Mining and Ranking Generators of Sequential Patterns

David Lo*   Siau-Cheng Khoo*   Jinyan Li†

Abstract

Sequential pattern mining first proposed by Agrawal and Srikant has received intensive research due to its wide range applicability in many real-life domains. Various improvements have been proposed which include mining a closed set of sequential patterns. Sequential patterns supported by the same sequences in the database can be considered as belonging to an equivalence class. Each equivalence class contains patterns partially-ordered by sub-sequence relationship and having the same support. Within an equivalence class, the set of maximal and minimal patterns are referred to as closed patterns and generators respectively. Generators used together with closed patterns can provide additional information which closed patterns alone are not able to provide. Also, as generators are the minimal members, they are preferable over closed patterns for model selection and classification based on the Minimum Description Length (MDL) principle. Several algorithms have been proposed for mining closed sequential patterns, but none so far for mining sequential generators. This paper fills this research gap by investigating properties of sequential generators and proposing an algorithm to efficiently mine sequential generators. The algorithm works on a three-step process of search space compaction, non-generator pruning and a final filtering step. We also introduce ranking of mined generators and propose mining of a unique generator per equivalence class. Performance study has been conducted on various synthetic and real benchmark datasets. They show that mining generators can be as fast as mining closed patterns even at low support thresholds.

1 Introduction

Sequential pattern mining first proposed by Agrawal and Srikant [1] has been the subject of active research [3, 6, 21, 28]. Given a database containing sequences, sequential pattern mining identifies sequential patterns appearing with enough support. It has been used in various fields including bioinformatics [29], customer and weblog analysis [12], security [9], and even recently in software engineering [16, 18, 19]. Addressing challenges to mining all frequent sequential patterns, namely large number of mined patterns and long running time, *closed sequential pattern mining* has been

proposed recently [27, 26]. In this paper, along a similar line of research, we investigate the concept of equivalence classes of frequent sequential patterns and develop an algorithm to efficiently mine generators, the minimal members of equivalence classes as compared to closed patterns, the maximal members of equivalence classes.

The concept of equivalence class is first introduced by Pasquier *et al.* and applied to frequent itemsets [20]. An equivalence class corresponds to the set of frequent patterns that is supported by the same set of transactions in the database. Translating to sequential patterns, an equivalence class corresponds to a set of sequential patterns supported by the same set of sequences in the sequential database. Each equivalence class contains patterns partially-ordered by sub-sequence relationship and having the same support. Within an equivalence class, closed patterns refer to those without any super-sequence in the same class (*i.e.*, the maximal ones). On the other hand, generators refer to patterns without any sub-sequence in the same class (*i.e.*, the minimal ones).

The set of closed patterns and generators are potentially of much smaller size than the full-set of frequent patterns. Yan *et al.* and Wang *et al.* have shown that closed sequential patterns can be mined much more efficiently than the full set of frequent patterns [27, 26]. As generators are minimal members of equivalence classes, according to the Minimum Description Length (MDL) principle [7], generators are the preferred descriptions or representations of the classes. Also, as argued by Li *et al*, generators are better candidates than closed patterns for their applications in model selection and classification [13].

Generators used together with closed patterns can provide additional information which closed patterns alone are not able to provide. Consider the marketing campaign design example presented in [5]. Given a database of customer purchase histories, one would like to mine and predict the behaviors of customers. Generators used with closed patterns can be used to form rules expressing minimum pre-cursor series of purchasing events that causes a maximum series of resultant purchasing events to happen. Suppose the rule is $\langle A, B \rangle \rightarrow \langle C, D, E \rangle$, where $\langle A, B \rangle$ is a generator and $\langle A, B, C, D, E \rangle$ is a closed pattern. The rule states

---
*Department of Computer Science, National University of Singapore. Email: {dlo,khoosc}@comp.nus.edu.sg.

†School of Computer Engineering, Nanyang Technological University, Singapore. Email: jyli@ntu.edu.sg.

that a customer buying A and then B is likely to buy C, D and E. A marketing manager can then send advertisements of products C, D and E to clients who have bought A and then B. Also, if products C, D and E have high profits and B is an inexpensive product, the marketing manager can promote product B (*e.g.*, by giving discounts) to customers who have bought A.

Using a closed pattern (which is longer) rather than a generator as the premise of such a customer behavioral rule might cause the marketing strategy to fail. This is the case as the identification of a customer behavior only happens latter after the longer series of purchasing events corresponding to the closed pattern occurred. Also, since the closed pattern is longer, the premise might tend to 'overfit' existing data and potentially has less power in predicting future customer behaviors (see also discussion in Section 5).

In frequent itemset mining, Li *et al.* have introduced efficient mining of itemset generators [13, 14]. Unfortunately, in sequential pattern mining, there has not been any research done in mining sequential generators. Our attempt to reuse the techniques devised for mining generators of frequent itemsets has failed due to inherent differences exhibited between sequential pattern mining and itemset mining. We elaborate these differences below.

First, the number of closed sequential patterns found in an equivalence class might be more than one. This is in contrast with the itemset mining where there is only a single closed itemset for every equivalence class. Table 1 provides a comparison on the singularity of closed itemsets/patterns and generators in an equivalence class for itemsets and sequential patterns.

Possible	In an Equivalence Class of	
Number of	Sequential Patterns	Itemsets
Closed patterns	Many	Single
Generators	Many	Many

Table 1: Singularity of Closed Itemsets/Patterns and Generators within an Equivalence Class

Also, more importantly, itemset generators exhibit a nice *a-priori* property stating that every subset of a generator must also be a generator. Employing this property, effective search space pruning strategy can be deployed to significantly reduces runtime. As we shall see, sequential generators do *not* possess this property. Thus, a new scheme for mining generators need to be devised.

This paper fills this research gap by investigating properties of sequential generators, and by proposing a method to mine these generators efficiently. Also, since an equivalence class might have more than one closed pattern, the number of equivalence classes is less than or equal to the number of closed patterns (similarly for generators). Consequently, we propose a technique for fast identification of equivalence classes of sequential patterns. Furthermore, we propose a method to rank generators within an equivalence class and to efficiently mine the one with the highest rank from each class.

Our algorithm works on a three-step process of search space compaction, non-generator pruning and a final filtering step. After the non-generator pruning step, our algorithm produces a set of candidate generators which is a super-set of all generators. This set is later subjected to a final filtering step to reduce it to a set of generators. Our generation process is driven by the application of a novel *apriori* property of *non*-generators. This property enables effective pruning of sub-search spaces of *non*-generators.

In this paper, we first present an algorithm to mine a full-set of generators. We then extend it to identify generators belonging to the same equivalence class, rank generators in the same class and present one generator with the highest rank from each class.

A performance study has been conducted on various simulated and real benchmark datasets. The study shows that mining generators greatly reduces the time needed to mine all frequent sequential patterns. It also shows that the runtime of our algorithm can be on par with, and at times faster than, that of mining closed patterns using state-of-the-art closed pattern miners.

The contributions of this work are as follows:

1. We explore and present *novel properties* of sequential generators. Also, we propose a novel algorithm to efficiently mine *all sequential generators* from a sequence database.

2. We are the first to investigate the concept of equivalence class in sequential pattern mining. Based on the concept, we develop an efficient method to *identify generators belonging to an equivalence class*, *rank* generators within an equivalence class, and for each equivalence class, *mine the one with the highest rank*.

3. We show, by our performance study, *new interesting results and observations* on the *efficiency* of closed sequential pattern mining algorithms in comparison with our generator mining algorithms on various datasets.

The structure of this paper is as follows. Section 2 discusses related work. Section 3 describes terminologies used and highlights differences between generators and closed patterns. Section 4 presents novel properties and characteristics of generators. Section 5 outlines a description on why MDL favors generators. Sec-

tion 6 describes our mining algorithm. Section 7 discusses ranking of generators and efficient mining of the highest rank generator per equivalence class. Section 8 presents the performance study results on several benchmark datasets. Finally, Section 9 summarizes and concludes our work.

2 Related Work

Two related research threads are mining itemset generator (*e.g.*, [20, 13, 14]) and sequential pattern mining (*e.g.*, [1, 27, 26, 25]). In this section, we will compare the above studies with our work and discuss their differences.

Pasquier *et al.* first introduces the concept of equivalence classes to characterize the set of mined frequent itemsets [20]. Li *et al.* improves the algorithm in [20], by employing a depth-first mining strategy rather than a breadth-first strategy previously employed in [20]. In [14], Li *et al.* extends their work further by concurrently obtaining both closed itemsets & generators and computing delta discriminative non-redundant equivalent classes.

Similar to the above work, we investigate equivalence classes of frequent patterns and develop an algorithm to efficiently mine generators. However, we focus on sequential patterns, rather than itemsets, mined from a sequence database. In sequential pattern mining, ordering of events is important. Much real-life data is in sequence format, *e.g.*, purchase history, employment history, protein sequence, DNA, program traces, etc.

Agrawal and Srikant first proposes sequential pattern mining in [1]. The above work has been improved by others *e.g.*, [3, 21, 28, 27, 26]. In [3, 21, 28], a full set of frequent sequential patterns is mined. However, mining a full-set of frequent sequential patterns is often not feasible, as the number of patterns mined can be too large and correspondingly the runtime can be too long. To address these problems Yan *et al.* present an algorithm named *CloSpan* to mine a closed set of sequential patterns [27]. In [26], Wang *et al.* proposed another algorithm called *BIDE* which has been shown to be more efficient than *CloSpan* on a click stream and two biological datasets.

Similar to mining frequent itemsets, equivalence classes can characterize the set of mined sequential patterns. Closed patterns correspond to the members of equivalence classes each having no super-sequence in its class. Generators corresponds to those members having no sub-sequence.

Our work is the first on mining sequential generators. According to the Minimum Description Length (MDL) principle [7], generators are the preferred representation of a class. Also, as argued in [13], generators are better candidates than closed patterns for their applications in model selection and classification (see also Section 5).

An equivalence class of frequent sequential patterns can contain more than one closed patterns and generators. Hence, the number of equivalence classes is less than or equal to the number of closed patterns and generators. This is the first work that identifies equivalence classes of sequential patterns by identifying generators belonging to the same class, ranking them and extracting the highest rank generator for each class.

Rather than re-inventing the wheel, we borrow the concept of *projected database* first introduced in [21] and *prefix sequence lattice* in [27]. In our final filtering step, we also adapt the *fast subsumption checking* mechanism first introduced in [28]. However there are a number of significant differences. *Output-wise* we produce a set of sequential generators rather than a full set of frequent sequential patterns, a set of closed sequential patterns, or a set of closed itemsets produced by techniques in [21], [27], [28] respectively. *Data structure-wise*, different from the prefix sequence lattice (PSL) introduced in [27], we need to attach labels to PSL's edges and manage these labels in order to aid pruning of non-generators. *Algorithm-wise*, we present a set of unique properties of sequential generators and employ new pruning strategies. *Additionally*, we also introduce new data structure and methods to identify equivalence classes, rank generators of each class and mine the generator with the highest rank from each class efficiently.

3 Preliminaries

In this section, we describe preliminaries on: notations used, basic property of sequential patterns, basic operations performed on a sequential database and the concept of equivalence class.

Let I be a set of distinct items. Let a *sequence* S be an ordered list of events. We denote S by $\langle \mathtt{e_1}, \mathtt{e_2}, \ldots, \mathtt{e_{end}} \rangle$ where each $\mathtt{e_i}$ is an item from I. A pattern P_1 ($\langle \mathtt{e_1}, \mathtt{e_2}, \ldots, \mathtt{e_n} \rangle$) is considered a *subsequence* of another pattern P_2 ($\langle \mathtt{f_1}, \mathtt{f_2}, \ldots, \mathtt{f_m} \rangle$), denoted by $P_1 \sqsubseteq P_2$, if there exist integers $1 \leq \mathtt{i_1} < \mathtt{i_2} < \mathtt{i_3} < \mathtt{i_4} \ldots < \mathtt{i_n} \leq \mathtt{m}$ such that $\mathtt{e_1} = \mathtt{f_{i_1}}, \mathtt{e_2} = \mathtt{f_{i_2}}, \cdots, \mathtt{e_n} = \mathtt{f_{i_n}}$. We also say that P_2 is a *super-sequence* of P_1. The sequence database under consideration is denoted by $SeqDB$. The length of P is denoted by $|P|$. An empty pattern has length 0 and is written as $\langle \rangle$. A pattern $P_1 \!+\!\!+ P_2$ denotes the concatenation of pattern P_1 and pattern P_2.

The *absolute support* of a pattern with respect to a sequence database $SeqDB$ is the number of sequences in $SeqDB$ that are super-sequences of the pattern. The *relative support* of a pattern *wrt* $SeqDB$ is the ratio of

its absolute support to the total number of sequences in $SeqDB$. The support (either absolute or relative) of a pattern P in a sequence database $SeqDB$ is denoted by $sup(P, SeqDB)$. We ignore the mentioning of the database when it is clear from the context (*i.e.*, denoted as $sup(P)$).

From the definitions of "support" and "frequent pattern", sequential patterns possess *'apriori' property* [8]: *If a sequential pattern is frequent then all its subsequences are also frequent.* In other words, support of a pattern is greater or equal to support of its super-sequences.

A pattern P is *frequent* when its support, $sup(P)$, exceeds a certain threshold (min_sup). A frequent pattern P is *closed* if there exists no super-sequence of P having the same support as P. A frequent pattern P is a *generator* if there exists no sub-sequence of P having the same support as P.

Similar to the work by Wang and Han in [26], we only consider *single-item* sequences (*i.e.*, all transactions in a sequence is of size 1). This simplifies our presentation. Furthermore, single-item sequences also represent many important types of sequences (*eg.* DNA sequences, protein sequences, web click streams, program API traces, *etc.*).

Patterns supported by the same set of sequences in the database can be viewed as belonging to an equivalence class. Mathematically, the problem can be described as follows. Consider a dataset being described as $Data = \langle E, SDB \rangle$, where E is a set of events and $SDB \subseteq E^*$ is a sequence database. For a pattern P, we define a function $f(P) = \{S \in SDB | P \sqsubseteq S\}$. Two patterns P_1 and P_2 belong to the same equivalence class iff $f(P_1) = f(P_2)$. Given an equivalence class EQ, the set $\{P \in EQ | \neg \exists P_S \in EQ.(P \neq P_S \wedge P_S \sqsupseteq P)\}$ is the set of closed patterns in EQ. Similarly, the set $\{P \in EQ | \neg \exists P_S \in EQ.(P \neq P_S \wedge P_S \sqsubseteq P)\}$) is the set of generators in EQ.

To illustrate the concepts of equivalence class, generator and closed pattern consider the following database shown in the following table.

Seq ID.	Sequence	Seq ID.	Sequence
$S1$	$\langle A, D, A \rangle$	$S2$	$\langle B, A, D, A \rangle$
$S3$	$\langle A, B, C, B \rangle$	$S4$	$\langle A, B, B, C \rangle$
$S5$	$\langle B, A, B \rangle$	$S6$	$\langle D, X, Y \rangle$

Considering min_sup set at 2, the frequent pattern space corresponds to the following lattice in Figure 1. There are 16 frequent patterns including the empty pattern $\langle \rangle$ which is trivially frequent.

Among the 16 frequent patterns, there are 8 equivalence classes marked by the dotted lines and referred to as EQ_1–EQ_8 in Figure 1. In each equivalence classes, generally those patterns at the bottom are the closed
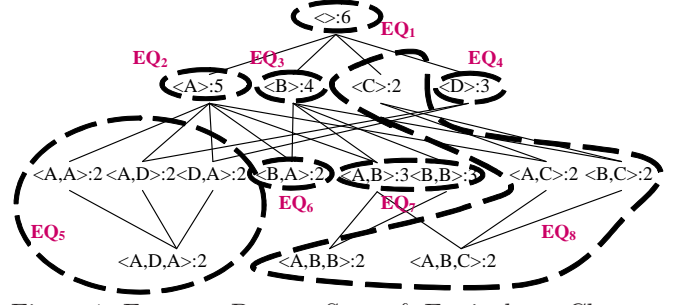

Figure 1: Frequent Pattern Space & Equivalence Classes

patterns while those at the top are the generators. For example, consider the equivalence class EQ_5 which is supported by $S1$ and $S2$ in the database. The closed pattern of EQ_5 is $\langle A, D, A \rangle$, while the set of generators is $\{\langle A, A \rangle, \langle A, D \rangle, \langle D, A \rangle\}$. Also consider EQ_8 which is supported by $S3$ and $S4$ in the database. The set of closed patterns of EQ_8 is $\{\langle A, B, B \rangle, \langle A, B, C \rangle\}$. The set of generators of EQ_8 is the set $\{\langle A, B, B \rangle, \langle C \rangle\}$. Note that an equivalence class can have more than one closed pattern and more than one generator.

Another important concept is the notion of projected database [21, 27]:

DEFINITION 1. (**Projected Database**) *A sequence database* projected *on a pattern p is defined as a multi-set as follows:*
$$SeqDB_P = \{ sx | s \in SeqDB, s = px{+}{+}sx,$$
$$px \text{ is the minimum prefix of } s \text{ containing } p\}.$$

To illustrate, let's refer to the last example sequence database as $SeqDB$. The projected database $SeqDB_{\langle A,D \rangle}$ is $\{\langle A \rangle, \langle A \rangle\}$. Also, the projected database $SeqDB_{\langle A,B \rangle}$ is $\{\langle C, B \rangle, \langle B, C \rangle, \langle \rangle\}$.

4 Unique Characteristics of Sequential Generators

Itemset generators exhibit an elegant property that enables its efficient mining. This is termed as the *apriori* property of itemset generators [13]. It is defined as follows:

PROPERTY 1. (**Apriori P. of Itemset Generators**) *If an itemset is a generator, then all its subsets are also generators.*

One might be tempted to adapt this property to mining of generators for sequential patterns. That is, *"If a sequential pattern is a generator, then all its subsequences are also generators"*. Unfortunately, this property does not hold for sequential generators. Consider the following counter-example:

Seq ID.	Sequence	Seq ID.	Sequence
S1	$\langle A, B, C \rangle$	S2	$\langle A, C, B \rangle$
S3	$\langle A, B, C \rangle$	S4	$\langle B, C \rangle$

The set of sequential generators, with min_sup set as 2, is $\{\langle\rangle : 4, \langle \mathtt{A}\rangle : 3, \langle \mathtt{B,C}\rangle : 3, \langle \mathtt{A,B,C}\rangle : 2\}$. Note that although $\langle \mathtt{A,B,C}\rangle$ is a generator (of support 2), some of its subsequences, such as $\langle \mathtt{A,B}\rangle$, $\langle \mathtt{B}\rangle$ and $\langle \mathtt{C}\rangle$, are not generators. The pattern $\langle \mathtt{A,B}\rangle$ is not a generator since it is in the same class as its subsequence $\langle \mathtt{A}\rangle$. The patterns $\langle \mathtt{B}\rangle$ and $\langle \mathtt{C}\rangle$ are not generators since they are in the same class as the trivial generator $\langle\rangle$.

Non-generators of sequential patterns, on the other hand, possess the following apriori property which can help to speed up the mining of sequential generators.

PROPERTY 2. (**Apriori P. of Non-Generators**)
Given a sequential pattern P_1, if $\exists$ another pattern P_2 such that $SeqDB_{P_1} = SeqDB_{P_2}$ and $P_1 \sqsupset P_2$ then P_1 and any extension of it (i.e., $P_1{+}{+}P_X$ where P_X is a sequence of events) are not generators.

Proof. That P_1 is not a generator is obvious from the premises. Next, since P_1 and P_2 has the same projected database, if we can extend pattern P_1 (resp. P_2) by a series of events P_X to form a subsequence of a sequence in the database, we can always extend pattern P_2 (resp. P_1) by the same P_X to form a subsequence of that sequence. Thus, for any series of events P_X, the support of $P_1{+}{+}P_X$ will always be the same as $P_2{+}{+}P_X$. Hence, all extensions of P_1 (*i.e.*, $P_1{+}{+}P_X$) are not generators.□

One might think that the number of generators in an equivalence class must be more than the number of closed patterns. This is true for frequent itemsets: There is only one closed itemset for each equivalence class. For frequent sequential patterns, however, an equivalence class can have more than one closed pattern. In fact, the following property is observed.

PROPERTY 3. (|*Closed*| vs. |*Generators*|) *There exists sequential databases where the number of closed patterns is more than the number of generators.*

Proof. An example of such a database is as follows:

Seq ID.	Sequence	Seq ID.	Sequence
S1	$\langle \mathtt{D,C,E,E,D,C}\rangle$	S2	$\langle \mathtt{E,C,D,D,C,E}\rangle$
S3	$\langle \mathtt{X}\rangle$		

By setting min_sup to 2, the set of closed patterns is: $\{\langle\rangle{:}3, \langle \mathtt{C,D,C}\rangle{:}2, \langle \mathtt{D,C,E}\rangle{:}2, \langle \mathtt{D,D,C}\rangle{:}2, \langle \mathtt{E,D,C}\rangle{:}2, \langle \mathtt{E,E}\rangle{:}2\}$. The set of generators is: $\{\langle\rangle{:}3, \langle D\rangle{:}2, \langle C\rangle{:}2, \langle E\rangle{:}2\}$. Thus, the number of closed patterns is more than that of generators. Other databases observing Property 3 can be similarly constructed. □

5   MDL Favors Generators

Generators can be more useful than closed patterns in some applications, especially those where the principle of Minimum Description Length (MDL) applies.

Rissanen first proposed the notion of Minimum Description Length (MDL) [24]. A recent book [7] describes this subject. This principle has been widely used for model selection (*e.g.*, [22, 23]). It is founded on well-established concepts of Bayesian-inference and Kolmogorov complexity (*c.f.*, [15]). In [13], Li *et al.* propose an MDL formula for frequent itemsets and argue for the benefit of itemset generators over closed itemsets. We adapt their formulation and argument on the benefits of generators and apply them to sequential patterns.

Consider a set of hypothesis, $\mathcal{H} = \{H_1, H_2, ..., H_n\}$ describing a piece of data D. The MDL principle dictates that the best hypothesis is one with minimum length (the simplest one – Occam's Razor). The description of a piece of data usually comprises of two elements, namely the model and the description of the data given the model. Hence, the best hypothesis $H \in \mathcal{H}$ describing the data D, is the one that minimizes the description length formula, $L(H, D) = L(H) + L(D|H)$, where:

- $L(H)$ is the description length (in bits) of the hypothesis H.
- $L(D|H)$ is the description length (in bits) of the data D encoded with prior knowledge of the hypothesis H.

Following the description in [13], closed patterns and generators can be considered as two hypotheses describing the sequence datasets. Consider an equivalence class of patterns EQ supported by a set of sequences $SSet$. For a closed pattern $C \in EQ$ describing $SSet$, the Minimum Description Length formula is $L(C, SSet) = L(C) + L(SSet|C)$. For a generator $G \in EQ$ describing $SSet$, the formula is $L(G, SSet) = L(G) + L(SSet|G)$. Since, both C and G occur in the same data $SSet$, $L(SSet|C) = L(SSet|G)$. Since length of C is often longer than G, $L(C)$ is often larger than $L(G)$. Hence, $L(C, SSet) > L(G, SSet)$ in most cases. According to MDL principle, the generator G is hence preferred over the closed pattern C.

This preference is particularly clear in considering classification problem using a training set and an independent test set. Learning classification rules from closed patterns mined from the training set results in more specific rules than ones from generators. Closed patterns tend to be longer and 'overfit' the training data. Generators are shorter and more tolerant to noise present in the training data. Adapting the example given in [13], consider an application where only two classes of sequences are involved: *positive* and *negative*. Assume all sequences corresponding to the equivalence class of a closed pattern C ($\langle e_1, e_2, \ldots, e_n\rangle$ ($n{>}2$)) and a generator G ($\langle e_1, e_2\rangle$) belong to the *positive* class. Then

the following two rules can be obtained from C and G respectively:

- If a sequence is a super-sequence of ev_1 followed by ev_2, followed by ev_3, ..., followed by ev_n, then it is positive.
- If a sequence is a super-sequence of ev_1 followed by ev_2, then it is positive.

Both rules are satisfied by the same set of sequences in the training set. The rule derived from G tends to be more predictive (as an unknown test data will more likely be a super-sequence of a sequence of two events rather than n events), generalize better, not 'overfit' the training data and more tolerant to noise.

6 Mining Sequential Generators

We mine sequential generators in a three-step compact-generate-and-filter approach. The first step traverses the search space of frequent patterns and produces a compact representation of the space of *frequent* patterns in a lattice format. This lattice is referred to as *prefix search lattice* (PSL). This data structure is an extension of the PSL previously described by Yan *et al.* in [27]. The second step retrieves a set of candidate generators which is a super-set of all generators from the compact lattice. Two pruning strategies derived from Property 2 are applied to effectively prune the sub-search spaces containing non-generators and help to ensure that the candidate generator set is not too large. In the final step, non-generators from this candidate set are filtered away, leaving behind a set of generators. Let us use the sequence database shown in the following table as a running example to illustrate the various operations described in this section.

Seq ID.	Sequence	Seq ID.	Sequence
S1	$\langle A, C, A, A \rangle$	S2	$\langle A, B, B \rangle$
S3	$\langle A, B, C, A, B \rangle$		

6.1 Prefix Search Lattice. Before describing *prefix search lattice* (PSL), we will first introduce *prefix search tree* (PST). A PSL is a compacted form of a PST. Both PSL and PST represent the space containing all frequent sequential patterns. By traversing PST or PSL, one will be able to recover the full set of frequent patterns with their corresponding support.

Each node in a PST is labeled with: a corresponding event and the support of the pattern formed by traversing the tree from its root to this node. Given a node N, we denote the support and event of the node by $N.Sup$ and $N.Ev$ respectively. Given an edge E connecting two nodes, we denote its source node by $E.Source$ and sink node by $E.Sink$. An example of a PST is shown in Figure 2(a).

A PST represents a *set of patterns*. A path in a PST is a sequence of edges from the root to a non-root node. A pattern then corresponds to a path in the tree. Consider a path PTH $(E_1, E_2, \ldots, E_X)$. This path *corresponds to a pattern* $P_X \langle E_1.Sink.Ev, \ldots, E_X.Sink.Ev \rangle$. The support of this pattern is given by the support attached at the end node (*i.e.*, $\sup(P_X) = E_X.Sink.Sup$). A pattern P_X is in the PST iff there is a path in the PST corresponding to P_X.

PSL is meant to avoid space explosion by identifying and grouping common subtrees occurring in a PST. The corresponding PSL of the PST shown in Figure 2(a) is drawn in Figure 2(b). As is shown later in subsection 6.2, it is not necessary to build the PST first before compacting it to a PSL. In fact, a PSL can be built more efficiently than a PST.

6.2 PSL Building Step. This step begins by adding events as nodes to a tree in a fixed order (*i.e.*, lexicographic order) and in depth-first fashion to form a PST. In order to prevent a space explosion in PST building, a compact representation of PST is built by merging common subtrees. The resultant representation is a prefix sequence lattice (PSL). Identification of common subtrees is based on the following theorem:

THEOREM 6.1. (**Equivalent Projected Database**)
Consider two patterns P_1 and P_2, where $P_1 \sqsubset P_2$ and $SeqDB_{P_1} = SeqDB_{P_2}$. Let N_1 and N_2 be the respective end nodes of the paths corresponding to P_1 and P_2 in PSL, then any sub-tree extending from N_1 and N_2 will be equivalent.

Proof. The proof is similar to that of Property 2. □

Conceptually, merging of common subtrees is described in Figure 3.
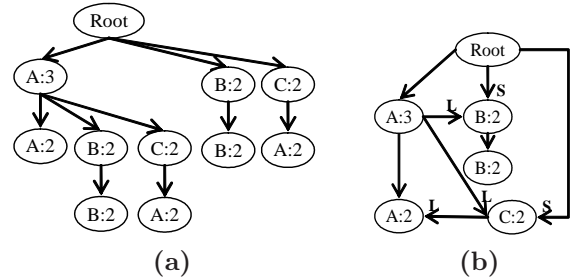

(a)　　　　　　　　　(b)
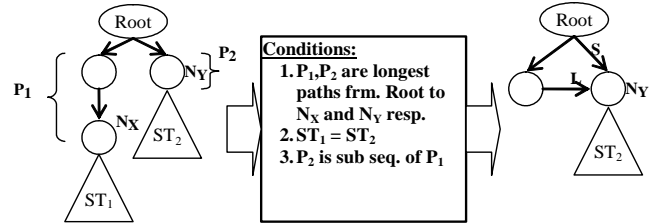
Figure 2: A sample PST and PSL

Figure 3: PST to PSL

Merging is slightly complicated by the need to label edges for pruning performed in the next step of the mining algorithm. Specifically, labels are used to identify some of the non-generator patterns in order to facilitate pruning. To this end, an edge in PSL is labeled with either 'U', 'L' or 'S', as explained below.

DEFINITION 2. (EDGE LABELS – 'U', 'L' AND 'S')
An edge E_L sinking in a node N is labeled 'L' when there is a path from root ending with E_L which is a super-sequence of another path from root ending in N. Similarly, an edge E_S sinking in a tree node N is labeled 'S' when there is a path from root ending with E_S which is a sub-sequence of another path from root ending in N. The label 'L' takes precedence over 'S', meaning that an edge that can both be labeled 'L' and 'S', is labeled 'L'. All other edges are labeled 'U'.

We write $E.Label$ to denote the label associated with edge E. Diagrammatically, we omit the labeling of 'U' for clarity.

DEFINITION 3. $(Pat(PTH), L\text{-}Pat(N_X))$ *Given a path or a sequence of edges PTH, $Pat(PTH)$ returns the pattern corresponding to PTH. Given a node N_X, $L\text{-}Pat(N_X)$ returns the longest path from root to N_X.*

During construction of a PSL, every time a leaf node N_P is extended with a new event ev_{new} represented by the node N_{new}, a new pattern P_{new} corresponding to a unique path from root to N_{new} is added to the PSL. This path is unique because the PSL is constructed depth-first. A check will be made at this moment for the following two cases [1]:

1. $\exists$ a node $N_O \in PSL$ and a pattern $P_O = L\text{-}Pat(N_O).(P_O \sqsupset P_{new} \wedge SeqDB_{P_O} = SeqDB_{P_{new}})$.
2. $\exists$ a node $N_O \in PSL$ and a pattern $P_O = L\text{-}Pat(N_O).(P_O \sqsubset P_{new} \wedge SeqDB_{P_O} = SeqDB_{P_{new}})$.

If either of the cases occurs, the sub-tree rooted at N_O will be identical to the sub-tree rooted at N_{new}. We can avoid re-traversing the search space by dropping N_{new} and adding a new edge E_{New} from N_P to N_O. If case 1 occurs, E_{New} will be labeled as 'S', otherwise if case 2 occurs, it will be labeled as 'L'. Hence using Theorem 6.1, it is not necessary to build the PST first before compacting it to a PSL. This is the case as common sub-trees can be identified early by checking for the equivalence of projected databases even before one of the sub-trees is generated.

Appropriate edge re-labeling will also be made at this time. Two cases require consideration:

[1]The check involves detection of equivalence of projected databases. Fast detection technique has been proposed in [27] and is utilized in our algorithm.

1. An edge E_X labeled as 'S' and sinking at node N_O will be re-labeled as 'L' if the pattern $L\text{-}Pat(N_O)$ is a super sequence of P_{new}.
2. An edge E_X labeled as 'U' will be re-labeled as 'L' or 'S' if $L\text{-}Pat(N_O)$ is a super-sequence or sub-sequence of P_{new} respectively.

Based on the labels, the following property of the PSL can then be inferred. The property identifies some non-generator occurring in the PSL.

PROPERTY 4. (PATH PROPERTY) *Consider an arbitrary path $PTH_X(E_0, E_1 \ldots E_X)$ composed of edges traversing nodes: root, ..., N_{X-1} and N_X, such that $\forall i < X$, $E_i.Label = 'U'$ and $E_X.Label = 'L'$. There always exists another path PTH_Y from root to N_X where $Pat(PTH_Y) \sqsubset Pat(PTH_X)$ and $SeqDB_{Pat(PTH_Y)} = SeqDB_{Pat(PTH_X)}$.*

Proof. Consider PTH_{X-1} $(E_0, E_1 \ldots E_{X-1})$ ending in N_{X-1}. PTH_{X-1} contains only 'U'-labeled edges and hence it must be the unique path from root to N_{X-1}. Extending this argument, the path PTH_X must then be the *unique path* from root to N_X ending at edge E_X.

Since PTH_X is unique and E_X is labeled as 'L', from Definition 2 the following must hold: $\exists$ $PTH_Y(E_{Y_0}, \ldots, E_{Y_I})$ where $E_{Y_0}.Source = root$ and $E_{Y_I}.Sink = N_X$ and $Pat(PTH_Y) \sqsubset Pat(PTH_X)$. Also since both PTH_X and PTH_Y end at the same tree node N_X, $Pat(PTH_X)$ and $Pat(PTH_Y)$ share the same projected database. Hence, the above property is proven. □

In this step, a compact representation of the space of frequent patterns in the form of PSL is produced. Several strategies are employed to avoid redundant exploration of the search space. Labels are also computed to aid pruning performed in the next step of the algorithm. The PSL of our running example is shown in Figure 2(b).

6.3 S-Gen Generation Step. Next, we mine a set of candidate generators which is a super-set of all generators (referred to as S-Gen) from the PSL.

To generate patterns from PSL, one can simply traverse the tree from root to leaves and generate patterns on the way. The resultant pattern set will be a *full-set of sequential patterns* which can be very large. The novelty of this step is the identification and pruning of sub-search spaces containing non-generators. This is based on the following two pruning lemmas.

LEMMA 6.1. (**Labeled Path Pruning**) *Given a traversal from root to N_L following path PTH $(E_1, \ldots, E_{L-1}, E_L)$, where E_i $(1 \leq i \leq L)$ is a traversed edge, if $E_L.Label = 'L'$, we can omit generating patterns corresponding to the extensions of PTH if $(\neg \exists$ an edge $E \in \{E_1, \ldots, E_{L-1}\}$ where $E.Label \neq 'U')$.*

Proof. The edge E_L is the first non-'U' labeled edge in path PTH. It is labeled by 'L'. From Property 4, there exists another path PTH_Y where $Pat(PTH) \sqsupset Pat(PTH_Y)$ and $SeqDB_{Pat(PTH)} = SeqDB_{Pat(PTH_Y)}$. From Property 2, $Pat(PTH)$ and its extensions are not generators. Hence, there is no need to generate patterns by extending PTH.$\square$

LEMMA 6.2. (**C-Steps Look-Backward Pruning**)

Consider a traversal from root to N_L following a path PTH $(E_1, \ldots, E_{L-C+1}, ..., E_{L-1}, E_L)$. The set of C-Look backward events (CLB) are the corresponding last C events in path PTH. If there exists another path PTH_Y where $Pat(PTH_Y) = P_1{+}{+}P_2$ such that the following holds:

1. $P_1{+}{+}ev{+}{+}P_2 = Pat(PTH)$ and $ev \in CLB$

2. $SeqDB_{Pat(PTH_Y)} = SeqDB_{Pat(PTH)}$

Then we can omit generation of patterns corresponding to the extension of PTH.

Proof. From the lemma, $Pat(PTH) \sqsupset Pat(PTH_Y)$ and $SeqDB_{Pat(PTH_Y)} = SeqDB_{Pat(PTH)}$. From Theorem 2, $Pat(PTH)$ and its extensions are not generators. Hence, we can stop generating patterns by extending PTH.$\square$

Lemma 6.2 can be checked on-the-fly during the PSL traversal. In our experiments, the parameter C of Lemma 6.2 is set to 2.

From the two pruning lemmas, we can generalize the generation of S-Gen by Theorem 6.2.

THEOREM 6.2. (**S-Gen Generation**) *A super-set of all generators (referred to as S-Gen) can be generated by traversing the PSL from root until either a leaf node is reached or one or both of Lemmas 6.1 & 6.2 hold.*

The set of S-Gen patterns generated from the running example is $\{\langle A \rangle : 3, \langle A, A \rangle : 2, \langle B \rangle : 2, \langle B, B \rangle : 2, \langle C \rangle : 2, \langle C, A \rangle : 2\}$.

6.4 Final Filtering Step. The result of the steps outlined in previous sub-sections is a set of candidate generators, which is a super-set of all generators (*i.e.*, Gen-S). Next, we need to eliminate members of the candidate set that are not generators. To check this we need to find each pair of patterns P_1 and P_2 in Gen-S such that $P_1 \sqsubset P_2$ and $sup(P_1) = sup(P_2)$. Here, P_2 is not a generator and should be removed.[2]

The set of generators generated from our running example is $\{\langle A \rangle : 3, \langle A, A \rangle : 2, \langle B \rangle : 2, \langle C \rangle : 2\}$.

Our algorithm, called GenMiner, performs the PSL building, S-Gen generation and final filtering steps. It is shown in Figure 4.

---

[2]A fast checking of sub-sequence relationships among patterns having the same support can be performed via hashing. We adapt the fast subsumption checking algorithm first introduced in [30] and adapted in [27].

Procedure GenMiner
Inputs: *SeqDB*: A sequence database;
          min_sup: Minimum support threshold;
Outputs: *Gen*: The set of mined generators;
Method:
1: Let $FreqEvs = \{<ev, s> \mid (sup(ev, SeqDB) = s)$
                    $\wedge (s \geq min\_sup)\}$
2: Let $root$ = Create new root node
3: Let $PSL = root$
4: For each pattern $<ev, s> \in FreqEvs$
5:    Let N_{new} = Create new node (ev, s)
6:    Append N_{new} as child of $root$
7:    Call ExtendGen $(SeqDB_{ev}, N_{new}, PSL, min\_sup)$
8: Let $Cand$ = Generate Gen-S acc. to. Theorem 6.2.
9: Set Gen = Perform final filtering step to $Cand$
10: **Output** Gen

Procedure ExtendGen
Inputs: *PDB*: A projected database;
          ParentNode: Current node in *PSL*;
          PSL: Prefix Sequence Lattice
          min_sup: Minimum support threshold;
Method:
1: Let $FrNxEvs = \{<ev, s> \mid (sup(ev, PDB) = s)$
                    $\wedge (s \geq min\_sup)\}$
2: For each pattern $<ev, s> \in FrNxEvs$
3:    Let N_{new} = Create new node (ev, s)
4:    Append N_{new} as child of $ParentNode$
5:    Let P_{new} = A new pattern corresponding to the
                unique path from the PSL's root
                to N_{new}
6:    If $\exists$ node $N_O \in PSL \wedge$ pattern $P_O = L\text{-}Pat(N_O).$
           $(P_O \sqsupset P_{new} \wedge SeqDB_{P_O} = SeqDB_{P_{new}})$
7:       Update PSL (see sub-section 6.2)
8:    Else If
         $\exists$ node $N_O \in PSL \wedge$ pattern $P_O = L\text{-}Pat(N_O).$
           $(P_O \sqsubset P_{new} \wedge SeqDB_{P_O} = SeqDB_{P_{new}})$
9:       Update PSL (see sub-section 6.2)
10:   Else Call ExtendGen $(PDB_{ev}, N_{new}, PSL, min\_sup)$

Figure 4: Main Algorithm – GenMiner

7   Ranking Generators

The algorithm described in Section 6 mines a full set of generators. In this section, we will describe how we can relate together generators belonging to the same equivalence class efficiently. Also, an approach to rank generators belonging to the same equivalence class will be described. Finally, we will describe the approach to mine a representative generator with the highest rank from each equivalent class efficiently.

7.1   Identifying Equivalence Classes. Before one can rank generators belonging to an equivalence class, one need to first group generators according to its class. In mining frequent itemsets, as demonstrated in [14] this can be done by growing a generator to its corresponding closed itemset. As a closed itemset of an equivalence

class is unique, it can serve as a unique identifier and can then be used to identify the class an itemset belongs to. However, this is no longer possible in mining sequential patterns. The number of closed patterns per equivalence class can be more than one. There can even be more closed patterns than generators in a class as shown in Section 3.

By definition, an equivalence class is a set of patterns supported by the same set of sequences in the database. Hence, a unique identifier for each equivalence class can be the list of its supporting sequences. A naive approach is to keep a set of links from *every candidate generators* to its supporting sequences. However, this is not feasible as the number of links is prohibitive. This is the case if the set of candidates are many and for each its support is high.

To address the above problem, we propose the following data structure and approach that will reduce the required memory and runtime required in identifying the equivalence classes. The data structure supporting the process is shown in Figure 5.
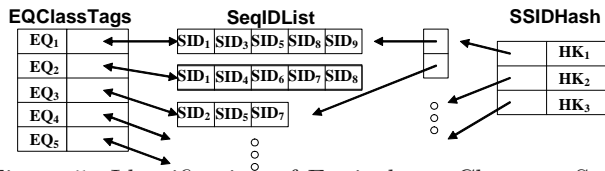
Figure 5: Identification of Equivalence Classes – Supporting Data Structure

Rather than attaching a set of links to a candidate generator, we attach a unique identifier tag corresponding to its equivalence class to each candidate. We cannot assign a tag for each possible equivalence class statically since the number of possible generators is the power-set of the number of sequences in the database. Rather every time a new equivalence class is identified, we assign an unassigned tag to it. For each tag, we then keep a set of links to its supporting sequences. These unique equivalence class tags are kept in a `EQClassTags` data structure shown in Figure 5. Each member of the `EQClassTags` has a pointer to a `SeqIDList` data structure which is a list of identifiers of sequences supporting the corresponding equivalence class.

To identify future candidates belonging to the same equivalence class, the naive approach will be to compare a candidate list of identifiers of supporting sequences to every other existing `SeqIDList` data structures. However, this approach incurs high computation cost.

Clearly, since patterns in the same equivalence class are supported by the same sequences in the database, the sum of identifiers of their supporting sequences should be the same. We use this sum of identifiers as a hash key and these keys are kept in another data

structure `SSIDHash`. Each bucket corresponding to a hash key contains pointer(s) to one or more `SeqIDList` data structures identifying one or more equivalence class(es). The *sparse distributivity* of sums of identifiers of supporting sequences and the feasibility of using the sums as hash keys have been investigated by Zaki and Yan *et al.* [28, 27] (see the footnote at sub-section 6.4). With a well distributed key, hashing can significantly reduce the computation cost in computing and assigning tags to candidate generators.

Whenever a new candidate generator G is identified the following steps are performed:

1. Compute the sum of identifiers of sequences supporting the new candidate G. This value will serve as G's hash key.

2. Access `SSIDHash` to find a list of possible equivalence classes G belongs.

3. If an equivalence class is found, attach the unique equivalence class tag to it. Otherwise, create new entries in `SSIDHash` and `EQClassTags` accordingly.

With the above strategy we can identify generators belonging to the same equivalence class. The memory consumption will not be excessive as we only keep one `SeqIDList` data structure for every equivalence class. Also, due to `SSIDHash` data structure only a minimal amount of comparisons are needed to identify the identifier of a new candidate generator.

7.2 Ranking Criteria. After identifying generators belonging to the same class, we can then rank them according to some criteria. By ranking generators, we can distinguish generators belonging to the same equivalence class. Consequently, it aids selection of the best representative generator to be mined, one for every class (see sub-section 7.3). The size of the resultant representative generator set will then be equal to the number of equivalence classes. We propose three different ranking criteria as follows:

1. Pattern Length. According to the MDL principle, patterns with the shortest description are preferred. Hence, we rank the generators according to their length in ascending order.

2. Maximum Jump. A generator in an equivalence class must have its subsequences belonging to another class. We define the maximum jump of a generator to be the maximum ratio between the support of the generator to that of one of its subsequences. Generators in an equivalence class with the maximum jump are the ones which are the most different from its immediate sub-sequence patterns and hence are more interesting than the other generators. We can then rank the generators according to their maximum jumps.

3. Average Jump. This is a variant of the second criteria. Rather than considering the maximum jump of a generator we average its jumps. Generators are then ranked based on their average jump.

In our study, we consider the above three criteria in a pipeline. We first sort the generators based on their length, if there is a tie, we then rank those in the tie cases based on their maximum jumps, if a tie occurs again, we finally rank those in the tie cases based on their average jumps. Note that at the end of the pipeline there might be a tie. To resolve the tie, we further sort the generators based on lexicographic order with respect to the alphabet set considered.

7.3 Mining One Generator per Equivalence Class. In block diagram, our basic algorithm described in Section 6 is composed of 4 blocks shown in the 'Default Steps' box in Figure 6. We can then insert additional blocks to identify equivalence classes, rank generators and mine a single generator with the maximum rank for every equivalence class.
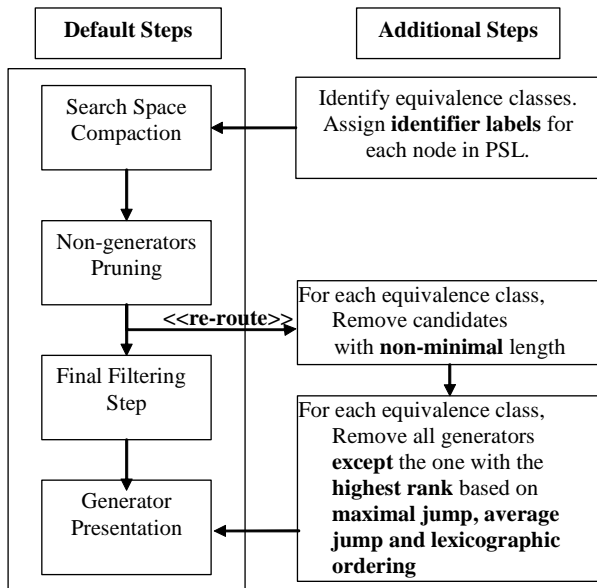
Figure 6: Algorithm Block Diagram - GenMiner-EQ

During the PSL building step, assignments of tags can be performed on the fly using the method described in sub-section 7.1. After pruning of non-generators we will obtain a set of candidate generators. From this set, for each equivalence class, we will remove candidates with non-minimum length.

Note that every candidate is a pattern in an equivalence class. If a pattern is one of the shortest patterns in the class, it must be a generator. This is the case since none of its sub-sequences is in the class. Hence, we can skip the final filtering step and proceed to further

rank the generators based on maximal jump, average jump and lexicographic order. The end result is a set of generators each uniquely corresponds to the highest rank generator of an equivalence class. We refer to the modified algorithm returning one generator per equivalence class as GenMiner-EQ. The number of generators returned will correspond to the number of equivalence classes in the input sequence database.

Let us consider the frequent pattern space shown in Figure 1 and its corresponding database. There are 8 equivalence classes, hence GenMiner-EQ should return 8 generators. Three of the classes, EQ_5, EQ_7 and EQ_8 each contains more than one generator. For EQ_8, there are 2 generators, namely $\langle A, B, B \rangle$ and $\langle C \rangle$. Ranking them based on pattern length, $\langle C \rangle$ has a higher rank than $\langle A, B, B \rangle$. Hence, we keep $\langle C \rangle$. For EQ_7, there are 2 generators, namely $\langle A, B \rangle$ and $\langle B, B \rangle$. These two are of the same length, hence we rank them based on their maximum jump. Generators $\langle A, B \rangle$ and $\langle B, B \rangle$'s maximum jumps are 1.67 and 1.33 respectively. Hence, we keep the one with the larger maximum jump, which is $\langle A, B \rangle$. For EQ_5, there are 3 generators, namely $\langle A, A \rangle$, $\langle A, D \rangle$ and $\langle D, A \rangle$. All three generators have the same length and the same maximum jump. Hence, we sort them based on average jump. The average jumps of the generators $\langle A, A \rangle$, $\langle A, D \rangle$ and $\langle D, A \rangle$ are 2.5, 2 and 2 respectively. Hence we keep the one with the highest average jump which is $\langle A, A \rangle$. Each of the generators returned by GenMiner-EQ corresponds to the highest rank generator in an equivalence class. For the frequent pattern space in Figure 1, the set of highest rank generators is $\{\langle\rangle, \langle A \rangle, \langle B \rangle, \langle C \rangle, \langle D \rangle, \langle A, A \rangle, \langle A, B \rangle, \langle B, A \rangle\}$.

8 Performance Study

A performance study was performed on both synthetic and real datasets to demonstrate the scalability of our method. Also, we would like to compare the performance of our algorithms with that of algorithms mining a full set of frequent sequential patterns (as a baseline) and a closed set of sequential patterns. The problem of mining sequential generators is novel and hence it is not our aim to reduce the runtime below that of closed pattern miners. They mine different things and thus are not comparable. Rather, our aim is to show that mining generators can be as efficient as closed patterns. This study also presents new insight on the performance of existing closed pattern miners.

Datasets. We use three datasets in our study: a synthetic and two real datasets. Synthetic data generator provided by IBM was used with modification to ensure generation of single-item sequences (*i.e.*, transactions are of size 1). We also experimented on a click stream dataset (*i.e.*, Gazelle dataset) from KDD Cup

2000 [11] which was also used to evaluate CloSpan [27] and BIDE [26]. It contains 29369 sequences with an average length of 3 and a maximum length of 651.

Recently, there have been active interests in analyzing program traces (*e.g.*, [2, 16, 18, 17, 19]). We generate traces from a simple Traffic alert and Collision Avoidance System (TCAS) from Siemens Test Suite [10] which is used as one of benchmarks for research in error localization (*e.g.*, [4]). The test suite comes with 1578 correct test cases. We run the test cases and obtain 1578 traces. Each trace is treated as a sequence. The sequences are of average length of 61 and maximum length of 97. It contains 106 different events – the events are the line numbers of the statements being executed. We refer to it as the TCAS dataset.
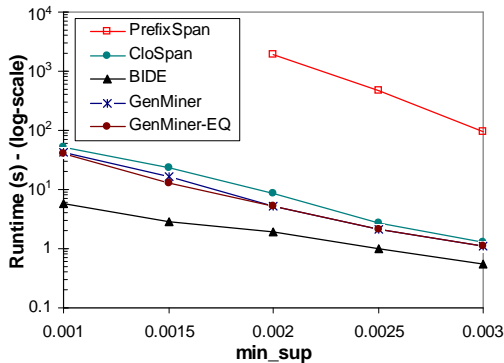
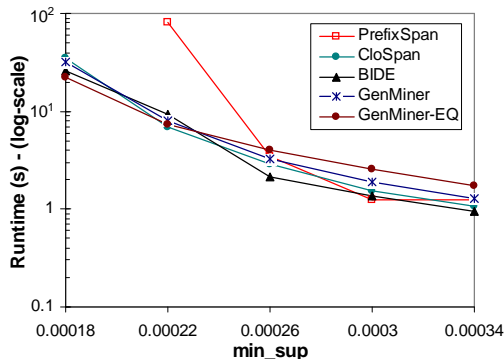Figure 7: **Performance results of varying** min_sup **for the D5C20N10S20 dataset**

Figure 8: **Performance results of varying** min_sup **for the Gazelle dataset**

Pattern Miner and Environment. 'Full-set' frequent pattern miner used is PrefixSpan [21]. Closed pattern miners used are CloSpan [27] and BIDE [26]. We compare two versions of our algorithm: with and without ranking. We refer to the algorithm returning all generators and the one returning one generator per equivalence class as GenMiner and GenMiner-EQ respectively. This study was performed on an Intel Pentium 4, 3.0 GHz PC with a 2GB main memory, running Windows XP Professional. PrefixSpan, CloSpan, BIDE,
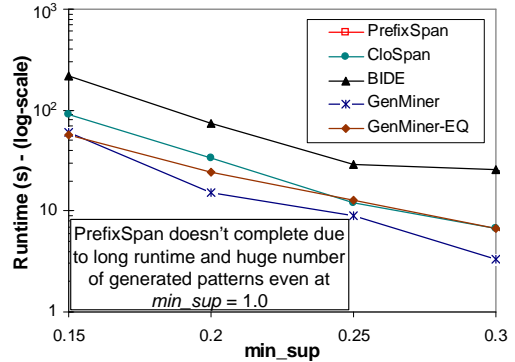

Figure 9: **Performance results of varying** min_sup **for the TCAS dataset**

GenMiner and GenMiner-EQ are written in C++.
Experiment Results and Analysis. The experiment results for the synthetic dataset D5C20N10S20 are shown in Figure 7. The parameters D, C, N and S correspond to the number of sequences (in 1000s), the average number of events per sequence, the number of different items (in 1000s) and the average number of events in the maximal sequences. Experiment results using the Gazelle and TCAS datasets are shown in Figures 8 & 9 respectively. The ranges considered for the D5C20N10S20 and Gazelle datasets follow similar ranges considered in [27, 26]. Among the 3 datasets, the dataset TCAS is the hardest to mine, shown by the fact that the performance is poorer at much higher support thresholds.

From the plotted results, it is noted that the performance of our algorithms and closed pattern miners are comparable. GenMiner and GenMiner-EQ perform best at TCAS dataset. BIDE performs best at the simulated and Gazelle dataset. Also, the performance of GenMiner and GenMiner-EQ are comparable. GenMiner-EQ needs to identify equivalence classes and compute identifier labels to nodes in PSL which increase the computation cost. However, GenMiner-EQ can skip the final filtering step which reduces the computation costs. In the experiments, these opposite factors cancel each other out and the performance of the two algorithms are more or less comparable.

TCAS dataset is interesting as the 'full-set' frequent pattern miner is not able to complete even at the support level of 100% (there is only one closed pattern at the support level of 100% and it is of length 27). ~~Also, from the TCAS dataset results plotted in Figure 9, performance of BIDE is worse than our algorithm (GenMiner and GenMiner-EQ) and CloSpan. Compared with Gazelle dataset, the average length of sequences in TCAS dataset is much longer (61 vs. 3) and the closed patterns are long and few. The long sequences cause the BackScan operation of BIDE to~~

~~be costly.~~[3]. Also, at support level 0.15, BIDE requires more memory (over 200MB) than GenMiner (33MB), GenMiner-EQ (35MB) and CloSpan (22MB). BIDE doesn't require to keep a tree or lattice in memory, while the other 3 algorithms need to do so. However, using BIDE for a closed pattern of length X, it needs to store X projected databases in the memory (at min_sup of 0.15, the longest pattern is of length 85). On the other hand, the other 3 algorithms rely on the equivalence of projected databases. A long pattern of length X in effect is built by a series of concatenation of shorter patterns. This finding sheds new light on closed sequential pattern mining. In [26], it is demonstrated that on Gazelle and various biological datasets, the performance of BIDE is significantly faster by up to an order of magnitude while the memory consumption is order(s) of magnitude less as compared to CloSpan. The above result shows that there is still room for improvement to develop new closed sequential pattern miners which excel in all datasets.

Similar to related studies in [27, 26, 3, 14, 13, 21], in this paper, we focus on mining algorithms and performance issues. As future work, we are investigating using mined sequential generators for the classification of sequential data. Also, we are looking into using both generators and closed patterns to form rules with shortest pre-conditions and longest post-conditions.
Post Conference Clarification & Correction. In the above performance study result, we compile BIDE source code using Visual Studio .Net 2005 (with release, i.e., optimized, option). Thanks to Jianyong Wang enquiry and advice we try to run a binary compiled using Visual Studio 6. We find that the with the latter, the above result is reversed. BIDE is faster than the other 3 algorithms on the TCAS dataset. We are not aware of this performance slowdown due to compiling using Visual Studio .Net 2005 before. We thus retract our argument on runtime performance of BIDE and CloSpan (see the text being strike-throughed above).

The result on peak memory requirement stays the same and hence we encourage researchers in closed sequential pattern mining to continue working in this area to further develop closed sequential pattern miners which excel in all datasets in terms of memory and runtime. Note that the thrust of the paper is to present a novel technique on mining sequential generators.

9 Conclusion

Sequential patterns supported by the same set of sequences in the database belong to the same equivalence class. Closed patterns and generators of an equivalence class correspond to the maximal and minimal mem-
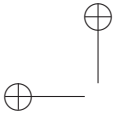
---

bers of the class respectively. Generators used together with closed patterns can provide additional information which closed patterns alone are not able to provide. Also, as generators are usually shorter than closed patterns, they are preferable over closed patterns for model selection and classification based on the Minimum Description Length (MDL) principle. Several algorithms have been proposed on mining closed sequential patterns. However, there has not been any algorithm proposed for mining sequential generators. In this paper, we fill this research gap by proposing a novel algorithm to mine sequential generators efficiently. We also introduce ranking of mined generators and propose mining of a unique generator per equivalence class. Performance study shows that our method can run a lot faster than mining a full-set of frequent sequential patterns and its speed can be on par with or at times faster than that of mining a closed set of sequential patterns.

References

[1] R. Agrawal and R. Srikant. Mining sequential patterns. In *ICDE*, 1995.
[2] G. Ammons, R. Bodik, and J. R. Larus. Mining specification. In *SIGPLAN-SIGACT POPL*, 2002.
[3] J. Ayres, J.E. Gehrke, T. Yiu, and J. Flannick. Sequential pattern mining using bitmaps. In *KDD*, 2002.
[4] H. Cleve and A. Zeller. Fault localization: Locating causes of program failures. In *ICSE*, 2005.
[5] G. Dong and J. Pei. *Sequence Data Mining*. Springer, 2007.
[6] M. Garofalakis, R. Rastogi, and K. Shim. SPIRIT: Sequential pattern mining with regular expression constraints. In *VLDB*, 1999.
[7] P. Grunwald, I.J. Myung, and M. Pitt. *Advances in Minimum Description Length: Theory and Applications.* MIT Press, 2005.
[8] J. Han and M. Kamber. *Data Mining Concepts and Techniques.* Morgan Kaufmann, 2001.
[9] Y. Hu and B. Panda. A data mining approach for database intrusion detection. In *SIGAPP SAC*, 2004.
[10] M. Hutchins, H. Foster, T. Goradia, and T. Ostrand. Experiments of the effectiveness of dataflow- and controlflow-based test adequacy criteria. In *ICSE*, 1994.
[11] R. Kohavi, C. Brodley, B. Frasca, L. Mason, and Z. Zheng. KDD-Cup 2000 organizers' report: Peeling the onion. *SIGKDD Explorations*, 2:86–98, 2000.
[12] R. Kosala and H. Blockeel. Web mining research: A survey. *SIGKDD Explorations*, pages 1–15, 2000.
[13] J. Li, H. Li, L. Wong, J. Pei, and G. Dong. Minimum description length principle: Generators are preferable to closed patterns. In *AAAI*, 2006.
[14] J. Li, G. Liu, and L. Wong. Mining statistically important equivalence classes and delta-discriminative emerging patterns. In *KDD*, 2007.
[15] M. Li and P. Vitanyi. *An Introduction to Kolmogorov Complexity and Its Applications.* Springer-Verlag, 1999.
[16] D. Lo and S-C. Khoo. SMArTIC: Toward building an accurate, robust and scalable specification miner. In *SIGSOFT FSE*, 2006.

---

[3]See Post Conference Clarification & Correction.

[17] D. Lo and S-C. Khoo. Software specification discovery : A new data mining approach. In *NSF NGDM*, 2007.

[18] D. Lo, S-C. Khoo, and C. Liu. Efficient mining of iterative patterns for software specification discovery. In *KDD*, 2007.

[19] D. Lo, S-C. Khoo, and C. Liu. Efficient mining of recurrent rules from a sequence database. In *DASFAA*, 2008.

[20] N. Pasquier, Y. Bastide, R. Taouil, and L. Lakhal. Discovering frequent closed itemsets for association rules. In *ICDT*, 1999.

[21] 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.

[22] A. Raman, P. Andreae, and J.D. Patrick. A beam search algorithm for PFSA inference. *Pattern Analysis and Applications*, 1998.

[23] A.V. Raman and J.D. Patrick. The sk-strings method for inferring PFSA. In *Proc. of the Work. on Automata Ind., Gram. Inf.and Lang. Acq. at ICML*, 1997.

[24] J. Rissanen. Modelling by shortest data description. *Automatica*, 14:465–471, 1978.

[25] M. Spiliopoulou. Managing interesting rules in sequence mining. In *PKDD*, 1999.

[26] J. Wang and J. Han. BIDE: Efficient mining of frequent closed sequences. In *ICDE*, 2004.

[27] X. Yan, J. Han, and R. Afshar. CloSpan: Mining closed sequential patterns in large datasets. In *SDM*, 2003.

[28] M. Zaki. SPADE: An efficient algorithm for mining frequent sequences. *Machine Learning*, 40:31–60, 2001.

[29] M.J. Zaki. Mining data in bioinformatics. *Handbook of Data Mining*, pages 573–596, 2003.

[30] M.J. Zaki and C.J. Hsiao. Charm: An efficient algorithm for closed itemset mining. In *SDM*, 2002.