JOURNAL OF INFORMATION SCIENCE AND ENGINEERING 36, 145-169 (2020)
DOI: 10.6688/JISE.202001_36(1).0009

On Mining Progressive Positive and Negative Sequential Patterns Simultaneously

JEN-WEI HUANG, YONG-BIN WU AND BIJAY PRASAD JAYSAWAL
Institute of Computer and Communication Engineering
National Cheng Kung University
Tainan, 701 Taiwan
E-mail: {jwhuang; Q36021135}@mail.ncku.edu.tw; bijay@jaysawal.com.np

Positive sequential pattern (PSP) mining focuses on appearing items, while negative sequential pattern (NSP) mining tends to find the relationship between occurring and non-occurring items. There are few works involved in NSP mining, and the definitions of NSP are inconsistent in each work. The support threshold for PSP is always applied on NSP, which cannot bring out interesting patterns. In addition, PSP has been discovered on incremental databases and progressive databases, while NSP mining is only performed on static databases. Progressive sequential pattern mining finds the most up-to-date patterns, which can provide more valuable information. However, the previous progressive sequential pattern mining algorithm contains some redundant process. In this paper, we aim to find NSP on progressive databases. A new definition of NSP is given to discover more meaningful and interesting patterns. We propose an algorithm, Propone, for efficient mining process. We also propose a level-order traversal strategy and a pruning strategy to reduce the calculation time and the number of negative sequential candidates (NSC). By comparing Propone with some modified previous algorithms, the experimental results show that Propone outperforms comparative algorithms.

Keywords: progressive mining, negative sequential pattern, frequency ratio of interest, sequential pattern mining, data mining

1. INTRODUCTION

With advances in technology, the amount of data that is now being collected has increased dramatically, along with its diversity. However, finding useful information hidden in such data is very difficult. Therefore, data mining is applied to mine valuable knowledge from a large amount of data. The knowledge and information discovered in this way can be used for a wide range of applications, such as market analysis, biotechnology, customer behavior analysis, and fraud detection.

Among these research topics, sequential pattern mining has been extensively studied and attracted a lot of research interest. The concept of sequential pattern mining was first introduced by Agrawal and Srikant in [1], which described the problem as follows: "Given a sequence database, where each sequence consists of an ordered list of elements and each element contains a set of items, and a user-defined minimum support threshold *min_sup*, sequential pattern mining aims to find all subsequences whose occurrence frequency is no less than *min_sup* in the set of sequences." Sequential pattern mining is important because it can be applied in many applications related to sequence data, such as customer

Received February 26, 2018; revised July 25, 2018; accepted August 29, 2018.
Communicated by Wen-Chih Peng.

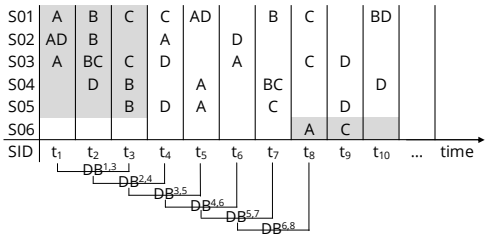


Fig. 1. An example progressive database.

purchasing records, web clickstreams, DNA sequences, sequences of the symptoms of a patient, and sequences of social events. Extensive research have been done on the mining of sequential patterns in a large database [2–5].

The sequential pattern mining algorithms mentioned above focus on static databases. In the other words, the data in these works do not change over time. The algorithms mine the whole database and return a complete set of results. However, in the real world, many applications deal with databases where the amount of data grows incrementally. Static sequential pattern mining algorithms are not suitable for such databases, as the results could be incorrect once the database is updated. In addition, re-mining the updated database is an inefficient process. Some algorithms for incremental sequential pattern mining have been proposed to solve this problem, such as [6–9].

Incremental sequential pattern mining handles newly arriving data and finds sequential patterns in the updated database. However, an incremental database where the old data stays may not be applicable for some applications. Finding more recent or up-to-date sequential patterns without considering the obsolete data is thus of interest, because this could provide more accurate and valuable information. A database which not only adds new data but also removes obsolete data at the same time is defined as a "progressive database [10]." In addition, new sequential patterns in recent data may not be considered as frequent sequential patterns if the database size is never reduced. Therefore, those sequences which have no newly arriving data should be removed from the database.

Neither static sequential pattern mining nor incremental sequential pattern mining can cope efficiently with progressive databases. Therefore, progressive sequential pattern mining was introduced in [10]. The authors proposed an efficient algorithm called Progressive mIning of Sequential pAtterns (Pisa) for mining sequential patterns in a progressive database. In [10], the concept of period of interest (POI) was first introduced, which is used for the progressive mining task. POI is defined below.

Definition 1 ***Period of Interest (POI)***
POI is a sliding window whose length is user-defined, advancing with time. The part within the POI can be seen as a subset of the whole database. The sequences having elements whose timestamp falls in this period, POI, contribute to the size of the database for the current sequential patterns. However, the sequences having only elements whose timestamp is older than the POI should be pruned away from the database and do not contribute to the size of the database.

As per the POI, progressive sequential pattern mining finds the complete set of frequent sequential patterns for each progressing POI whose occurrence frequency is greater than the user-defined threshold min_sup times the number of sequences in the current POI of the progressive database. Fig. 1 shows an example of a progressive database. There are six sequences S01 to S06 and four different items A, B, C, and D in the database. t_1 to t_{10}

represent timestamps, and for each timestamp there may be some elements which contain single or multiple items appearing in different sequences. The length of the POI is three timestamps in this example, and thus each subset of the database contains elements in a particular interval of three successive timestamps. A subset of the database is denoted by $DB^{p,q}$, where p and q are the start and the end timestamps respectively. Suppose the *min_sup* is 0.5, then $< A\ B >$ is a frequent pattern in $DB^{1,3}$ since its occurrence frequency, 3, is greater than 5 (the size of $DB^{1,3}$) * 0.5 = 2.5. However, $< A\ B >$ is no longer a frequent pattern in the later POIs.

Rather than mining items occurring in the database, some research focuses on non-occurring items. Non-occurring items are called *negative* items, and sequential patterns consisting of both occurring and non-occurring items are called Negative Sequential Patterns (NSP). In contrast, the sequential patterns composed of only positive items, *i.e.*, only occurring items, are also called Positive Sequential Patterns (PSP). The notation of a negative item or element begins with a $\neg$, for example, $s_1 = < A\ B\ C >$ is a positive sequence and $s_2 = < A\ \neg B\ C >$ is a negative sequence without B occurring between A and C. The relationship between occurring and non-occurring items can be useful and give important information. For example, the reaction to a series of treatments that were provided or not provided may help monitor patients' conditions. Zhao *et al*. [11] improved the accuracy of debt detection by using both PSP and NSP rather than PSP alone for the sequence classification. However, NSP cannot be found by traditional sequential pattern mining algorithms, and although there are also some works on the mining of NSP [12–14], the development is very limited. In [12–14], the Negative Sequential Candidates (NSC) are generated by joining the shorter patterns, resulting in an enormous number of candidates. The calculation of the number of sequences that contain a NSC also requires multiple scans of the database. For these two reasons, the mining of NSP is difficult and it is not easy to find meaningful NSP in a database. Dong *et al*. [15] then proposed the concept of converting the "negative containment" problem into a positive containment problem, and presented an algorithm, named e-NSP, to mine NSP efficiently. By applying the conversion, the support of an NSC can be calculated by using the mined PSP instead of scanning the database many times.

Although the concept of NSP, *i.e.*, sequential patterns containing negative items or elements, has been discussed for several years, the definitions of NSP and negative containment remain inconsistent in these previous works. For example, in [12], a data sequence is not allowed to contain an NSC which has a negative element before the beginning element of the data sequence, while that in [13–15] is. The authors had different views on whether a data sequence can contain an NSC because the negative containment problem may vary depending on the applications. Besides, to the best of our knowledge, so far NSP mining has been performed only on static databases. However, NSP can provide more information when mined from a progressive database because of the dynamic nature of progressive database where new data are added and obsolete data are deleted based on the POI. The patterns mined using progressive database are not influenced by the obsolete data and thus NSP in progressive database provide information based on the data inside a POI and without influence of obsolete data. We thus combine progressive sequential pattern mining and negative sequential pattern mining, leveraging the notions of data structure and negative containment from [10] and [15], respectively. The data structure used in [10] is a tree structure called a , and it records the information of all sequential patterns in the database. For NSP mining, we follow the constraints on negative sequences and the concept of negative containment in [15]. However, in [10], the PS-tree is maintained by the Pisa algorithm with the post-order traversal, and thus the descendants

of a tree node cannot be pruned efficiently. In addition, in the previous works, the NSP are defined as negative sequences whose occurrence frequency is greater than the user-defined threshold *min_sup*, which is the same as that for PSP. While the number of NSP can still be large with such a threshold, most of them are not necessarily meaningful. To improve the mining process and find NSP which better fit the users' needs, in this paper we propose an efficient algorithm **Propone**, which stands for **PROgressive mining of POsitive and NEgative sequential patterns**, combining progressive and negative sequential pattern mining. Instead of maintaining the PS-tree in post-order, **Propone** processes the nodes of the PS-tree in level-order, avoiding the redundant processing of the nodes that will be performed if post-order traversal is adopted. We also give a new definition of NSP, which requires the occurrence frequency of an NSC to be greater than multiple times that of the NSC's corresponding PSP, *i.e.*, the positive sequence obtained by turning each element in the NSC into a positive one, rather than the same threshold *min_sup* which is for PSP. The ratio between an NSC's frequency and PSP's is defined as the **Frequency Ratio of Interest (FRI)**. This new definition helps users find the NSP that contradicts the PSP, so that they can know which elements in the PSP may not be necessary, thus improving their decision making strategies. Moreover, by combining NSP with the concept of POI, the POI can help find out the longest period where a negative element does not occur. This is helpful when we want to say that something not occurring leads to some other things occurring. Some approaches for pruning the infrequent NSC are proposed and adopted in **Propone** to reduce the search space. The experimental results show that **Propone** speeds up the mining process by preventing the nodes which should be deleted from being processed. When the minimum support is small or the number of sequences is large, the advantages of **Propone** become more significant. The pruning approaches also have good effects on reducing the number of infrequent NSC, especially when FRI gets larger.

The rest of the paper is organized as follows. Section 2 introduces some previous works on PSP mining and NSP mining. The preliminaries of PSP and NSP and the problem definition are given in Section 3. Section 4 describes the algorithm **Propone** in detail. We present and analyze its performance in Section 5. Finally, the conclusions of this work are given in Section 6.

2. RELATED WORK

In this section, we introduce the works related to PSP mining and NSP mining. PSP mining can be separated into static, incremental, and progressive sequential pattern mining, depending on the type of the databases used.

2.1 Positive Sequential Pattern Mining

Sequential pattern mining was first addressed by Agrawal and Srikant [1], and initially focused on static databases. AprioriAll [1] and GSP [2], which are very important for the development of sequential pattern mining, use a property "*any superset of an infrequent subset is infrequent*" from an association rule mining algorithm, Apriori [16], to prune infrequent candidates effectively. Zaki proposed SPADE [3] that transforms the database into a vertical id-list database format and utilizes lattice search techniques. Pei *et al*. proposed PrefixSpan [4] that utilizes the pattern growth approach and takes advantage of the projected databases. SPAM [5] was then introduced by Ayres *et al*., using a vertical bitmap representation of the database and a depth-first traversal of a lexicographic sequence lattice for efficient candidate generation and support counting. Besides these typical sequential pattern mining algorithms, several extensions have been proposed

for different fields and different types of data. Some works were proposed on closed sequential pattern mining [17–19] and on maximal sequential pattern mining [20, 21] in order to obtain a compact result set. Constrained sequential pattern mining [22–24] helps in cases where some constraints are required. Several other extensions with various kinds of data are also presented, for example, data stream mining [25, 26], and temporal pattern mining [27–29].

Static databases may not be suitable for cases in the real world, because data can grow with time. New data are added into database over time and this type of database is called incremental database. The typical sequential pattern mining algorithms for static databases cannot maintain the sequential patterns efficiently. Incremental sequential pattern mining algorithms [6–9] were thus proposed to deal with such incremental databases.

Updating databases by only adding new data to the original databases cannot meet the needs of some situations, as the old data may affect the quality of the results. However, maintaining the sequential patterns while removing the obsolete data and appending new ones at the same time is a difficult task. To handle this problem, Huang *et al*. [10] proposed the concept of a progressive database which considers updating database not only by addition of the new data, but also by the removal of old data. The Pisa algorithm in [10] uses a special data structure named a PS-tree to store information about all the sequential patterns in a database. The PS-tree is then maintained by Pisa in the post-order, including the insertion of the information of new data, removal of the obsolete candidates, and the updating of existing nodes.

2.2 Negative Sequential Pattern Mining

There are a limited number of works on NSP mining, but to the best of our knowledge these only deal with static databases. Lin *et al*. proposed the algorithms NSPM [30] and PNSPM [31], and allowed only the last element of a negative sequence to be negative. Ouyang *et al*. provided the GNSP approach [32] to mine sequential patterns which are in the form of the relation between a pair of itemsets. Another work by Zhao *et al*. [33] is similar to [32], and aims to find positive and negative impact-oriented sequential rules. Ouyang *et al*. proposed an algorithm called CPNFSP [34], which is able to deal with multiple minimum supports so that sequential patterns hidden among the rare sequences can be found. Hsueh *et al*. proposed the algorithm PNSP [12] that uses the concepts of n-cover and n-contain to determine whether a negative candidate is contained in a data sequence. Zheng *et al*. introduced the algorithm Negative-GSP [13] based on the PSP mining algorithm GSP, using joining and pruning strategies. The authors of [13] presented another work [14] which applies the concept of the genetic algorithm (GA). The GA-based NSP mining utilizes the crossover and mutation operations, avoiding the generation of negative candidates. Dong *et al*. proposed the conversion of the negative containment problem into a positive containment problem in their algorithm, the e-NSP [15], to prevent the re-scanning of the database. With their conversion method the support of an NSC can be calculated by using the mined PSP instead of re-scanning the database. The conversion strategy and the approach to candidate generation are applied in this work to efficiently mine NSP. However, the definition of NSP, the negative containment, and the constraints on negative sequences are different in these earlier works. For instance, NSP in [12] are required to satisfy both the min_sup and an extra parameter $miss\_freq$ while there is only the min_sup in most of the algorithms. In addition, the authors of [12] do not allow a data sequence to contain a negative sequence which has negative elements before the corresponding first element in the data sequence or after the last one, while those of [13] and [14] do. For example, a data sequence $ds = < B\ C >$ does not contain negative

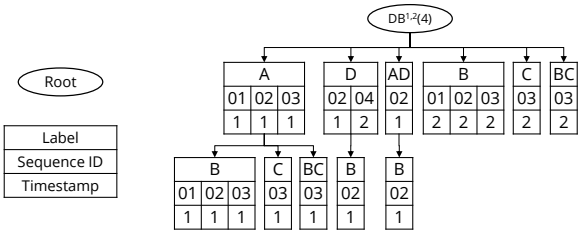


Fig. 2. *left*) Root and common node; *right*) an example PS-tree.

sequences $n_1 = < \neg A\ B >$ and $n_2 = < B\ \neg C >$. We present a different perspective on the definition of NSP but keep the same constraints as those in [15], in order to find NSP that are of more interest to users.

3. PRELIMINARIES

3.1 Positive Sequential Pattern (PSP)

First we give the definitions in typical sequential pattern mining. Let $I = \{I_1, I_2, ..., I_n\}$ be a set of distinct items. An *element* e (or called *itemset*), denoted by $(I_i I_j ...)$, is a subset of I whose items appear at the same time. A *sequence* $s = < e_1 e_2 ... e_m >$ is an ordered list of elements. Without loss of generality, we assume that the items in an element are ordered alphabetically. The *size* of a sequence, denoted by $|s|$, is the number of elements in the sequence. The *length* of a sequence, denoted by $l(s)$, is the total number of items in each element in the sequence. A sequence $\alpha = < a_1, a_2, ..., a_n >$ is a *subsequence* of another sequence $\beta = < b_1, b_2, ..., b_m >$ and β is a *supersequence* of α, denoted by $\alpha \sqsubseteq \beta$, if there exist integers $1 \leq i_1 < i_2 < \cdots < i_n \leq m$ such that $a_1 \subseteq b_{i_1}, a_2 \subseteq b_{i_2}, ...,$ *and* $a_n \subseteq b_{i_n}$. We say β *contains* α if α is a subsequence of β. A sequence database $DB = \{s_1, s_2, ..., s_k\}$ is a set of sequences and $|DB|$ represents the number of sequences in DB. Each sequence in the sequence database has a sequence id, *sid*. The *support set* of a sequence α in a database DB, denoted by $\{\alpha\}$, is the set of sequences in DB containing α, and the *support* of α, denoted by $sup(\alpha)$, is the length of the support set, that is, $sup(\alpha) = |\{s | s \in DB \text{ and } \alpha \sqsubseteq s\}|$. Given a user-defined minimum support threshold *min_sup*, a sequence α is $frequent$ in a database DB if $sup(\alpha) \geq min\_sup * |DB|$. A frequent sequence is also called a $(positive)$ $sequential\ pattern$.

For progressive sequential pattern mining, each element in a sequence is associated with a *timestamp* at which the element appears. A sequence is a list of elements ordered by $timestamp$ in ascending order. A time interval, denoted by $[p,q]$, represents the period from timestamp p to timestamp q. The subset of a database DB containing the elements of sequences in the time interval $[p,q]$ is denoted by $DB^{p,q}$.

The data structure adopted by Pisa [10] is PS-tree. The right side of Fig. 2 shows an example PS-tree, which represents the PS-tree of the example database in Fig. 1 from t_1 to t_2. PS-tree stores the information of all sequences and candidates in a progressive database and keeps the frequent sequential patterns in each POI. There are two types of nodes in a PS-tree: the root and common nodes, as shown on the left side of Fig. 2. The root of PS-tree is an empty node with only a list of common nodes as its children. Every common node contains a label and a sequence list. The label represents an element in a sequence. The sequence list records the ids of the sequences where this element appears. Each sequence id in the sequence list is associated with a timestamp, showing the starting timestamp of a sequential candidate contained in this sequence. When there is a sequence

with a single or multiple elements, a node with the first element as the label and the sequence id is connected to the root, then another node with the second element, if any, and the same sequence id will be connected to the first node. The following nodes are connected in the same manner. Note that, different from [10], we keep the timestamps in the nodes that are below the second level as well, because we mine the sequential patterns with repeated elements.

3.2 Negative Sequential Pattern (NSP)

Negative sequences are sequences with non-occurring items. The non-occurring items or elements are called *negative items/elements*. The symbol $\neg$ is used to indicate the negative items and elements. However, using only this definition, the number of negative sequences can be considerably large, but most of them are meaningless. Constraints should thus be placed on negative sequences to reduce the number of NSC and find useful NSP.

3.2.1 Constraints on negative sequence

There are numerous possible combinations to generate an NSC, but most of them lack useful information. To simplify the problem and avoid considering meaningless sequences, this work applies some constraints to negative sequences. Let us take the frequency of sequences as an example. Users tend to be more interested in the absences of certain items or elements in a frequent sequential pattern. In addition, PSP is the most commonly used information in sequential pattern mining that users then apply to their decision making. Therefore, we focus on the negative sequences related to the frequent PSP. Other related constraints are introduced below.

Definition 2 ***Positive Partner***
The positive partner of a negative element $\neg e$, denoted by $p(\neg e)$, is its corresponding positive element e. The positive partner of a positive element e, denoted by $p(e)$, is e itself. The positive partner of a negative sequence $ns = < e_1 e_2 ... e_n >$, denoted by $p(ns)$, is a positive sequence obtained by changing each element in ns to their positive partners, i.e., $p(ns) = < e'_1 e'_2 ... e'_n >$ where $\forall e'_i \in p(ns), e'_i = p(e_i)$. For example, $p(< A \neg(BC)\ (AD)\ \neg E >) = < A\ (BC)\ (AD)\ E >$.

Constraint 1 ***Frequency constraint***
The positive partner of a negative sequence ns should be frequent. This is because users tend to be more interested in the absences of certain items or elements in a frequent sequential pattern. Note that we follow the constraint in [15], while in [12] and [13] only the positive partner of each element in ns is required to be frequent.

Constraint 2 ***Continuity constraint***
A negative sequence must not have any contiguous negative elements. For example, $< A \neg(BC)\ \neg B\ D >$ is not allowed. Here, positions of the two contiguous negative elements can be exchanged, while the meaning of the sequence is still the same, i.e., there are no (BC) and B between A and D. This constraint is the same as Constraint 2 in [15].

Constraint 3 ***Element consistency constraint***
The minimum negative unit in a negative sequence is an element, that is, all the items in an element should be positive or negative at the same time. For example, $< A\ (\neg BC)\ D >$ is not allowed while $< A\ \neg(BC)\ D >$ is. It is because we can add as many items as we

want to the element $(\neg BC)$*, such as* $(\neg B \neg DC)$*, but the meanings are the same, that is, a simple C appearing. This constraint is the same as Constraint 3 in [15].*

In this work, we focus on the negative sequences satisfying these three constraints and generate NSC.

3.2.2 Subsequence of negative sequence

Given an arbitrary, *i.e.*, positive or negative, sequence $\alpha = < a_1, a_2, ..., a_n >$ and a negative sequence $\beta = < b_1, b_2, ..., b_m >$ where a_i and b_j are either positive or negative elements, $1 \leq i \leq n, 1 \leq j \leq m$, we say α is a subsequence of β and β is a supersequence of α, denoted by $\alpha \sqsubseteq_n \beta$, if there exist integers $1 \leq i_1 < i_2 < \cdots < i_n \leq m$ such that $a_1 = b_{i_1}, a_2 = b_{i_2}, ...,$ *and* $a_n = b_{i_n}$. In addition, the subsequence which contains all the positive element(s) in β is the *Maximum Positive Subsequence* of β, denoted by $MPS(\beta)$.

Example 1 *Given a negative sequence* $\beta = < A\ \neg(AC)\ B\ \neg(DE) >$*,* $< A\ \neg(AC)\ B >$ *and* $< \neg(AC)\ B\ \neg(DE) >$ *are subsequences of* β*, while* $< A\ \neg(AC)\ \neg(DE) >$ *is not because there are two contiguous negative elements.* $< A\ B >$ *is the maximum positive subsequence of* β*, i.e.,* $MPS(\beta) = < A\ B >$*.*

3.2.3 Support counting of negative sequences

The support counting of negative sequences can be very complicated. In earlier works, such as [12] and [13], the task of support counting involves multiple scans of the databases However, the authors of [15] proposed a conversion strategy to convert the problem of negative containment to the positive containment that helps to calculate the support of negative sequences using the mined PSP instead of rescanning the databases. For more details, we refer the readers to [15]. To describe the negative containment, a special sequence named *1-neg-size Maximum Subsequence* (1-NMS) is first defined.

Definition 3 ***1-neg-size Maximum Subsequence (1-NMS)***
Given a negative sequence ns, let E_n *be the set of all negative elements in ns. A subsequence of ns that includes the maximum positive subsequence* $MPS(ns)$*, and one negative element* $e \in E_n$*, is called a 1-neg-size Maximum Subsequence (1-NMS) of ns, denoted by* $1\text{-}NMS(ns, e)$*. The set of all 1-neg-size maximum subsequences of ns is called 1-neg-size maximum subsequence set, denoted by* $1\text{-}NMSS(ns)$*, i.e.,* $1\text{-}NMSS(ns) = \{1\text{-}NMS(ns, e_i) | \forall e_i \in E_n\}$*.*

Example 2 *Given a negative sequence* $ns = < A\ \neg(AC)\ B\ \neg(DE) >$*,* $1\text{-}NMS(ns, \neg(AC)) = < A\ \neg(AC)\ B >$ *is a 1-neg-size maximum subsequence of ns. The 1-neg-size maximum subsequence set of ns is* $1\text{-}NMSS(ns) = \{< A\ \neg(AC)\ B >, < A\ B\ \neg(DE) >\}$*.*

Definition 4 ***Negative Containment***
Given a data sequence $ds = < d_1, d_2, ..., d_k >$*, and a negative sequence* $ns = < e_1, e_2, ..., e_m >$*, the data sequence ds contains the negative sequence ns if both the following conditions hold: (1)* $MPS(ns) \subseteq ds$*; (2)* $\forall 1\text{-}NMS \in 1\text{-}NMSS(ns),\ p(1\text{-}NMS) \nsubseteq ds$*.*

Example 3 *Given a data sequence* $ds = < A\ (BCD)\ A\ (DE)\ C >$ *and negative sequences* $ns_1 = < A\ \neg E\ (DE)\ \neg(AB) >$ *and* $ns_2 = < \neg B\ A\ \neg(BD)\ C >$*, (1) ds contains* ns_1 *because (i)* $MPS(ns_1) = < A\ (DE) > \subseteq ds$*, and (ii)* $1\text{-}NMSS(ns_1) = \{< A\ \neg E\ (DE) >, < A\ (DE)\ \neg(AB) >\}$ *such that* $p(< A\ \neg E\ (DE) >) \nsubseteq ds$ *and* $p(< A\ (DE)\ \neg(AB) >) \nsubseteq ds$*;*

(2) ds does not contain ns_2 *because* $1\text{-}NMSS(ns_2) = \{< \neg B\ A\ C >,\ < A\ \neg(BD)\ C >\}$ *and* $p(< \neg B\ A\ C >) \subseteq ds$.

The support of a negative sequence *ns* is obtained by counting the number of sequences containing $MPS(ns)$ but not containing the positive partners of any negative elements in *ns* between the corresponding neighboring positive elements. We give the formula below.

Corollary 1 ***Support of negative sequence***
Given a m-size and n-neg-size negative sequence ns and a sequence database DB, let $E_n = \{e_1, e_2, ..., e_n\}$ *be the set of all negative elements in ns. The support of the negative sequence ns in DB is:*

$$sup(ns) = |\{ns\}| = |\{MPS(ns)\} - \bigcup_{i=1}^{n}\{p(1\text{-}NMS(ns, e_i))\}| \tag{1}$$

Since $\bigcup_{i=1}^{n}\{p(1\text{-}NMS(ns, e_i))\} \subseteq \{MPS(ns)\}$, *equation (3.1) can be rewritten as:*

$$\begin{aligned} sup(ns) &= |\{MPS(ns)\}| - |\bigcup_{i=1}^{n}\{p(1\text{-}NMS(ns, e_i))\}| \\ &= sup(MPS(ns)) - |\bigcup_{i=1}^{n}\{p(1\text{-}NMS(ns, e_i))\}| \end{aligned} \tag{2}$$

If there is only one negative element in ns, the formula can be simplified as:

$$sup(ns) = sup(MPS(ns)) - sup(p(ns)) \tag{3}$$

Particularly, if ns contains nothing but a negative element $\neg e$, *the support is:*

$$sup(ns) = sup(< \neg e >) = |DB| - sup(< e >) \tag{4}$$

The support of the negative sequences can then be obtained from only the information of the mined PSP instead of the rescans of the database. $|\bigcup_{i=1}^{n}\{p(1\text{-}NMS(ns, e_i))\}|$ can be calculated using the data structure explained in Section 4.2, which is able to quickly access the support of the required PSP from PS-tree.

3.2.4 Definition of NSP

This work aims at finding the NSP which contradicts its positive partner by overwhelming support. Because the support of an NSP can usually be large, applying the same *min_sup* on both PSP and NSP is not feasible. We define that the support of an NSP should be no less than the support of its positive partner times a user-defined parameter, **Frequency Ratio of Interest (FRI)**, so that the NSP is sure to be much more frequent than the corresponding PSP. The definition of NSP is given below.

Definition 5 ***Negative Sequential Pattern (NSP)***
Given a user-defined ***Frequency Ratio of Interest (FRI)*** α, $\alpha \geq 1$, *a negative sequence ns is a Negative Sequential Pattern (NSP) if the support of ns is greater than or equal to the support of the positive partner of ns times* α, *i.e.,* $sup(ns) \geq sup(p(ns)) * \alpha$.

Note that, to the best of our knowledge, all the previous works take the *min_sup* for PSP as the threshold for NSP, except for [12]. The authors of [12] take into account another threshold called *miss_freq*, which is used to limit the support of negative elements in a negative sequence. Finally, we state the problem definition of this work, as follows.

Algorithm 1 : Propone(*DB, min_sup, POI, FRI*)

Require: A progressive database *DB, min_sup, POI, FRI*
Ensure: Output the complete set of PSP and NSP for each POI
1: *PST* = a empty root node; ▷ PS-tree
2: *current_time* $= t_1$; ▷ current timestamp, initialized as the first timestamp
3: *eleSet* $= \phi$; ▷ a set of elements at *current_time*
4: **while** there still exist new transaction in *DB* **do**
5: *eleSet* = read all *ele* at *current_time*;
6: **traverse**(*DB, min_sup, current_time, PST, POI, FRI*);
7: *current_time*++;

Problem Definition 1 *Given a progressive database DB, a length of POI, a minimum support threshold min_sup, and a Frequency Ratio of Interest* α*, find the complete set of both positive and negative sequential patterns in the recent POI of DB.*

4. PROGRESSIVE MINING of POSITIVE and NEGATIVE SEQUENTIAL PATTERNS

In this section, we present the algorithm **Propone** which mines PSP and NSP progressively. **Propone** leverages PS-tree to maintain the information of PSP in the current POI, and applies the techniques of negative containment and support counting of negative sequence to find NSP. The following subsections describe the details of **Propone** algorithm.

4.1 Propone: Algorithm Overview

The framework of **Propone** is composed of mining PSP progressively and finding NSP based on mined PSP. To progressively mine the up-to-date PSP in the database, **Propone** utilizes PS-tree to maintain the information of sequential patterns from one POI to the next. The concept of maintaining the PS-tree is traversing the PS-tree of the previous POI in level-order and transforming it to the PS-tree of next POI. This traversal involves: 1) the updating of sequences, 2) the deletion of obsolete elements, and 3) the insertion of new elements in the PS-tree. If a positive sequential candidate is certified to be a PSP while traversing PS-tree, we generate NSC based on the PSP and then check whether the NSC can be NSP. The algorithm is introduced step-by-step as follows.

Step 1: If there is new data at the current timestamp in the database, collect the elements from each sequence.

Step 2: Traverse PS-tree in level-order. For each node of PS-tree, process the node and update its children according to the type of node.

Step 2.1: If the node is a common node and the positive sequential candidate it represents is frequent, a) output the positive sequential candidate as a PSP, and b) generate NSC and output them as NSP if they are frequent after checking the support.

Step 3: Move POI forward. Repeat Steps 1 and 2.

The main procedure of the algorithm **Propone** is shown in Algorithm 1.

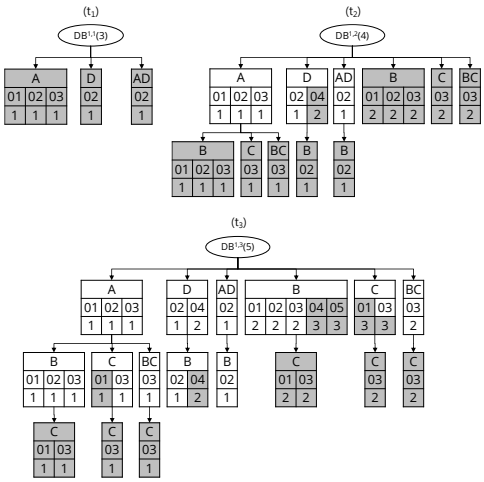


Fig. 3. PS-tree of the example database (t_1 to t_3).

4.2 PS-tree Traversal

In this section, we illustrate how to update and maintain the PS-tree. The algorithm **traverse** is shown in Algorithm 2, and Fig. 3 presents the PS-tree of the example database from t_1 to t_3. **Propone** traverses PS-tree in level-order. The root is processed first, then **Propone** goes to the nodes on the first l evel. O nce a ll t he n odes o n t he fi rst le vel are processed, we start with the nodes on the second level and so on. The main task of the algorithm **traverse** is to add the newly arriving elements to PS-tree while removing the obsolete elements. The nodes in the PS-tree can be divided into two types: the root and common nodes. For the root, as shown in Fig. 3 (t_1), **Propone** takes each element collected at the current timestamp in *eleSet* and uses them to update the children under the root. In the example database shown in Fig. 1, there are sequences *S01*, *S02* and *S03* having elements (*A*), (*AD*) and (*A*), respectively. **Propone** then appends the elements to the children through the procedure UPDATE_CHILD, which is given in Procedure 1. The newly appearing elements in the *eleSet* are used for generating all combinations of candidate elements. For example, if the element is (*ABC*), we generate (*A*), (*B*), (*C*), (*AB*), (*AC*), (*BC*), and (*ABC*) as the candidate elements. For each candidate element, **Propone** checks whether there is already a child under the root with the same element as the label. If there is a child with the same element as the label, meaning that this element has appeared at the previous timestamps, **Propone** examines if the sequence id has been in the sequence list of the child. If so, then this means that the element occurred in the same sequence. Therefore, **Propone** updates the timestamp of that sequence with the given timestamp, which is *current_time* for the root. Otherwise, we create a new sequence in the sequence list with *current_time*. If no child under the root has the same label, **Propone** creates a new child with the candidate element as its label, the corresponding sequence id, and *current_time*. The newly created node is then also appended to the rear of the traversal queue of next level.

For common nodes, such as the node labeled *A* in Fig. 3 (t_2), **Propone** checks if the node is not a newly created one. The newly created nodes, *i.e.*, the gray nodes in Fig. 3, represent the newest data in the database. These nodes themselves have been processed in the earlier iterations, and there is no possibility that they have descendants. Therefore, we do not need to perform any operation on them except examining if there are some sequential patterns that need to be output. If the node is not newly created,

Algorithm 2 : traverse(*DB, min_sup, current_time, PST, POI, FRI*)

Require: A progressive database *DB, min_sup*, the current timestamp *current_time*, a PS-tree *PST*, *POI*, and *FRI*

Ensure: An updated PS-tree *PST*

1: *PSP-Hash* = ϕ; ▷ a hash table mapping from PSP to their supports or corresponding nodes
2: **for** each tree node, *p*, of *PST* in level-order **do**
3: **if** *p* is Root **then**
4: **for** *ele* of every *seq* in *eleSet* **do**
5: UPDATE_CHILD(*ele, p, seq, current_time*)
6: **else** ▷ the node is a common node
7: **if** *p* is NOT newly created **then**
8: **for** every *seq* except the newly created ones in *p.seq_list* **do**
9: **if** $seq.timestamp \leq current\_time - POI$ **then**
10: delete *seq* from *p.seq_list* and continue to the next *seq*;
11: **if** there is new *ele* of the *seq* in *eleSet* **then**
12: **if** *p.seq_list.seq* is updated **then**
13: **if** $seq.old\_timestamp \leq current\_time - POI$ **then**
14: continue to the next *seq*;
15: **else**
16: $ts = seq.old\_timestamp$;
17: **else**
18: $ts = seq.timestamp$;
19: UPDATE_CHILD(*ele, p, seq, ts*)
20: *noDesc* = |newly created *seq*| + |*seq* whose $old\_timestamp \leq current\_time - POI$|
21: **if** $p.seq\_list.size - noDesc == 0$ **then**
22: delete all *p*'s children from *p*;
23: **if** $noDesc == 0$ **then**
24: delete *p*;
25: **if** $p.seq\_list.size \geq min\_sup * |DB|$ **then**
26: **output_patterns**(*p, PSP-Hash*);

then **Propone** checks each sequence in the sequence list except the newly created ones. If the sequence is obsolete, it is removed from the sequence list. If the sequence is not obsolete and there is a new element of the sequence in *eleSet*, **Propone** checks further whether the sequence is updated. If the sequence is updated, **Propone** takes the previous timestamp, namely $old\_timestamp$, which is the timestamp before being updated. Note that there can be several ways to retrieve the $old\_timestamp$. In this work, we update a timestamp in a child node with the result of $old\_timestamp$ plus new timestamp times $current\_time$. While we are processing the child node, we can retrieve the $old\_timestamp$ and the correct new timestamp from the remainder and the quotient of the division of the result by $current\_time$, respectively. Another approach is storing the $old\_timestamp$ in tree nodes, although this would require more memory usage. If the $old\_timestamp$ of the sequence is obsolete, **Propone** continues to the next sequence because the same sequence in the children must be obsolete as well. Otherwise, **Propone** takes $old\_timestamp$ and executes UPDATE_CHILD. The reason is that in level-order the parent is processed first and the timestamp has been updated, but the new timestamp is incorrect for the candidate sequential patterns of the children. It is the old timestamp where the candidate sequential patterns of the children begin. If the sequence is not updated, which means the timestamp remains the same, **Propone** uses the timestamp of the sequence directly and updates the children. After all the sequences in the sequence list are processed, **Propone** calculates the

Procedure 1 UPDATE_CHILD(*ele, p, seq, time*)

Require: An element *ele*, a PS-tree node *p*, a sequence id *seq*, and a timestamp *time*
Ensure: Update all the children of node *p*

```
1: for each combination comb of elements in the ele do
2:     if comb == label of one of p.child then
3:         if seq is in p.child.seq_list then
4:             update timestamp of seq in p.child to time;
5:         else
6:             create a new sequence in p.child.seq_list with time;
7:     else
8:         create a new child with comb, seq and time;
9:         add this child to traversal queue of next level;
```

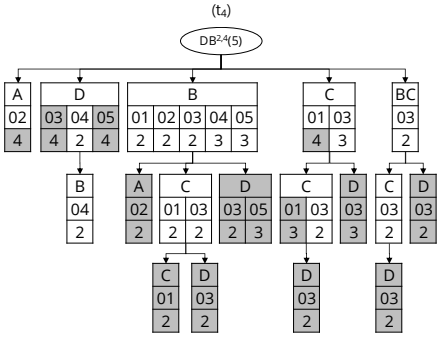


Fig. 4. PS-tree of the example database (t_2 to t_4).

number of sequences which cannot have any descendants, *noDesc*. The sequences having no descendants include those which are newly created and those whose $old\_timestamp$ is obsolete. The number *noDesc* is used to determine whether the subtree under the node, and even the node itself, should be pruned from PS-tree. If the number of sequences in the sequence list of the node minus *noDesc* is 0, it means that all the remaining sequences in the node are either newly created or possess an obsolete $old\_timestamp$. This indicates that every sequence in each descendant is obsolete, and thus **Propone** deletes all the children of the node from PS-tree. For example, the subtree of node A in the first level in Fig. 3 (t_3) is removed after the POI advances, which is shown in Fig. 4. Then, if *noDesc* is equal to 0, **Propone** removes the node from the PS-tree because there is no sequence left in the sequence list of the node. For instance, the node AD and its subtree in Fig. 3 (t_3) are discarded in Fig. 4. At the last step of processing a common node, **Propone** examines the support of a candidate sequential pattern. If the number of sequences in the sequence list of a common node is greater than or equal to the minimum support *min_sup* times the number of sequences in the current POI, $|DB|$, a procedure named **output_patterns** is executed to output the corresponding PSP and to discover potential NSP.

4.3 Outputting PSP and NSP

The output of PSP and NSP is handled by the procedure **output_patterns**. Note that **output_patterns** requires a special data structure named *PSP-Hash*. *PSP-Hash* is a hash table which maps from PSP to either their supports or corresponding PS-tree nodes. Fig. 5 shows the *PSP-Hash* of the example PS-tree in Fig. 2. The keys of *PSP-Hash* are the PSP mined in each traversal. The values are the supports of the PSP if the size of the PSP is 1, or a link to the corresponding PS-tree node that represents the PSP if the size

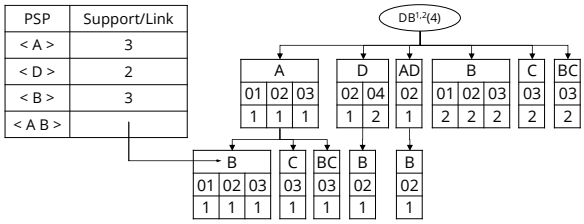


Fig. 5. PSP-Hash of the example PS-tree in Fig. 2.

Algorithm 3 : output_patterns(*p, PSP-Hash*)

Require: A PS-tree node p and a map from PSP to their supports or corresponding nodes *PSP-Hash*

Ensure: Output the PSP and potential NSP

1: *PSP* = labels of path from Root to p;
2: $neg\_ele = \phi$; ▷ a set of necessarily negative elements
3: **if** $PSP.size == 1$ **then**
4: *PSP-Hash.put*(*PSP*, $sup(PSP)$);
5: **else**
6: *PSP-Hash.put*(*PSP*, pointer to p);
7: **for** each element e of *PSP* **do**
8: **if** $sup(e) - sup(PSP) < sup(PSP) * FRI$ **then**
9: insert e into *neg_ele*;
10: **if** all the elements in *neg_ele* are not contiguous in *PSP* **then**
11: **for** each *NSC* generated based on *neg_ele* **do**
12: **if** *NSC* contains more than 1 negative element **then**
13: **if** $sup(MPS(NSC)) - sup(PSP) < sup(PSP) * FRI$ **then** ▷ Avoid support checking
14: continue to the next *NSC*;
15: **if** $sup(NSC) \geq sup(PSP) * FRI$ **then**
16: output *NSC* as an *NSP*;

of the PSP is larger than 1. The reason for storing the link to the corresponding node is that **Propone** can retrieve the sequence id set of a PSP quickly. We can use *PSP-Hash* to speed up the calculation of the support of NSP because the formulas for this involves the union of sequence id sets. For 1-size PSP, only the support is stored because the formulas do not require the calculation of union for such PSP.

To output NSP, we need to use an identified PSP mined during the PS-tree traversal to generate Negative Sequential Candidates (NSC), and then check their supports to determine whether they are NSP. The approach used for generating NSC follows the method in [15]. NSC are generated by changing any non-contiguous elements in a PSP to their corresponding negative ones. The definition of NSC Generation is given as below.

Definition 6 ***Negative Sequential Candidate Generation***
Given a k-size PSP, NSC are generated from the PSP by changing any m non-contiguous element(s) to the corresponding negative one(s), where $m = 1, 2, \ldots, \lceil \frac{k}{2} \rceil$.

Example 4 *Based on a PSP* $< A\ (BC)\ D >$*, the NSC generated are:*
$m = 1$, $< \neg A\ (BC)\ D >$, $< A\ \neg(BC)\ D >$, $< A\ (BC)\ \neg D >$;
$m = 2$, $< \neg A\ (BC)\ \neg D >$.

With a PSP mined during the PS-tree traversal, many NSC can be generated. However, we can utilize the information from the given PSP to reduce the number of NSC. We

propose a pruning strategy to discard infrequent NSC in advance, reducing the time spent on counting support. The pruning strategy is based on a property as follows.

Property 1 *Given a PSP s, a subsequence of s, s', an NSC n, and a FRI α, the NSC n is not frequent if all the following conditions hold: (1) $p(n) = s$; (2) n contains s'; (3) $sup(s') - sup(s) < sup(s) * \alpha$.*

Pruning Strategy 1 *Given a PSP s, an NSC n, and a FRI α, for each element e in s, let $< e >$ be a 1-size PSP containing only e. If $sup(< e >) - sup(s) < sup(s) * \alpha$, NSC which contain e, and whose positive partner is s, must be infrequent. Therefore, e should be negative while generating NSC.*

Example 5 *Given a PSP $s = < (AB)\ C\ D\ (EFG)\ H >$, we can obtain five 1-size PSP: $s_1 = < (AB) >$, $s_2 = < C >$, $s_3 = < D >$, $s_4 = < (EFG) >$, $s_5 = < H >$.*
1) Suppose s_1 and s_3 satisfy the inequality, then NSC containing (AB) or D must be infrequent. Thus, (AB) and D should be negative while generating NSC. According to this information, the NSC generated are:
neg-size $= 1 \Rightarrow \phi$
neg-size $= 2 \Rightarrow < \neg(AB)\ C\ \neg D\ (EFG)\ H >$
neg-size $= 3 \Rightarrow < \neg(AB)\ C\ \neg D\ (EFG)\ \neg H >$.
The number of NSC is reduced from 12 (5, 6, 1 for neg-size 1, 2, 3 NSC respectively) to 3.
2) Suppose s_1 and s_2 satisfy the inequality, then NSC containing (AB) or C must be infrequent. Thus, (AB) and C should be negative while generating NSC. According to this information, NO NSC are generated because there must be two contiguous negative elements in the NSC.

By taking advantage of Property 1, the pruning strategy examines what elements in a PSP should be negative while we are generating NSC, and thus can avoid producing infrequent NSC. However, there may be few infrequent NSC being generated after the pruning. Before the supports of these infrequent NSC are calculated, which takes some time, we can leverage Property 1 again to skip the calculation. We propose a simple technique, named *avoiding support checking*, to avoid the calculation of support of the infrequent NSC not pruned by the pruning strategy, as described below.

Avoiding Support Checking 1 *Given an NSC n and a FRI α, suppose n is not pruned by the pruning strategy. If $sup(MPS(n)) - sup(p(n)) < sup(p(n)) * \alpha$, then n is not frequent. Therefore, we skip the support checking for n.*

With the techniques presented above, **Propone** can deal with the output of NSP once it discovers a PSP during the PS-tree traversal. The detailed steps of the procedure **output_patterns** are shown in Algorithm 3. As per the PSP, *PSP-Hash* is updated. Then, NSC are generated with the utilization of pruning strategy and avoiding support counting strategy discussed above. Finally, support is calculated to output NSP according to FRI.

5. PERFORMANCE EVALUATION

In this section, a series of experiments are conducted to evaluate the performance of the **Propone** algorithm with different parameters. Since there are still no works which find PSP and NSP at the same time in a progressive database, we modify GSP [2] and Pisa [10] to suit the purpose of this study and for comparison purposes. We let GSP

Table 1. Parameters for synthetic data generation.

Symbol	Meaning
D	Number of sequences
C	Average number of elements per sequence
N	Number of different items
T	Total number of timestamps

statically remine the database in each POI, and combine this with the same approach to mining NSP as in e-NSP [15], *i.e.*, the approach also adopted by **Propone**, but without our proposed pruning strategy, with both of the modified GSP and Pisa. The two comparative algorithms are named GSP_PN and Pisa_PN, respectively. For the testing datasets, we use the IBM data generator [1] to generate synthetic datasets and then transform them into the desired format of a progressive database. A real dataset from Netflix, which was used in the 2007 KDD Cup [35], is also included in the experiments. GSP_PN, Pisa_PN and **Propone** are all implemented in Python 2.7.9, and the experiments are performed on a computer with Intel Core i5 3.4-GHz CPU and 8GB memory.

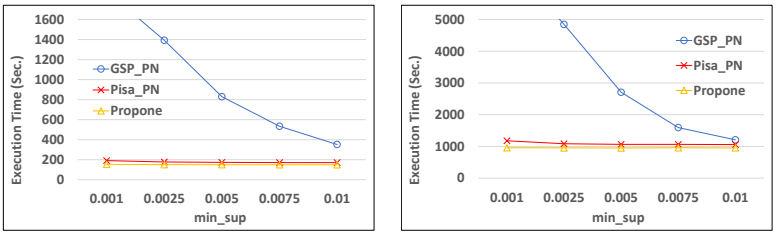
5.1 Experimental Setup

The synthetic datasets are generated by the IBM data generator and transformed into the format of a progressive database. There are several adjustable parameters in the IBM data generator, such as the number of sequences and the average length of sequences, so that various datasets can be generated to meet different needs. Moreover, we include a real dataset employed by the 2007 KDD Cup, the Netflix Prize training dataset, which consists of 480189 users, 17770 movies, and over 100 million rating records.

Unlike an incremental database, which only appends new data to the end of the sequences, a progressive database has to further remove the obsolete data. For this reason, each element in the sequences should be assigned a timestamp after the datasets are generated by the IBM data generator. Once each element is given a timestamp, we divide the whole generated dataset into m timestamps. Suppose the length of POI is set as n ($n < m$), then every n timestamp is seen as a subset to be processed. Note that m should be larger than the maximal number of elements in a sequence.

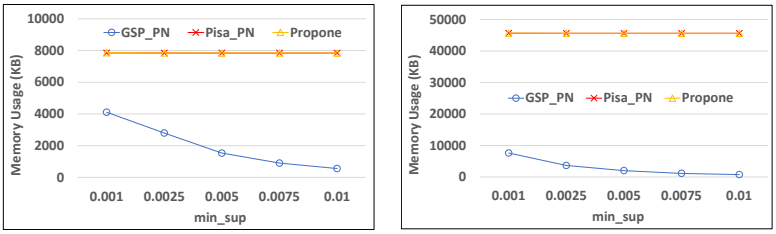
The major parameters used in our experiments, as shown in Table 1, are: number of sequences (D), average number of elements per sequence (C), and number of different items (N). Another parameter not provided by the IBM data generator is the total number of timestamps (T). This parameter indicates how many timestamps the datasets should be divided into. The total number of timestamps (T) is fixed at 40, which is the same setting adopted in [10]. For the average number of elements per sequence (C), we choose two values to generate two kinds of datasets with different density, so that we can observe their effects. Because the total number of timestamps (T) is 40, we set one C as 20, which is a half of 40, and the other one as 25, which is a higher value without generating sequences longer than 40. For the number of different items (N), we set the value as 10K.

To shape the real dataset into the same format as the synthetic datasets, we follow the approach that is taken in [10] to obtain an equal comparison environment. We randomly extract 120 successive days from the Netflix dataset and set three days as a timestamp, gathering the items in every three-day interval as an element. In this way there are 40 timestamps in the dataset, the same as the total number of timestamps in the synthetic datasets. The resulting dataset contains 46,871 sequences and 10,102 items.



(a) sparser (C = 20) (b) denser (C = 25)

Fig. 6. Execution time with different minimum supports on the sparser and denser synthetic dataset.



(a) sparser (C = 20) (b) denser (C = 25)

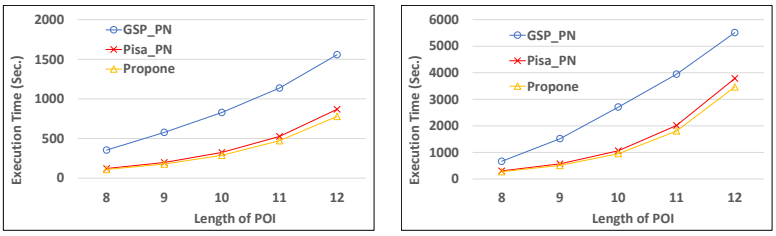
Fig. 7. Memory usage with different minimum supports on the sparser and denser synthetic dataset.

5.2 Different Minimum Support Thresholds

First we perform the experiments on different minimum support thresholds. Fig. 6a and Fig. 6b show the execution time of the three algorithms over the minimum supports, ranging from 0.001 to 0.01, on the two synthetic datasets, which are the sparser one (C = 20) and the denser one (C = 25). The number of sequences (D) is 20K. The POI and FRI both are set at 10. When the minimum support decreases, the execution time of all three algorithms increases, because there will be more sequential patterns to be processed. However, the execution time of GSP_PN increases significantly, while there is only a slight increase in those of Pisa_PN and **Propone**. This is because GSP_PN has to scan the subdatabases many times, while Pisa_PN and **Propone** only need to process the newly arriving elements. Furthermore, **Propone** takes less execution time than Pisa_PN because the level-order traversal skips the redundant processing of the nodes which need to be deleted, and the pruning strategy discards many infrequent NSC. In addition, with the denser dataset the difference in execution time between Pisa_PN and **Propone** is larger when the minimum support is small. Fig. 7a and Fig. 7b show the memory usage for the two datasets. With regard to the memory usage, GSP_PN requires less because it rescans the database rather than stores the information of all the sequential patterns. Although Pisa_PN and **Propone** use more memory, the maximum memory usage is not too large and the execution time is much less than GSP_PN's, as shown in Figs. 6a and 6b. **Propone** needs more memory than Pisa_PN due to the utilization of the hash table data structure, *PSP-Hash*, but the difference is very small. Both Pisa_PN and **Propone** require much more memory than GSP_PN in the denser dataset, because more elements appearing in the POI leads to an increase in the number of candidate sequences.

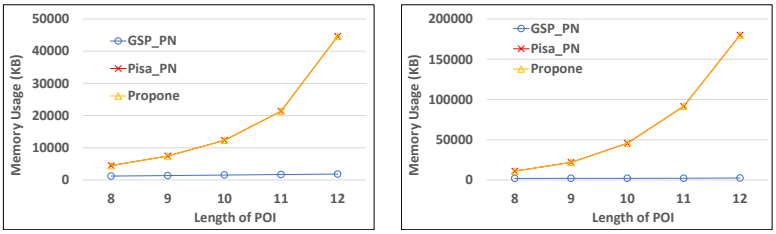
5.3 Different Length of POI

We then examine the performance when the length of the POI is varying. The length of the POI ranges from 8 to 12. The number of sequences (D) is 20K. The minimum support and FRI are set at 0.005 and 10 respectively. As shown in Figs. 8 (a) and (b), the execution time of all the algorithms increases as the POI becomes longer. GSP_PN has to



(a) sparser (C = 20) (b) denser (C = 25)

Fig. 8. Execution time with different lengths of POI on the sparser and denser synthetic dataset.



(a) sparser (C = 20) (b) denser (C = 25)

Fig. 9. Memory usage with different lengths of POI on the sparser and denser synthetic dataset.

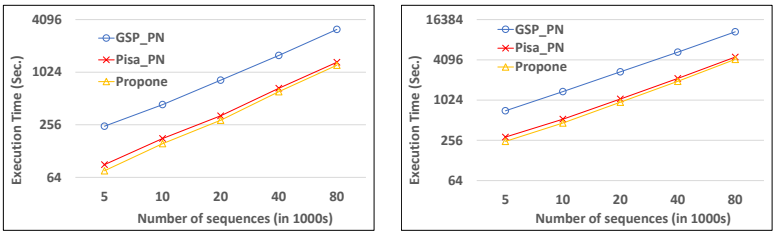
deal with more elements in the sequences or even more sequences as POI grows longer. The number of levels of the PS-tree also depends on the length of the POI. A longer POI will lead to a larger PS-tree for Pisa_PN and **Propone** to traverse, and thus more execution time will be needed. However, the difference between GSP_PN and the two PS-tree based algorithms is still significant, and the execution time of **Propone** increases slower than that of Pisa_PN. Furthermore, by comparing Figs. 8 (a) and (b) we can see that in the denser dataset the execution time of the two PS-tree based algorithms grows a little faster than that in the sparser one. Figs. 9 (a) and (b) show that the memory usage of Pisa_PN and **Propone** grows faster than that of GSP_PN. This is because when the POI is longer, there are more elements lying in the POI, and the PS-tree will thus have to generate more branches to store the candidate sequences.

5.4 Different Number of Sequences

To observe the scalability of the three algorithms, we generate several synthetic datasets with different number of sequences (D). There are five datasets with 5K, 10K, 20K, 40K, and 80K sequences for each kind of densities. The minimum support is set at 0.005. The POI and FRI both are set at 10. Figs. 11 (a) and (b) show that the execution time of all algorithms becomes longer as the number of sequences (D) increases, but Pisa_PN and **Propone** take much less time than GSP_PN to deal with extra sequences. The reason is that when the size of the database increases, GSP_PN needs to perform multiple rescans on more sequences, but Pisa_PN and **Propone** only have to deal with the candidate sequences in the PS-tree, whose quantity is smaller than that of additional sequences in the database. However, there is an overhead that Pisa_PN and **Propone** utilize more memory as the number of sequences (D) increases, as shown in Figs. 11 (a) and (b), because the sequence id list in each PS-tree node will contain more sequence entries.

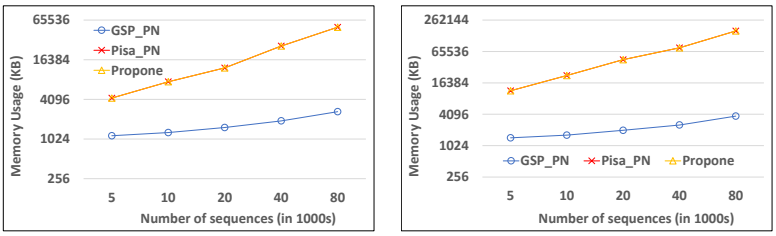
5.5 Analyses of FRI

In this section, we analyze the performance with different FRI and the pruning effect of the pruning strategy. FRI is a ratio between the support of an NSP and the support of



(a) sparser (C = 20) (b) denser (C = 25)

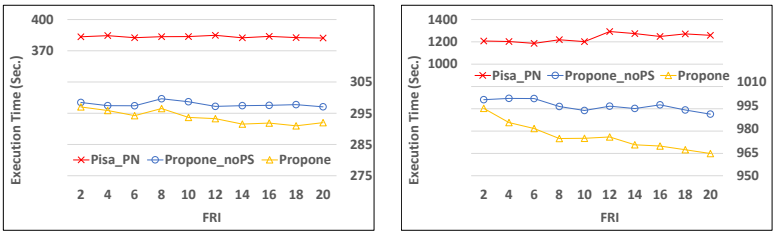
Fig. 10. Execution time with different no. of sequences on the sparser and denser synthetic dataset.



(a) sparser (C = 20) (b) denser (C = 25)

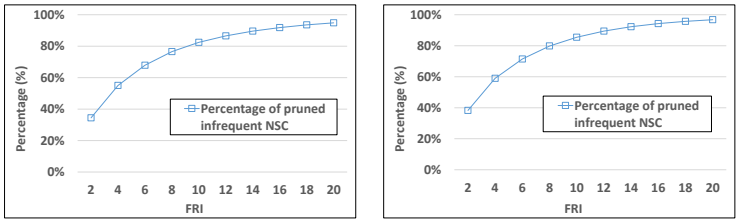
Fig. 11. Memory usage with different no. of sequences on the sparser and denser synthetic dataset.

the NSP's corresponding PSP, *i.e.*, the positive partner of the NSP. It is used to extract those NSP which are overwhelming against their positive partner in support, giving new applications to NSP. When FRI gets larger, the condition with regard to the NSP is stricter, and thus the number of NSP will be reduced. Once less NSC are not able to become NSP and thus can be pruned before they are processed, the execution time of the algorithm will decrease. In this experiment, the number of sequences (D) is 20K, and the length of the POI is 10. We set the minimum support as 0.0005, which is smaller than in the previous experiments, to induce more PSP and also more NSP. Furthermore, we implement another algorithm, called Propone_noPS, which is obtained by removing the pruning strategy from **Propone**, to observe the effects of doing this. In addition, GSP_PN is dropped in this experiment because of its considerable execution time. Figs. 12 (a) and (b) show the execution time of the three algorithms with different FRI ranging from 2 to 20. From Figs. 12 (a) and (b), we can see that the execution time of Pisa_PN and Propone_noPS do not change across different FRI, because Pisa_PN and Propone_noPS lack pruning strategies. On the other hand, the execution time of **Propone** deceases as FRI becomes larger. This is because a larger FRI results in the reduction in the number of NSP, and thus there are more potential infrequent NSC to be pruned. We can also see that the difference in execution time between Propone_noPS and **Propone** is larger in the denser dataset than in the sparser one, because there are more PSP, and thus more NSC will be generated, increasing the space of reduction. To investigate the effectiveness of the pruning strategy, an experiment on the percentage of pruned infrequent NSC is conducted, with the results shown in Figs. 13 (a) and (b). The trend shows that the pruning strategy is more effective when FRI is larger and the percentage of pruned infrequent NSC rises quickly. About 80% of the infrequent NSC can be pruned when FRI is at 10, and 90% can be pruned when FRI equals 14. The percentages are even higher in the results of the denser dataset. Therefore, the pruning strategy is effective in reducing the number of the infrequent NSC to speed up the mining process.



(a) sparser (C = 20) (b) denser (C = 25)

Fig. 12. Execution time with different FRI on the sparser and denser synthetic dataset.



(a) sparser (C = 20) (b) denser (C = 25)

Fig. 13. Pruning effect with different FRI on the sparser and denser synthetic dataset.

5.6 Number of Patterns

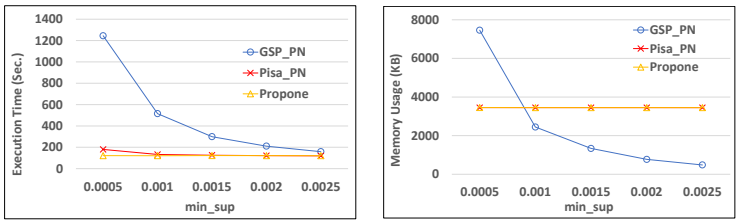
To understand the densities of the two datasets, we give an insight into the number of PSP, NSC, and NSP which are mined or generated during the mining processes. Table 2 shows the average number of PSP, NSC, and NSP in each run (there are 31 runs in each experiment) in the two datasets, with the minimum supports ranging from 0.001 to 0.01. The number of sequences (D) is 20K. The length of the POI is 10, and FRI is set as 10. As we can see in Table 2, there are more PSP in the denser dataset, because the data sequences are longer in the POI, and thus longer patterns could appear. Due to the increase in the number and length of PSP, the number of NSC also becomes more, along with that of NSP. Moreover, a smaller minimum support allows more and longer PSP, and a longer PSP can generate more NSC. This is the reason that the number of NSC grows fast as the minimum support becomes smaller. The columns of NSP show that the numbers of NSP are all smaller than that of PSP, indicating that setting FRI as 10 is strict to NSP.

5.7 Test on Real Data

Finally, we conduct the experiments on the real dataset to study the practicability of **Propone** algorithm. To show the effect of varying minimum support, we executed the experiment with the minimum support set at 0.0005 to 0.0025. The POI and FRI are set at

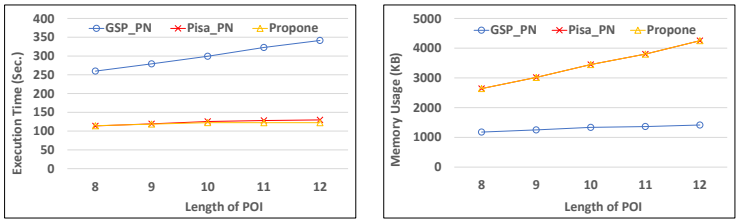
Table 2. Average no. of patterns in the two datasets with different minimum supports.

min_sup	Sparse (C = 20)			Dense (C = 25)		
	PSP	NSC	NSP	PSP	NSC	NSP
0.001	2495.6	6110.3	1264.0	6098.0	21030.9	1984.1
0.0025	1013.6	1823.5	505.2	2015.1	4865.7	683.3
0.005	459.7	640.3	321.3	718.9	1151.8	437.9
0.0075	280.0	341.8	231.6	378.7	470.4	309.8
0.01	190.9	217.9	169.8	232.7	253.7	214.8



(a) Execution time (b) Memory usage

Fig. 14. Effect of varying minimum supports, POI 10 and FRI 15 on the real dataset.



(a) Execution time (b) Memory usage

Fig. 15. Effect of varying POI, minsup 0.0015 on the real dataset.

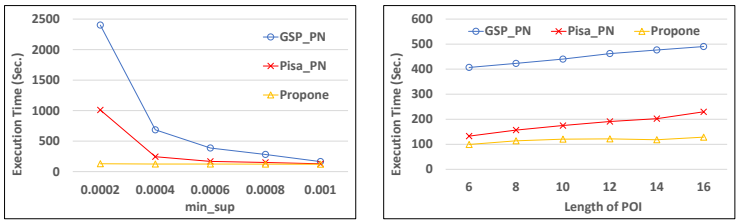
10 and 15 respectively. As shown in Fig. 14 (a), although the difference in execution time between Pisa_PN and **Propone** is not obvious with a larger minimum support, it is more significant when the minimum support gets s maller. This is because when the minimum support is small, the number of PSP increases, and thus leads to an increase in NSP. The pruning strategy and *PSP-Hash* help speed up the mining of NSP. The memory usage of GSP_PN becomes higher than those of Pisa_PN and **Propone** when the minimum support is at 0.0005, as shown in Fig. 14 (b), because the Netflix dataset is much sparser, which will not make the PS-tree too big, while GSP_PN still has to generate many candidate sequences.

In the experiment on different length of the POI, the minimum support is set at 0.0015 and FRI is set at 15. Fig. 15 (a) shows that the execution time of **Propone** remains nearly the same across various POI, while that of Pisa_PN keeps increasing slowly. The difference is small because the Netflix dataset is sparse, and increasing the length of the POI does not substantially make the sequences longer. The memory usage of all three algorithms, as shown in Fig. 15 (b), are growing, and Pisa_PN and **Propone** still require more memory than GSP_PN does.

We executed another set of experiments to show the effect of low minimum supports. Fig. 16 (a) shows the execution time for varying minimum supports. For this experiment, the minimum support ranges from 0.0002 to 0.0010. The POI and FRI are set at 10 and 1 respectively. Fig. 16 (b) shows the execution time for varying POIs. For this experiment, the POI ranges from 6 to 16. The minimum support and FRI are set at 0.0005 and 1 respectively.

6. CONCLUSIONS

NSP is useful but its definition varies among the previous works depending on the views of users and the field of a pplication. In this paper, we combine NSP mining with progressive sequential pattern mining by presenting an efficient algorithm, **Propone**. **Propone** traverses the PS-tree in level-order instead of post-order to reduce the unnecessary processes. We also give a new definition of NSP which requires that the support of an NSP



(a) Varying min. sup., POI 10 (b) Varying POI, min. sup. 0.0005

Fig. 16. Execution time showing effect of low minimum supports on the real dataset.

should be greater than or equal to the PSP's support times a parameter FRI. A pruning strategy is proposed to reduce the number of infrequent NSC in this paper. The experimental results show that **Propone** outperforms the comparative algorithms and is efficient in execution time without requiring too much memory. The experiments on FRI also show that the pruning strategy is effective in discarding the infrequent NSC.

REFERENCES

1. R. Agrawal and R. Srikant, "Mining sequential patterns," in *Proceedings of the 11th International Conference on Data Engineering*, 1995, pp. 3-14.
2. R. Agrawal and R. Srikant, "Mining sequential patterns: Generalizations and performance improvements," in *Proceedings of the 5th International Conference on Extending Database Technology*, 1996, pp. 1-17.
3. M. J. Zaki, "Spade: An efficient algorithm for mining frequent sequences," *Machine Learning*, Vol. 42, 2001, pp. 31-60.
4. 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 *Proceedings of the 17th International Conference on Data Engineering*, 2001, pp. 215-224.
5. J. Ayres, J. Flannick, J. Gehrke, and T. Yiu, "Sequential pattern mining using a bitmap representation," in *Proceedings of the 8th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, 2002, pp. 429-435.
6. S. Parthasarathy, M. J. Zaki, M. Ogihara, and S. Dwarkadas, "Incremental and interactive sequence mining," in *Proceedings of the 8th International Conference on Information and Knowledge Management*, 1999, pp. 251-258.
7. F. Masseglia, P. Poncelet, and M. Teisseire, "Incremental mining of sequential patterns in large databases," *Data & Knowledge Engineering*, Vol. 46, 2003, pp. 97-121.
8. H. Cheng, X. Yan, and J. Han, "Incspan: Incremental mining of sequential patterns in large database," in *Proceedings of the 10th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, 2004, pp. 527-532.
9. S. N. Nguyen, X. Sun, and M. E. Orlowska, "Improvements of incspan: Incremental mining of sequential patterns in large database," in *Proceedings of the 9th Pacific-Asia Conference on Knowledge Discovery and Data Mining*, 2005, pp. 442-451.
10. J.-W. Huang, C.-Y. Tseng, J.-C. Ou, and M.-S. Chen, "A general model for sequential pattern mining with a progressive database," *IEEE Transactions on Knowledge and Data Engineering*, Vol. 20, 2008, pp. 1153-1167.
11. Y. Zhao, H. Zhang, S. Wu, J. Pei, L. Cao, C. Zhang, and H. Bohlscheid, "Debt detection in social security by sequence classification using both positive and negative

patterns," in *Proceedings of European Conference on Machine Learning and Principles and Practice of Knowledge Discovery in Databases*, 2009, pp. 648-663.

12. S.-C. Hsueh, M.-Y. Lin, and C.-L. Chen, "Mining negative sequential patterns for e-commerce recommendations," in *Proceedings of IEEE Asia-Pacific Services Computing Conference*, 2008, pp. 1213-1218.
13. Z. Zheng, Y. Zhao, Z. Zuo, and L. Cao, "Negative-gsp: An efficient method for mining negative sequential patterns," in *Proceedings of the 8th Australasian Data Mining Conference*, Vol. 101, 2009, pp. 63-67.
14. Z. Zheng, Y. Zhao, Z. Zuo, and L. Cao, "An efficient ga-based algorithm for mining negative sequential patterns," in *Proceedings of the 14th Pacific-Asia Conference on Knowledge Discovery and Data Mining*, 2010, pp. 262-273.
15. X. Dong, Z. Zheng, L. Cao, Y. Zhao, C. Zhang, J. Li, W. Wei, and Y. Ou, "e-nsp: Efficient negative sequential pattern mining based on identified positive patterns without database rescanning," in *Proceedings of the 20th ACM International Conference on Information and Knowledge Management*, 2011, pp. 825-830.
16. R. Agrawal and R. Srikant, "Fast algorithms for mining association rules," in *Proceedings of the 20th International Conference on Very Large Data Bases*, 1994, pp. 487-499.
17. X. Yan, J. Han, and R. Afshar, "Clospan: Mining closed sequential patterns in large datasets," in *Proceedings of SIAM International Conference on Data Mining*, 2003, pp. 166-177.
18. J. Wang and J. Han, "Bide: Efficient mining of frequent closed sequences," in *Proceedings of the 20th International Conference on Data Engineering*, 2004, pp. 79-90.
19. A. Gomariz, M. Campos, R. Marin, and B. Goethals, "Clasp: An efficient algorithm for mining frequent closed sequences," in *Proceedings of the 17th Pacific-Asia Conference on Knowledge Discovery and Data Mining*, 2013, pp. 50-61.
20. C. Luo and S. M. Chung, "Efficient mining of maximal sequential patterns using multiple samples," in *Proceedings of SIAM International Conference on Data Mining*, 2005, pp. 415-426.
21. P. Fournier-Viger, C.-W. Wu, and V. S. Tseng, "Mining maximal sequential patterns without candidate maintenance," in *Proceedings of the 9th International Conference on Advanced Data Mining and Applications*, 2013, pp. 169-180.
22. M. N. Garofalakis, R. Rastogi, and K. Shim, "Spirit: Sequential pattern mining with regular expression constraints," in *Proceedings of the 25th International Conference on Very Large Data Bases*, 1999, pp. 223-234.
23. J.-Z. Ouh, P.-H. Wu, and M.-S. Chen, "Experimental results on a constraint based sequential pattern mining for telecommunication alarm data," in *Proceedings of the 2nd International Conference on Web Information Systems Engineering*, Vol. 2, 2001, pp. 186-193.
24. I. Miliaraki, K. Berberich, R. Gemulla, and S. Zoupanos, "Mind the gap: Large-scale frequent sequence mining," in *Proceedings of ACM SIGMOD International Conference on Management of Data*, 2013, pp. 797-808.
25. A. Marascu and F. Masseglia, "Mining sequential patterns from data streams: a centroid approach," *Journal of Intelligent Information Systems*, Vol. 27, 2006, pp. 291-307.
26. A. Koper and H. S. Nguyen, "Sequential pattern mining from stream data," in *Proceedings of the 7th International Conference on Advanced Data Mining and Applications*, 2011, pp. 278-291.

27. S.-Y. Wu and Y.-L. Chen, "Mining nonambiguous temporal patterns for interval-based events," *IEEE Transactions on Knowledge and Data Engineering*, Vol. 19, 2007, pp. 742-758.
28. Y.-C. Chen, J.-C. Jiang, W.-C. Peng, and S.-Y. Lee, "An efficient algorithm for mining time interval-based patterns in large database," in *Proceedings of the 19th ACM International Conference on Information and Knowledge Management*, 2010, pp. 49-58.
29. K.-Y. Chen, B. P. Jaysawal, J.-W. Huang, and Y.-B. Wu, "Mining frequent time interval-based event with duration patterns from temporal database," in *Proceedings of International Conference on Data Science and Advanced Analytics*, 2014, pp. 548-554.
30. N. P. Lin, H.-J. Chen, and W.-H. Hao, "Mining negative sequential patterns," in *Proceedings of the 6th WSEAS International Conference on Applied Computer Science*, 2007, pp. 654-658.
31. N. P. Lin, H.-J. Chen, W.-H. Hao, H.-E. Chueh, and C.-I. Chang, "Mining strong positive and negative sequential patterns," *WSEAS Transactions on Computers*, Vol. 7, 2008, pp. 119-124.
32. W.-M. Ouyang and Q.-H. Huang, "Mining negative sequential patterns in transaction databases," in *Proceedings of the 2007 International Conference on Machine Learning and Cybernetics*, 2007, pp. 830-834.
33. Y. Zhao, H. Zhang, L. Cao, C. Zhang, and H. Bohlscheid, "Mining both positive and negative impact-oriented sequential rules from transactional data," in *Proceedings of the 13th Pacific-Asia Conference on Knowledge Discovery and Data Mining*, 2009, pp. 656-663.
34. W. Ouyang and Q. Huang, "Mining positive and negative sequential patterns with multiple minimum supports in large transaction databases," in *Proceedings of the 2nd WRI Global Congress on Intelligent Systems*, 2010, pp. 190-193.
35. "Kdd cup 2007," https://www.cs.uic.edu/ liub/Netflix-KDD-Cup-2007.html.

Jen-Wei Huang received the BS and Ph.D. degrees in Electrical Engineering from National Taiwan University, Taiwan in 2002 and 2009 respectively. He was a visiting scholar in IBM Almaden Research Center from 2008 to 2009, an Assistant Professor in Yuan Ze University from 2009 to 2012, and a visiting scholar in. University of Chicago in 2016. He is now an Assistant Professor in the Department of Electrical Engineering, National Cheng Kung University, Taiwan. He majors in computer science and is familiar with data mining. His research interests include data mining, mobile computing, and bioinformatics. Among these, social network analysis, spatial-temporal data mining, and multimedia information retrieval are his special interests. In addition, some of his research is on data broadcasting, privacy preserving data mining, e-learning and bioinformatics.

Yong-Bin Wu received the BS degree in Mathematics from National Cheng Kung University, Taiwan in 2013. He is now working towards the MS degree in the Institute of Computer and Communication Engineering, National Cheng Kung University, Taiwan. His research interests include positive and negative sequential pattern mining, progressive mining, and mining time series.

Bijay Prasad Jaysawal received the MS degree in Computer Science and Engineering from Yuan Ze University, Taiwan in 2013. He is currently working towards the Ph.D. degree in the Institute of Computer and Communication Engineering, National Cheng Kung University, Taiwan. His research interests include sequential pattern mining, high utility pattern mining, mining data streams, mining time interval-based events data, and mining time series.