1, within less than 34 minutes; the set of all frequent patterns cannot be found by GSgrow within excessive time (> 6 hours) even at a relatively high support threshold, 886.

The plotted result shows that the number of closed patterns is much less than the number of all frequent ones. Moreover, the search space pruning strategy (Theorem 5) for mining closed patterns significantly reduces the running time, especially when the support threshold is low. So our mining algorithms can efficiently work on various benchmark datasets with different support thresholds. Comparison between perfor-

mance of GSgrow and CloGSgrow highlights the benefit and effectiveness of our closed pattern mining algorithm.

Comparing with sequential pattern miners, our approach is slightly slower than BIDE [19] but faster than CloSpan [5] and PrefixSpan [3] on D5C20N10S20 dataset. It is slower than all the three on Gazelle dataset. It is faster than PrefixSpan on TCAS dataset. But is should be noted that our miner solves a harder problem for the consideration of repetitions both in multiple sequences and within each sequence.

Experiment-2 (Number of Sequences). In this experiment, we use the synthetic data generator to get five datasets with different total numbers of sequences (|SeqDB|). Specifically, we fix N=10 (10K different events), C=S=50 (50 events in a sequence on average), and vary D (number of sequences) from 5(K) to 25(K). Support threshold min_sup is fixed to be 20. We report (a) the running time and (b) the number of patterns found by GSgrow and CloGSgrow in Figure 5.

GSgrow cannot terminate in a reasonable amount of time when there are around 15K sequences in SeqDB. We stop it after it runs for >8 hours (we still plot a point here). On the other hand, CloGSgrow can find the closed patterns using only around 10 minutes even when there are 25K sequences.

From Figure 5(b), it should also be noted that why GSgrow cannot terminate for the 15K dataset is not simply because this algorithm is "inefficient". The main reason is: there are too many frequent patterns in this dataset for GSgrow to find them (note there are already $> 10^6$ frequent patterns in the 10K dataset). On the other hand, the number closed patterns is much less. So it is easier both for the algorithm to compute closed patterns and for the users to utilize them.

Experiment-3 (Average Sequence Length). Also, we vary the average length of sequences in SeqDB by changing parameter C and S in the synthetic data generator. Five datasets are generated by fixing D=10 (10K sequences in SeqDB), N=10 (10K different events), and varying both C and S from 20 to 100 (average length 20-100). Support threshold min_sup is fixed to be 20. We test our two mining algorithms, and report (a) the running time and (b) the number of patterns in Figure 6.

Both GSgrow and CloGSgrow consume more time, when the average length of sequences in SeqDB is larger, because more patterns can be found with the same support threshold min_sup. For the similar reason as in Experiment-1 and 2 (the number of all frequent patterns is huge), GSgrow cannot terminate in a reasonable amount of time when the average length is no less than 80. We terminate GSgrow manually after it runs for >8 hours when the average length is 80. CloGSgrow always outperforms GSgrow on efficiency and outputs much less patterns. Even when the average length is 100, CloGSgrow can terminate in around 2 hours.

B. Case Study

Repetitive gapped subsequence mining is able to capture repetitive patterns from a variety of datasets. In this case study, we investigate its power on mining frequent program behavioral patterns from program execution traces. We use the
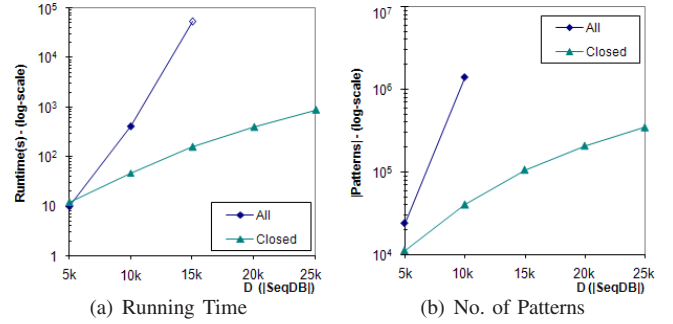

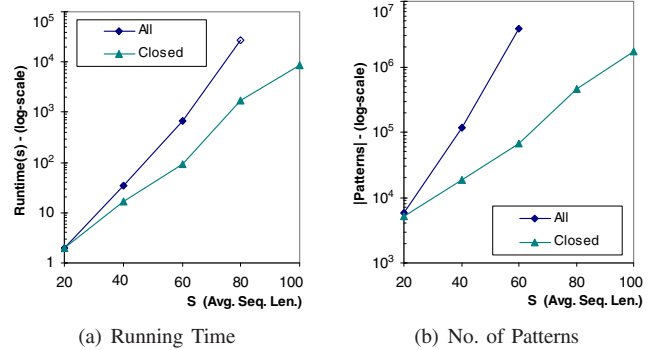Fig. 5.   Varying |SeqDB| (the Number of Sequences in Database)

Fig. 6.   Varying the Average Sequence Length in Database

dataset previously used in [7] (generated from the transaction component of JBoss Application Server). We show the benefit of our more generic pattern/instance/support definition by comparing our result to the result gotten in iterative pattern mining [7]: we are able to discover additional information from these traces using CloGSgrow.

The dataset was described in [7]. It contains 28 traces, and each consists of 91 events on average. There are 64 unique events. The longest trace is of 125 events. Using min_sup = 18, CloGSgrow completes in 5 minutes. GSgrow does not terminate even after running for >8 hours. A total of 6070 patterns are reported. This number is more than the 880 patterns mined in [7], because our pattern definition is more generic and carries less constraint. For both iterative pattern and repetitive gapped subsequences, the reported patterns are too many. So we perform the following post-processing steps adapted from the ones proposed in [7]:

1) Density: Only report patterns in which the number of unique events is >40% of its length.
2) Maximality: Only report maximal patterns.
3) Ranking: Order them according to length.

Then, 94 patterns remain. The longest pattern (Figure 7) is of length 66 and corresponds to the following behavior: Connection Set Up Evs → TxManager Set Up Evs → Transaction Set Up Evs → Resource Enlistment & Transaction Execution → Transaction Commit Evs → Transaction Disposal Evs (66 events can be divided into 6 blocks by their semantics).

Interestingly, our longest pattern contains the longest pattern (of length 32) found in iterative pattern mining [7] as a sub-pattern, but merges the two behaviors related to "resource enlistment" and "transaction commit". Specifically, before a

Connection Set Up

1. TransManLoc.getInstance
2. TransManLoc.locate
3. TransManLoc.tryJNDI
4. TransManLoc.usePrivateAPI

Tx Manager Set Up

5. TxManager.getInstance
6. TxManager.begin
7. XidFactory.newXid
8. XidFactory.getNextId
9. XidImpl.getTrulyGlobalId

Transaction Set Up

10. TransImpl.assocCurThd
11. TransImpl.lock
12. TransImpl.unlock
13. TransImpl.getLocId
14. XidImpl.getLocId
15. LocId.hashCode

Resource Enlistment & Transaction Execution

16. TxManager.getTrans
17. TransImpl.isDone
18. TransImpl.getStatus

19. TxManager.getTrans
20. TransImpl.isDone
21. TransImpl.enlistResource
22. TransImpl.lock
23. TransImpl.createXidBranch
24. XidFactory.newBranch
25. TransImpl.unlock
26. XidImpl.hashCode
27. XidImpl.hashCode
28. TransImpl.lock
29. TransImpl.unlock
30. XidImpl.hashCode
31. TxManager.getTrans
32. TransImpl.isDone
33. TransImpl.equals
34. TransImpl.getLocIdVal
35. XidImpl.getLocIdVal
36. TransImpl.getLocIdVal
37. XidImpl.getLocIdVal

Transaction Commit

38. TxManager.commit
39. TransImpl.commit
40. TransImpl.lock
41. TransImpl.beforePrepare
42. TransImpl.checkIntegrity
43. TransImpl.checkBeforeStatus
44. TransImpl.endResources
45. TransImpl.unlock
38. TxManager.commit
39. TransImpl.commit

40. TransImpl.lock
41. TransImpl.beforePrepare
42. TransImpl.checkIntegrity
43. TransImpl.checkBeforeStatus
44. TransImpl.endResources
45. TransImpl.unlock
46. XidImpl.hashCode
47. TransImpl.lock
48. TransImpl.unlock
49. XidImpl.hashCode
50. TransImpl.lock
51. TransImpl.completeTrans
52. TransImpl.cancelTimeout
53. TransImpl.unlock
54. TransImpl.lock
55. TransImpl.doAfterCompletion
56. TransImpl.unlock
57. TransImpl.lock
58. TransImpl.instanceDone

Transaction Dispose

59. TxManager.getInstance
60. TxManager.releaseTransImpl
61. TransImpl.getLocalId
62. XidImpl.getLocalId
63. LocalId.hashCode
64. LocalId.equals
65. TransImpl.unlock
66. XidImpl.hashCode

Fig. 7. Longest Repetitive Gapped Subsequence (of length 66) Mined from JBoss Transaction Component (read from top-to-bottom, left-to-right)

transaction commit, more than one resource enlistment operation can be made. In iterative pattern's definition, our longest pattern found here should be separated into two patterns. But when mining repetitive gapped subsequences, this information can be preserved, resulting in a more *complete* specification. Hence, our pattern contains *more complete information* based on our definition of *instance* and *repetitive support*.

Similar to the iterative patterns [7], our repetitive patterns can also capture more fine-grained repetitions; e.g. the most frequent pattern (a 2-event behavior): Lock → Unlock.

Some other sequences, like customer purchase histories, can be also used in our case study to find interesting behaviors.

V. Conclusion and Future Work

Much data is in sequential format, ranging from purchase histories to program traces, DNA, and protein sequences. In many of these sequential data sources, *patterns or behaviors of interests often repeat frequently within each sequence*. To capture this kind of interesting patterns, in this paper, we propose the problem of mining repetitive gapped subsequences.

Our work extends state-of-art research on sequential pattern mining, as well as episode mining. We introduce nice properties of our mining model, and efficient algorithms to mine both all and closed frequent gapped subsequences. In particular, we employ novel techniques, *instance growth* and *landmark border checking* to provide promising mining efficiency.

A performance study on several benchmark datasets shows that our closed-pattern mining algorithm is efficient even with low support thresholds. Furthermore, a case study on JBoss application server shows the utility of our algorithm in extracting behaviors from sequences generated in an industrial system. The result shows repetitive gapped subsequence mining provides additional information that complements the result found by a past study on mining iterative patterns [7].

As a promising future work, frequent repetitive gapped subsequences can be used as features for classifying sequences, like (buggy/un-buggy) program execution traces and purchase histories of different types of customers. The patterns which repeat frequently in some sequences while infrequently in others could be discriminative features for classification. Our algorithms find all frequent repetitive patterns and report their supports in each sequence as feature values; a future work is to select discriminative ones for classification.

Another possible future work is to extend our algorithms for mining *approximate* repetitive patterns with *gap constraints*, which is useful for mining subsequences from long sequences of DNA, protein, and text data.

References

[1] R. Agrawal and R. Srikant, "Mining sequential patterns," in *ICDE*, 1995.
[2] H. Mannila, H. Toivonen, and A. I. Verkamo, "Discovery of frequent episodes in event sequences," *Data Min. Knowl. Discov.*, vol. 1, no. 3, pp. 259–289, 1997.
[3] 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 *ICDE*, 2001.
[4] M. El-Ramly, E. Stroulia, and P. Sorenson, "From run-time behavior to usage scenarios: an interaction-pattern mining approach," in *KDD*, 2002.
[5] X. Yan, J. Han, and R. Afhar, "CloSpan: Mining closed sequential patterns in large datasets," in *SDM*, 2003.
[6] M. Zhang, B. Kao, D. Cheung, and K. Yip, "Mining periodic patterns with gap requirement from sequences," in *SIGMOD*, 2005.
[7] D. Lo, S.-C. Khoo, and C. Liu, "Efficient mining of iterative patterns for software specification discovery," in *KDD*, 2007.
[8] R. Agrawal and R. Srikant, "Fast algorithms for mining association rules," in *VLDB*, 1994.
[9] G. Ammons, R. Bodik, and J. R. Larus, "Mining specification," in *SIGPLAN POPL*, 2002.
[10] D. Lo and S.-C. Khoo, "SMArTIC: Toward building an accurate, robust and scalable specification miner," in *SIGSOFT FSE*, 2006.
[11] D. Lo, S. Maoz, and S.-C. Khoo, "Mining modal scenario-based specifications from execution traces of reactive systems," in *ASE*, 2007.
[12] J. Whaley, M. Martin, and M. Lam, "Automatic extraction of object oriented component interfaces," in *ISSTA*, 2002.
[13] J. Quante and R. Koschke, "Dynamic protocol recovery," in *WCRE*, 2007.
[14] D. Lo, S.-C. Khoo, and C. Liu, "Mining temporal rules from program execution traces," in *PCODA*, 2007.
[15] ——, "Efficient mining of recurrent rules from a sequence database," in *DASFAA*, 2008.
[16] T. Xie and D. Notkin, "Tool-assisted unit-test generation and selection based on operational abstractions," *Autom. Softw. Eng.*, vol. 13, no. 3, pp. 345–371, 2006.
[17] Z. Li and Y. Zhou, "PR-miner: Automatically extracting implicit programming rules and detecting violations in large software code," in *SIGSOFT FSE*, 2005.
[18] J. Ayres, J. Flannick, J. Gehrke, and T. Yiu, "Sequential pattern mining using a bitmap representation," in *KDD*, 2002.
[19] J. Wang and J. Han, "BIDE: Efficient mining of frequent closed sequences," in *ICDE*, 2004.
[20] G. Garriga, "Discovering unbounded episodes in sequential data," in *PKDD*, 2003.
[21] M. K. Warmuth and D. Haussler, "On the complexity of iterated shuffle," *J. Comput. Syst. Sci.*, vol. 28, no. 3, pp. 345–358, 1984.