Mining Local Periodic Patterns in a Discrete Sequence

Philippe Fournier-Viger[a,*], Peng Yang[b], Rage Uday Kiran[c], Sebastian Ventura[d], José María Luna[d]

[a]*School of Humanities and Social Sciences, Harbin Institute of Technology (Shenzhen), China*
[b]*School of Computer Science and Technology, Harbin Institute of Technology (Shenzhen), China*
[c]*Institute of Industrial Science, The University of Tokyo, Tokyo, Japan*
[d]*Department of Computer Science and Numerical Analysis, University of Córdoba, Cordoba, Spain*

Abstract

Periodic frequent patterns are sets of events or items that periodically appear in a sequence of events or transactions. Many algorithms have been designed to identify periodic frequent patterns in data. However, most assume that the periodic behavior of a pattern does not change much over time. To address this limitation, this paper proposes to discover a novel type of periodic patterns in a sequence of events or transactions, called Local Periodic Patterns (LPPs) which are patterns (sets of events) that have a periodic behavior in some non predefined time-intervals. A pattern is said to be a local periodic pattern if it appears regularly and continuously in some time-interval(s). Two novel measures are proposed to assess the periodicity and frequency of patterns in time-intervals. The *maxSoPer* (*maximal period of spillovers*) measure allows detecting time-intervals of variable lengths where a pattern is continuously periodic, while the *minDur* (*minimal duration*) measure ensures that those time-intervals have a minimum duration. To discover all LPPs, the paper presents three efficient algorithms. An experimental evaluation on real datasets shows that the proposed algorithms are efficient and can provide useful patterns that cannot be found using traditional periodic pattern mining algorithms.

Keywords: periodic pattern, itemset, time-interval, periodicity, local pattern, sequence

1. Introduction

Frequent itemset mining (FIM) [1, 11, 27, 30, 40] is a popular data analysis task. The goal is to identify all patterns (sets of values) that frequently appear in records of a transactional database. A pattern is said to be frequent if its support (occurrence frequency) is no less than a minimum support threshold. FIM has several applications and has inspired or has been used in many other data mining tasks including association rule mining [2, 18], sequential pattern mining [3, 10, 31], clustering [42] and classification [5, 41]. Although discovering frequent itemsets is useful, too many frequent patterns are often found, and many of them are uninteresting to users. Hence, it can be time-consuming to analyze frequent itemsets. To address these issues, several variations of FIM have been developed to select small sets of patterns that are interesting to users when considering various constraints. This includes discovering maximal frequent patterns [16], closed frequent patterns [29], high-utility patterns [12, 25], and emerging patterns [4].

Recently, identifying periodic patterns in data has become a popular research problem. Periodic frequent patterns (PFPs) [22, 36] are patterns that regularly appear in a database, that is patterns that have no time gap between two consecutive occurrences that exceeds a maximum gap size. Finding PFPs has many practical applications in various fields, such as analyzing stock market data to identify periodic stock movements, analyzing website clickstreams, improving the performance of recommender systems [34] and finding

*Corresponding author
Email addresses: philfv@hit.edu.cn (Philippe Fournier-Viger), pengyeung@163.com (Peng Yang), uday_rage@tkl.iis.u-tokyo.ac.jp (Rage Uday Kiran), sventura@uco.es (Sebastian Ventura), jmluna@uco.es (José María Luna)

frequently correlated genes in DNA sequences [43]. Another important application of PFPs is to study the purchase behavior of customers by discovering sets of products that are periodically bought, for the purpose of marketing and inventory management [8, 36].

However, traditional studies on periodic frequent pattern mining have two key limitations. The first one is that most algorithms identify periodic patterns by measuring the number of records (transactions) between occurrences of a pattern rather than relying on the actual timestamps of transactions. As a result, consecutive transactions that are one second apart are treated in the same way as transactions that are several hours apart, which is unrealistic. Recently, a few papers have adapted traditional periodic pattern mining to consider temporal databases (discrete sequences) where transactions have timestamps [9, 21, 23, 24, 33]. However, these studies and previous ones suffer from a second limitation, which is that the periodicity of patterns is assessed using a very strict measure called the maximum periodicity. These studies assume that a periodic pattern must have a periodicity below that threshold to be periodic. In fact, in traditional PFP mining, a pattern is deemed periodic only if *all* of its periods (intervals between consecutive occurrences of the pattern) are smaller than the maximum periodicity threshold. This is a problem because in real-life data, there can be a lot of variability between periods of a pattern. But if a pattern has a single period of length greater than the maximum threshold, the pattern is discarded. As a result, traditional periodic pattern mining algorithms are too strict and unable to find patterns that are only periodic in some specific time-intervals rather than in the whole database.

Discovering patterns that are only periodic in some non-predefined time-intervals is desirable for real applications as patterns rarely maintain a perfectly stable periodic behavior over time. For example, for market basket analysis, some products have seasonal trends that influence their periodic behavior. For instance, lychee is mainly sold during the summer. Thus, the time-intervals between occurrences of items such as lychee vary over time, and may be periodic only in some specific time-intervals such as during the first months of the summer. If traditional periodic pattern mining algorithms are applied on such data, they will discard items such as lychee and ignore that their sales are periodic in some time-intervals. Thus, important information will not be presented to the user about these time-intervals. To address these limitations, it is desirable to design a novel model for identifying local periodic patterns that can consider timestamps and is flexible by finding periodic patterns in time-intervals. There are three challenges:

- First, traditional models for identifying PFPs [9, 21, 22, 24, 33, 35, 36] are designed to find patterns that are periodic in a whole database, and they rely on the strict maximum periodicity model. It is not trivial to extend these traditional models to find specific time-intervals where a pattern is periodic. Recently, a study [23] has extended the traditional PFP model to find consecutive periods where a pattern is periodic but it still relies on the strict maximum periodicity constraint. Thus this model can discard a set of consecutive periods because a single period exceeds the threshold. Avoiding using that very strict definition of periodicity to find patterns that are locally periodic, requires to design novel interestingness measure(s) that consider the changing periodic behavior of patterns.

- Second, identifying a non predefined time-interval where a pattern is periodic requires to find its start point and end point. Criteria must thus be proposed to detect these points for each time-interval. Traditional change detection methods used in pattern mining are mainly based on identifying a single point of change rather than a time-interval, and do not consider periodicity [17, 38]. It is thus challenging to detect non predefined time-intervals where a pattern is periodic in a sequence of transactions (events).

- The third challenge is how to efficiently find the desired patterns while avoiding considering a large number of candidate patterns. Effective search space pruning technique(s) should thus be designed to avoid considering unpromising patterns, and time-intervals where they are not periodic. If a large number of candidate itemsets or time-intervals are considered, it may result in long execution times and huge memory consumption.

This paper addresses the above challenges. The contributions of this paper are fourfold:

- A new type of periodic patterns is proposed called *local periodic patterns*. A local periodic pattern is a pattern that exhibits a periodic behavior in some non-predefined time-interval(s) but not necessarily in the whole database. The novel problem of discovering these patterns can be viewed as an extension of traditional periodic pattern mining where real timestamps are used to select patterns. A key difference with traditional PFP mining is that the proposed model considers the time elapsed between consecutive transactions rather than the number of records between them. Note that because the latter is a special case of the former, the proposed model can also be applied on data that does not have timestamps by considering that transaction identifiers are the timestamps.

- Two novel measures are proposed to assess the periodicity and frequency of patterns in time-intervals. The *maxSoPer* (*maximal period of spillovers*) measure is used to detect time-intervals of variable lengths where a pattern is periodic and where the periodic behavior may vary, while the *minDur* (*minimal duration*) threshold is used to ensure that these time-intervals have a minimal duration.

- Three efficient algorithms named $LPPM_{breadth}$, $LPPM_{depth}$ and *LPP-Growth* are proposed. They respectively adopt a breadth-first search, depth-first search and pattern-growth approach by extending the *Apriori-TID* [1], *Eclat* [40] and *FP-Growth* [19] algorithms. The first two alorithms adopt a binary database representation, while the last one compresses the database using a compact tree structure to discover the complete set of local periodic patterns. The algorithms rely on novel techniques for reducing the search space to ensure that local periodic patterns are efficiently found.

- Multiple experiments have been carried out to evaluate the proposed pattern type and algorithms. Results show that the three algorithms are efficient, perform better in different situations, and that they can discover useful patterns in real-life retail sale data.

The rest of the paper is organized as follows. Section 2 surveys related work. Section 3 defines the proposed concept of local periodic pattern and the two novel *maxSoPer* and *minDur* measures. Section 4 presents the three proposed algorithms to find all local periodic patterns in a temporal database (a discrete sequence). Section 5 reports experimental results. Finally, Section 6 draws the conclusion and discusses future work.

2. Related Work

This section is divided into two subsections. The first one introduces the problem of frequent itemset mining and popular techniques for this problem. Then, the second one discusses more specifically how frequent pattern mining has been adapted to discover periodic patterns and key limitations of current techniques that are addressed in this paper.

2.1. Frequent Itemset Mining

Frequent itemset mining was initially proposed for market basket analysis to find sets of items (itemsets) that are frequently purchased by customers [1]. Then, FIM has been applied in numerous other applications where data can be represented using binary attributes. The goal of FIM is to find frequently occurring sets of values in database records [1, 27, 30, 40]. Because frequent itemset mining is a difficult problem, multiple algorithms have been proposed to efficiently enumerate all frequent patterns, including Apriori [1], FP-Growth [19], H-Mine [30], LCM [27] and Eclat [40]. These algorithms take as input a transaction database and a minimum support (frequency) threshold, and output the set of all patterns having an occurrence count greater than or equal to that threshold. These algorithms utilize various strategies to explore the search space of patterns and apply several optimizations to decrease the runtime and memory consumption of frequent pattern mining.

The first frequent itemset mining algorithm, called Apriori [1], adopts a breadth-first search. It first scans the database to calculate the support (frequency) of all items and selects those that are frequent. Then, Apriori combines those frequent items to generate candidate itemsets having two items. The support of these itemsets is then calculated by scanning the database to filter out the infrequent itemsets. Then,

Apriori generates patterns having three items by combining pairs of itemsets having a single item, and this process is repeated to find larger itemsets until no more frequent itemsets can be generated. Apriori applies two strategies to reduce the search space. First, as explained above, Apriori only combines two patterns if they are frequent to avoid processing some infrequent patterns. Second, Apriori applies another pruning property, which is that if a candidate itemset contains k items and has a subset of size $k-1$ that is infrequent, then this candidate can be ignored because it will be infrequent. The main drawbacks of Apriori from a performance perspective are that (1) Apriori repeatedly reads the database, which can be time-consuming, and that (2) Apriori can generate patterns that do not exist in the database [11].

To reduce the cost of database scans to find the support of patterns, the Apriori-TID [1] algorithm adopts a vertical database representation. The Apriori-TID algorithm first scans the database to identify all frequent items. Each item is associated to the list of transactions where it appears. Then, Apriori-TID performs the same breadth-first search as Apriori. But when Apriori-TID combines two itemsets X and Y to generate a larger itemset Z, Apriori-TID calculates the list of transactions of Z by intersecting the list of transactions of X and Y. Apriori-TID then derives the support of Z directly from its list of transactions, without scanning the database. For very large databases or databases where items appear in few transactions, this technique can greatly reduce the runtime compared to Apriori [11]. However, this approach has the drawback of Apriori that storing list of transactions for each itemset may consume a large amount of memory. Besides, Apriori-TID still generates candidates by combining itemsets, which can result in generating patterns that do not exist in the database. Another popular algorithm named Eclat [40] adopts a similar approach but performs a depth-first search rather than a breadth-first search. Eclat first identifies frequent items and create their transaction lists, and then combine pairs of items to generate itemsets having two items. While doing this, all items in itemsets are sorted according to a total order and itemsets that have the same prefix are grouped together in a same equivalence class. Then, Eclat processes each equivalence class separately. It combines all pairs of itemsets from an equivalence class to find frequent itemsets and generate other equivalence classes. Then, the remaining equivalence classes are explored in the same way using a depth-first search to find all frequent itemsets. An advantage of Eclat over Apriori-TID is that Eclat generally keeps less itemsets in memory at the same time because it adopts a depth-first search. However, Eclat can still generate candidate itemsets that do not exist in the database.

Other algorithms such as FP-Growth [19], H-Mine [30], and LCM [27] adopt a *pattern-growth* approach to reduce the cost of database scans. They first scan the database to find frequent items. Then, for each frequent item, the algorithms perform a depth-first search to find larger itemsets. To reduce the cost of database scans, a technique called database projection is applied that consists of ignoring all transactions where an itemset does not appear, as well as parts of transactions that are irrelevant for the depth-first search. Moreover, these algorithms adopts various database representations to further reduce the cost of database scans. The FP-Growth algorithm compresses the database in a tree structure called FP-Tree, while the H-Mine algorithm utilizes a hyper-structure, and the LCM algorithm utilizes a standard database representation, but merges identical transactions of each projected database to reduce its size. Pattern-growth approach such as FP-Growth tends to perform best on sparse databases where each itemset does not appears in many transactions. In this case, when performing a database projection for an itemset, the database size can be greatly reduced, which will speed up the algorithm. But for dense databases, the database projection technique will be less effective at reducing the database and keeping multiple projected databases in memory may require a considerable amount of memory.

Hence, the above approaches have various advantages and disadvantages, and thus they perform better than others in different situations, and some are easier to implement than others. The problem of frequent itemset mining has been extended in many different ways [11]. Algorithms for these extended problems generally extend the above FIM approaches with additional constraints, utilize modified data structures and apply novel search space pruning strategies.

In terms of performance, traditional algorithms for frequent itemset mining have been implemented by numerous researchers in numerous languages such as C++ and Java, with various optimizations and design decisions, and have been compared in multiple papers. Due to differences in implementations, results of performance comparison may vary. For instance, Agrawal et al. [1] reported that Apriori and Apriori-TID have similar performance but Apriori-TID performs best when transaction lists are small. Zaki et. [40]

reported that Eclat performs better than Apriori, especially for dense datasets and low minimum support values, while Han et al. [19] reported that FP-Growth can be much faster and scalable than Apriori especially for low minimum support values. In another study, Hipp et al. [20] concluded that there is no clear-winner between Apriori, Apriori-TID and Eclat, and that each performs better in some cases due to their characteristics. There was also two competitions for comparing the performance of frequent itemset mining implementations at the 2003 IEEE International Conference on Data Mining (ICDM) and 2004 ICDM conferences (see http://fimi.uantwerpen.be/). Results of these competitions have shown that various optimizations and design decisions for the basic algorithms such as Apriori, FP-Growth and Eclat can lead to big performance differences (more than five times faster in some cases). At FIMI 2003, it was found that FP-Growth generally performs better than Eclat and Apriori for low minimum support values, while some implementations of Eclat and Apriori performed better than FPGrowth for higher support values on dense datasets [15].

2.2. Periodic Frequent Pattern Mining

Though, frequent itemset mining is useful, it cannot be used to find patterns that appear periodically in a database. But in many domains such as market basket analysis discovering periodic behavior reveals important information. For instance, identifying that some customers buy some products every day or week can be used to improve marketing strategies and inventory management. To identify such patterns, Tanbeer et al. [36] proposed the problem of identifying Periodic Frequent Patterns (PFPs) in transactional databases. They pointed out that patterns having regularly repeating occurrences are interesting, and designed a pattern-growth algorithm called PF-tree to identify such patterns. A pattern is periodic if the gap (number of transactions between its consecutive occurrences) is always less than a user-defined maximum *maxGap* periodicity threshold. The definition of Tanbeer et al. has been used in several studies [9, 21, 22, 24, 33, 35]. However, that definition is too strict. In fact, a pattern is discarded if it has a single gap that exceeds the maximum periodicity threshold.

Several researchers have proposed variations of this definition. Surana et al. [35] have proposed to associate a minimum support threshold and a maximum periodicity threshold to each item, to allows evaluating each item in a different way. The motivation is that this would allow a more fair measurement of the periodicity of each item. However, it raises a novel problem, which is that the number of parameters to be set by the user becomes very large. Although it is possible to set all thresholds automatically using a function, selecting an appropriate function to assign values to each item and avoid missing interesting patterns can be difficult.

Another alternative to the traditional definition of PFP was presented by Rashid et al. [33]. To determine if a pattern is periodic, the designed algorithm considers the overall distribution of the periods (number of transactions between consecutive occurrences of the pattern). They suggest using the variance of these periods to find regular frequent patterns. In that work, an itemset is said to be a regular pattern if the variance of its periods in the entire database is not greater than a user-defined maximum variance threshold. However, the variance only measures how different periods are from each other. This can result in finding many useless patterns. For example, that definition assumes that a pattern having very long periods that are equal is as interesting as a pattern having short periods that are equal. Another problem with this definition is that the presence of outlier periods may greatly influence the variance. Because of these reasons, it can be difficult to find interesting patterns using that definition.

In another work, Kiran et al. [21] argued that the constraint that periods must always be no greater than the *maxGap* threshold is very strict. To address this issue, they relaxed that constraint by considering that a pattern is periodic if its *periodic ratio* is no less than a user-defined threshold. The periodic-ratio of a pattern is the number of times that the *maxGap* constraint is satisfied by the pattern, divided by the total number of consecutive period pairs. Using the periodic-ratio, one can discover patterns that are not always periodic in a database. However, a major drawback of this work is that the periodic ratio is calculated by simply looking at each pair of consecutive occurrences one at a time, while ignoring the other occurrences that happened before or after. Thus, the periodic ratio fails to assess whether the periodic behavior of a pattern is sustained over time. This can results in finding some misleading patterns. For example, consider a pattern that alternates between very short and very long periods. If all the short periods satisfy the

maxGap threshold, then its periodic ratio would be close to 0.5. One may thus think that this pattern is 50% of the time periodic. However, the pattern actually never satisfies the *maxGap* threshold for more than two consecutive occurrences, and may thus be uninteresting.

In another study, Fournier-Viger et al. [9] proposed the PFPM algorithm, which lets users discover periodic frequent patterns using three periodicity measures, called the minimum periodicity, maximum periodicity, and average periodicity. Using these measures gives more flexibility to users as they can more precisely specify the types of patterns that they are interested in. However, this study like many others assumes that the periodicity of patterns remains more or less stable over time. Thus, this approach cannot find interesting patterns that are only periodic in specific time-intervals or that are only sold during specific non predefined time periods. But in real-life, the periodic behavior of patterns may change over time. For example, in market basket analysis, products like ice cream may only be periodic during the summer, and other products such as Halloween costumes may only sell well during some specific months.

Motivated by the fact that previous studies on periodic pattern mining can neither capture the dynamic periodic behavior of patterns or pinpoint the time-intervals where patterns are periodic, this paper proposes a novel model based on the reasonable assumption that the periodic behavior of each pattern may change over time. The proposed model is specifically designed to identify time-intervals where patterns are periodic and where this periodic behavior is sustained for some minimum amount of time.

In the field of pattern mining, some studies have considered the changing temporal behavior of patterns. For example, Muthukrishnan et al. [28] proposed a seamless change detection method that does not require setting a fixed window size unlike prior work. The algorithm applies the concept of *Cumulative Sum* to accurately estimate changes in a data stream. In another study, Wan and An [38] introduced the concept of significant milestones of transitional patterns. A milestone is a time point at which the frequency of a pattern considerably changes. However this technique has an important limitation. It is that to detect change points of a pattern, at most two significant milestones (indicating an increase or decrease of the pattern's frequency) are considered per database. As a result, the algorithm cannot partition a database into more than three time-intervals. Moreover, the algorithm cannot discard a time-interval that does not satisfy the user requirements for selecting interesting patterns. As a result, the algorithm is not flexible, and should not be used on data where more than three time-intervals must be considered to understand a pattern. In a recent study, Hackman et al. [17] proposed an algorithm to find trending itemsets for the utility measure. However, that algorithm considers only two types of changes (increase or decrease) and views the database as a single time-interval. Thus, it also cannot identify changes in non predefined time-intervals. Fournier-Viger et al. [13] also considered the utility measure to find patterns having a utility that is higher than usual using moving average crossovers. However, this work cannot be easily adapted for the periodicity measure. In another study, Rasheed et al. [32] proposed a noise resilient model to identify periodic outlier patterns in time series when considering time tolerance. The traditional PFP model was also extended to identify consecutive periods where a pattern is periodic [23] but a major problem of that work is that it still relies on the strict maximum periodicity constraint. Thus, this model can discard a set of consecutive periods because a single period is greater than the maximum periodicity threshold. Hence, this model is unsuitable for real-world data where the periodic behavior of patterns frequently vary.

On overall, studies on periodic frequent pattern mining are based on the assumption that the periodic behavior of patterns is more or less stable over time. Some studies in frequent pattern mining have attempted to find concept drift points (change points) where the frequency or utility of a pattern may change [17, 38]. But these studies are not designed to find specific time-intervals where these changes occur, relies on the very strict maximum periodicity model, and are not designed to assess the periodicity of patterns by considering the time between consecutive transactions. Thus, it is not trivial to adapt existing FIM and PFP mining techniques to find non predefined time-intervals where patterns are periodic by considering the dynamic periodic behavior of patterns. This paper addresses this challenge by proposing a model where time-intervals are concisely represented by pairs of time points. Moreover, two novel measures are proposed to ensure that the periodic behavior of patterns is sustained in time-intervals while allowing some variability, and that only time-intervals having a minimum duration are considered. Novel algorithms and pruning strategies are then presented to find these patterns efficiently.

3. Preliminaries and Problem Definition

This section introduces preliminary definitions and important notations used in this paper. Then, it defines the two novel measures and the proposed problem of mining *local periodic patterns* (patterns that are periodic in non predefined time-intervals).

Definition 1 (Temporal database). Consider a set of items (e.g. events or symbols) denoted as I. Each subset $X \subseteq I$ is called an itemset. The length of an itemset containing w items is said to be w. Furthermore, all itemsets of a length w are called w-itemsets. A *temporal database* (also called *discrete sequence*) TDB is a set of pairs, where each pair contains a transaction and a timestamp indicating when the transaction occurred. A temporal database TDB containing m transactions is denoted as $TDB = \{(T_1, ts_1), (T_2, ts_2), \cdots, (T_m, ts_m)\}$, where $T_1, T_2 \ldots T_m$ are the transactions and $ts_1, ts_2 \ldots ts_m$ are their respective timestamps. Each transaction T_c $(1 \leq c \leq m)$ is a set of symbols $T_c \subseteq I$, where c is said to be the Transaction IDentifier (TID) of T_c. A temporal database is ordered by timestamps, that is $ts_1 \leq ts_2 \leq \ldots \leq ts_m$. Thus, multiple transactions can have the same timestamp. Furthermore, let ts_{min} and ts_{max} respectively denote the first and last timestamps of TDB. In other words, $ts_{min} = ts_1$ and $ts_{max} = ts_m$.

Table 1: A temporal transaction database.

TID	Transaction	Timestamp
1	a, b, c, e	$6^{th} June$ 2018
2	a, b, c, d	$7^{th} June$ 2018
3	a, b, e	$9^{th} June$ 2018
4	c, e	$10^{th} June$ 2018
5	b, d, e	$12^{th} June$ 2018
6	b, c, e	$15^{th} June$ 2018
7	b, c, d, e	$18^{th} June$ 2018
8	a, c	$22^{nd} June$ 2018
9	a, b, d	$23^{rd} June$ 2018
10	b	$25^{th} June$ 2018

Example 1. Consider the temporal database TDB of Table 1, which will be used as a running example. This database is a discrete sequence of customer transactions, ordered by time. These transactions contain the items (products) a, b, c, d and e. Thus, the set of items that can be purchased by a customer is $I = \{a, b, c, d, e\}$. The itemset $\{a, b\}$ contains both the itemsets $\{a\}$ and $\{b\}$. Moreover, the length of $\{a, b\}$ is two, i.e. it is a 2-itemset. The temporal database is a sequence of ten transactions $(T_1, T_2, ...T_{10})$. Hence, $|TDB| = 10$. The transaction T_1 indicates that a customer has bought the items a, b, c, and e on the 6th June 2018. The transaction T_2 indicates that a customer has bought the items a, b, c, and d on the 7th June 2018. Transaction T_3 indicates that items a, b and e appear in the third transaction. The TID of this transaction is 3 and its timestamp is $ts_3 = (9^{th} June$ 2018). It can be observed that the database is ordered by ascending order of timestamps. The first and the last timestamps of this database are (6^{th} June 2018) and (25^{th} June 2018), respectively. Therefore, $ts_{min} = ts_1 = (6^{th}$ June 2018) and $ts_{max} = ts_{10} = (25^{th}$ June 2018). In this example, timestamps are represented as days. But they could be expressed using other time units such as milliseconds. Also note that discrete sequences are not limited to customer transaction data. They can also be used to encode other types of data such as sequences of events and locations.

To discover interesting patterns that appear in a temporal database, the concept of appearance (occurrence) of an itemset is presented.

Definition 2 (Appearance of an itemset in a transaction). Let there be an itemset X and a database TDB containing m transactions. The itemset X is said to appear in a transaction T_i ($i \in [1, m]$) if $X \subseteq I$. In that case, it is also said that X appears at the timestamp ts_i, which is denoted as ts_i^X. For an itemset X, let the ordered list of all timestamps at which X occurred be denoted as $TS^X = \{ts_{g_1}^X, ts_{g_2}^X, \cdots, ts_{g_n}^X\}$, where $ts_{min} \leq ts_{g_1}^X < ts_{g_2}^X < ... < ts_{g_n}^X \leq ts_{max}$, and where the notation gi denotes the TID of the transaction containing the i-th occurrence of X. The number of transactions containing X in a database TDB is said to be the *support* of X, which is denoted and defined as: $sup(X) = |TS^X|$.

Example 2. In the database TDB of Table 1, the itemset $\{c, e\}$ appears at timestamps ts_1, ts_4, ts_6 and ts_7. Hence, $TS^{\{c,e\}} = \{ts_1, ts_4, ts_6, ts_7\}$, and $sup(\{c, e\}) = |TS^{\{c,e\}}| = 4$ transactions.

The concept of support has been used in traditional frequent pattern mining to discover itemsets that frequently appear in a transaction database [1, 2, 19, 27, 30, 40]. Although frequent pattern mining has some important applications, it is not designed to find periodic patterns. Thus, the concept of PFP was introduced based on a concept of *gap* between transactions[1] [36]. It is defined as follows.

Definition 3 (Gaps between transactions of an itemset). Consider an itemset X and a database TDB containing m transactions. Let the ordered list of transactions containing an itemset X be $GS^X = \{T_{g_1}^X, T_{g_2}^X, \cdots, T_{g_n}^X\}$. The gaps of an itemset X in TDB are defined as $gap(X) = \{g_1, g_2 - g_1, g_3 - g_2, \cdots, m - g_n\}$.

Example 3. In the database TDB of Table 1, the itemset $\{c, e\}$ appears in transactions T_1, T_4, T_6 and T_7. Thus, $GS^{\{c,e\}} = \{T_1, T_4, T_6, T_7\}$, and $gap(\{c, e\}) = \{1, 4-1, 6-4, 7-6, 10-7\} = \{1, 3, 2, 1, 3\}$. The itemset $\{b\}$ appears in transactions T_1, T_2, T_3, T_5, T_6, T_7, T_9, and T_{10}. Thus, $gap(\{b\}) = \{1, 1, 1, 2, 1, 1, 2, 1, 0\}$.

In traditional PFP mining, an itemset is said to be a PFP if none of its gaps is greater than a threshold *maxGap* set by the user [36]. For example, if $maxGap = 2$, the itemset $\{c, e\}$ is not a PFP but itemset $\{b\}$ is a PFP. As explained in the introduction, there are two major problems with this definition.

- First, this model considers the number of transactions between occurrences of an itemset rather than the elapsed time. Thus, it is considered that the gap between T_1 and T_2 is as large as the gap between T_7 and T_8 (a gap of one record). However, by looking at timestamps, this is wrong as there is only 1 day between T_1 and T_2, but there are 4 days between T_7 and T_8. Thus, the traditional model is unsuitable for handling data not having equidistant timestamps.

- Second, the periodic behavior of a pattern is assumed to remain stable in the whole database. For example, the pattern $\{c, e\}$ is not a PFP for $maxGap = 2$. But this definition is too strict as it can discard a pattern because of a single gap exceeding *maxGap*. Besides, this definition misses the fact that $\{c, e\}$ could be considered as a periodic pattern between the 10th June and the 18th June. Traditional PFP mining algorithms cannot find patterns that are periodic only in some time-intervals. Thus, the traditional PFP model is only suitable for finding patterns that have a stable periodic behavior in the whole database.

To allow considering timestamps and finding patterns that are periodic in non predefined time-intervals, the concept of gaps is adapted to consider timestamps.

Definition 4 (Consecutive timestamps). Consider an itemset X and a database TDB containing m transactions. Two timestamps $ts_{g_j}^X \in TS^X$ and $ts_{g_{j+1}}^X \in TS^X$ such that $j \in [1, sup(X)]$ and $ts_{min} \leq$

[1]Note that *gaps* are called *periods* in traditional PFP mining.

$ts^X_{g_j} \leq ts^X_{g_{j+1}} \leq ts_{max}$ are said to be consecutive timestamps where X appears in TDB. To facilitate timestamp calculations, an additional timestamp is added to the set TS^X, which is denoted and defined as $ts^X_{g_{sup(X)+1}} = ts_{max}$. The time difference between two consecutive timestamps where X appears $ts^X_{g_j}$ and $ts^X_{g_{j+1}}$ is called the period of X at timestamp $ts^X_{g_j}$. It is denoted as $per(X, ts^X_{g_j})$ and defined as $per(X, ts^X_{g_j}) = ts^X_{g_{j+1}} - ts^X_{g_j}$.

Example 4. The timestamps $ts_1 = (6^{th}$ June 2018) and $ts_3 = (9^{th}$ June 2018) are consecutive timestamps for the itemset $\{b, e\}$. The time difference between these two timestamps is three days. That is, $per(\{b, e\}, ts_1) = 3$ (days).

In the proposed model, the concept of gap (number of records between consecutive transactions) is replaced by the following concept of periods (time elapsed between consecutive transactions).

Definition 5 (Periods of an itemset). Consider an itemset X and a database TDB containing m transactions. The periods of X is the list of periods defined as $per(X) = \{ts^X_{g_2} - ts^X_{g_1}, ts^X_{g_3} - ts^X_{g_2}, \cdots, ts^X_{g_{sup(X)+1}} - ts^X_{g_{sup(X)}}\}$. Thus, $per(X) = \bigcup_{1 \leq z \leq sup(X)}(ts^X_{z+1} - ts^X_z)$ and $|per(X)| = |TS^X| = sup(X)$.

Example 5. For the itemset $\{b, e\}$, $TS^{\{b,e\}} = \{ts_1, ts_3, ts_5, ts_6, ts_7\}$. It can be found that: $per(\{b, e\}, ts_1) = ts_3 - ts_1 = 3$ (days), $per(\{b, e\}, ts_3) = ts_5 - ts_3 = 3$ (days), $per(\{b, e\}, ts_5) = ts_6 - ts_5 = 3$ (days), $per(\{b, e\}, ts_6) = ts_7 - ts_6 = 3$ (days), and $per(\{b, e\}, ts_7) = ts_{max} - ts_7 = ts_{10} - ts_7 = 7$ (days). Thus, $per(\{b, e\}) = \{3, 3, 3, 3, 7\}$, and $|per(\{b, e\})| = |TS^{\{b,e\}}| = sup(\{b, e\}) = 5$.

The proposed concept of period can be viewed as more general than the concept of gap. The proposed definition of period differs in two ways from the definition of gap used by traditional PFP mining algorithms [36]. First, traditional PFP algorithms do not use timestamps. They only consider gaps between transactions, which are defined based on transaction identifiers. But as explained, this definition is unrealistic for many applications as it assumes that transactions are equally spaced in time. Second, traditional PFP algorithms define the first gap of an itemset as the identifier of its first transaction. This is equivalent to assuming that there exists an empty transaction T_0 with a transaction identifier of 0. This paper does not add an empty transaction because we want to preserve the consistency between the number of periods and the support (the number of periods becomes equal to the support), which is not true in traditional PFP mining. In the following, all period calculations for the running example will be done using timestamps where days are the time unit, and the time unit will be ommitted for the sake of brevity. It is important to note that other time units can be used, and that the proposed model can also be applied to data without timestamps by considering that transaction identifiers are the timestamps. Choosing whether to use timestamps or not, and selecting an appropriate time unit is a choice that is left to the user, and must be made based on application needs. For example, to analyze customer transactions, it may be relevant to use days, weeks, or months as time unit, while data from other applications such as text mining may lack timestamps. Besides, note that it is also possible to apply other preprocessing techniques based on user needs such as to merge transactions having same timestamps. But this is not required for applying the algorithms proposed in this paper.

To detect non predefined time-intervals where itemsets are periodic, it is necessary to define measures that evaluate how the periodic behavior of a pattern changes over time and assess whether the periodic behavior is sustained or not in various time-intervals, while allowing some variability in the periodic behavior of patterns. To address these challenges and provide a flexible model, this paper proposes a novel approach inspired by the cumulative sum, a technique used in statistics to detect changes in a time series by accumulating the *difference* between each point and a fixed number ρ [14, 28]. This accumulated difference is called the *cumulative sum*. If over time the cummulative sum exceeds another fixed number α that is greater than ρ, then a change is said to have occurred in the time series. Formally, given a sequence of r numbers $z_1, z_2, \ldots, z_r$ the cumulative sum for the i-th data point ($0 < i \leq r$) is defined as $C_i = \max(0, C_{i-1} + z_i - \rho)$ where $C_0 = 0$. Although the cumulative sum is useful to detect a change in a time series by looking at how values changes over time, it is not designed for assessing the periodicity of patterns using their periods.

Table 2: Gaps, Periods, surplus, spillover, and time-interval(s) of itemset $\{b\}$.

Transactions T_x where $\{b\}$ appears, i.e. $GS^{\{b\}}$	T_1	T_2	T_3	T_5	T_6	T_7	T_9	T_{10}
Gaps of $\{b\}$, i.e. $gap(\{b\})$	1	1	1	2	1	1	2	1, 0
Timestamps t_x where $\{b\}$ appears, i.e. $TS^{\{b\}}$	t_1 6	t_2 7	t_3 9	t_5 12	t_6 15	t_7 18	t_9 23	t_{10} 25
Periods of $\{b\}$, i.e. $per(\{b\}, t_x)$	1	2	3	3	3	5	2	0
Surplus of $\{b\}$, i.e. $surPer(\{b\}, t_x)$	-2	-1	0	0	0	2	-1	-3
Spillover of $\{b\}$ if ts_1 is start point, i.e. $soPer(\{b\}, t_x)$	0	0	0	0	0	2	1	0
Spillover of $\{b\}$ if ts_6 is start point, i.e. $soPer(\{b\}, t_x)$					2	4	3	0
Time-interval(s) of $\{b\}$	$[t_1, t_{10}]$ duration: 19							
Periodic time-interval(s) of $\{b\}$	$[t_1, t_{10}]$							

Moreover, the goal of this paper is to discover not only a single change but multiple changes in a time series to find time-intervals where a pattern is periodic. To address this challenge, two novel measures are presented called the *surplus of an itemset during a period*, and the *spillover of a period*. These measures are inspired by the concepts of *difference* and *cummulative sum*, respectively, but are adapted for evaluating the periodicity of patterns and identifying time-intervals rather than single change points.

Definition 6 (Surplus of an itemset during a period). Let there be an itemset X appearing at two consecutive timestamps ts_{g_i} and ts_{g_i+1}. Then, let $per(X, ts^X_{g_i}) = ts_{g_{i+1}} - ts_{g_i}$ be the period of itemset X at ts_{g_i}. Furthermore, let there be a user-defined parameter called the maximum period (denoted as $maxPer$). The surplus of a period is defined as the difference between the period and $maxPer$. Formally, the surplus of the period at a timestamp ts_{g_i} of itemset X is defined as $surPer(X, ts^X_{g_i}) = per(X, ts^X_{g_i})$ − maxPer. Moreover, the surplus of the periods of itemset X in the whole database is defined as: surPer(X) = per(X) − maxPer = $\{per(X, ts^X_{g_1})$ − maxPer, $per(X, ts^X_{g_2})$ − maxPer, $\cdots$, $per(X, ts^X_{sup(X)})$ − maxPer$\}$.

Example 6. The item $\{b\}$ appears at timestamps ts_1, ts_2, ts_3, ts_5, ts_6, ts_7, ts_9, and ts_{10}. The period of the item $\{b\}$ at timestamp ts_1 is denoted as $per(\{b\}, ts_1)$. This period is calculated as $per(\{b\}, ts_1) = ts_2 - ts_1 =$ 1. Assume that the user sets $maxPer = 3$. Thus, the surplus of the period of item {b} at timestamp ts_1 is $surPer(\{b\}, ts_1) = per(\{b\}, ts_1) - maxPer = 1 - 3 = -2$. The periods of the item $\{b\}$ are $per(\{b\}) =$ {1, 2, 3, 3, 3, 5, 2, 0}. Hence, the surplus of the periods of item $\{b\}$ in the whole database are $surPer(\{b\})$ $= per(\{b\}) - maxPer =$ {1−3, 2−3, 3−3, 3−3, 3−3, 5−3, 2−3, 0−3} = {-2, -1, 0, 0, 0, 2, -1, -3}. These numbers can be interpreted as by how much the itemset $\{b\}$ exceeded $maxPer$ for each of its periods. A summary of this example is provided in Table 2.

In traditional PFP mining, if an itemset has only a single gap greater than $maxPer$, the itemset is considered as non periodic, and it is discarded. In this paper, if an itemset has a non positive surplus for a period (i.e. the difference between that period and $maxPer$ is positive), the itemset is considered as periodic for that period, and otherwise it could be considered as non periodic in that period. This approach allows to assess the periodicity of a pattern during a single period. However, to detect if a pattern is periodic over a time-interval, it is necessary to evaluate if the pattern remains more or less periodic for multiple consecutive time periods, and to detect the start point and end point of that time-interval. To achieve this, a concept of spillover is proposed, which is inspired by the concept of cumulative sum. The spillover of an itemset for a time period represents the accumulated amount by which that itemset has exceeded $maxPer$ in recent periods. As long as the latter accumulated value remains no greater than a user-defined minimum threshold $maxSoPer$, the itemset is considered as being periodic in that time-interval. Then, if another

period is added to that time-interval such that the spillover exceeds the threshold, it indicates the end of the time-interval where the itemset is periodic. The purpose of using the spillover is to be more flexible in finding periodic patterns by allowing a pattern to be considered as periodic even if some of its periods exceed $maxPer$, as long as the accumulated difference with $maxPer$ (the spillover) is not too large. The formal definition of spillover and time-interval is given next.

Definition 7 (Spillover of an itemset for a time-interval). Let there be an itemset X. Furthermore, let $ts^X_{g_j}$, $j \in [1, sup(X)]$ be a timestamp where the itemset X has appeared. If $surPer(X, ts^X_{g_j}) \leq 0$, then the timestamp ts_{g_j} is considered as a potential start point for a time-interval where X is periodic. Calculating the spillover of each time point where the pattern appears is a dynamic process that starts at a certain point in time and is gradually updated by accumulating surplus values of periods over time. From the start point of a spillover, the spillover of the current period for an itemset is defined as the cumulative sum of the spillover of the previous periods and the surplus of the current period. Moreover, the spillover of each period should be no less than zero. Formally, let $ts^X_{g_j}$ be the start timestamp, and the spillover of X for that start timestamp be $soPer(X, ts^X_{g_j}) = \text{MAX}(0, soPer(X, maxSoPer) + surPer(X, ts^X_{g_j}))$ where $maxSoPer$ is a user-defined parameter called maximal period of spillover. Then, the spillover of X for any latter timestamp $ts^X_{g_k}$ is $soPer(X, ts^X_{g_k}) = \text{MAX}(0, soPer(X, ts^X_{g_{k-1}}) + surPer(X, ts^X_{g_k}))$ where $k \in [j+1, sup(X)]$. Thus, the spillover of each timestamp can be calculated as:

$$soPer(X, ts^X_{g_i}) = \begin{cases} MAX(0, maxSoPer + surPer(X, ts_{g_i})), & \text{if } ts^X_{g_i} \text{ is a start point;} \\ MAX(0, soPer(X, ts^X_{g_{i-1}}) + surPer(X, ts_{g_i})), & \text{otherwise.} \end{cases} \quad (1)$$

Example 7. Continuing the previous example, the item $\{b\}$ appears at timestamps ts_1, ts_2, ts_3, ts_5, ts_6, ts_7, ts_9, and ts_{10}. The surplus of the periods of $\{b\}$ in the database are $surPer(\{b\})$ = {-2, -1, 0, 0, 0, 2, -1, -3}. Except ts_7, all the above timestamps can be used as a start point for the item $\{b\}$. The reason why ts_7 cannot be used as start point is that the surplus of item $\{b\}$ is greater than zero at ts_7, that is $surPer(\{b\}, ts_7) = 2 > 0$. Consider that the user-defined maximum period of spillover is set as $maxSoPer = 2$. The process of calculating spillovers starts at ts_1. At the start point ts_1, the spillover of the period is $soPer(\{b\}, ts_1) = \text{MAX}(\ 0,\ maxSoPer + surPer(\{b\}, ts_1)\ ) = \text{MAX}(\ 0,\ 2 - 2) = 0$. Then, the next time point of item $\{b\}$ is ts_2, where $soPer(\{b\}, ts_2) = \text{MAX}(\ 0,\ soPer(\{b\}, ts_1) + surPer(\{b\}, ts_2)\ ) = \text{MAX}(0,\ 0 + (-1)) = 0$. Similarly, it is then found that $soPer(\{b\}, ts_3) = 0$, $soPer(\{b\}, ts_5) = 0$, $soPer(\{b\}, ts_6) = 0$, $soPer(\{b\}, ts_7) = 2$, $soPer(\{b\}, ts_9) = 1$, and $soPer(\{b\}, ts_{10}) = 0$. These results are based on choosing ts_1 as the start point. If ts_6 is instead chosen as start point, the results are different. That is $soPer(\{b\}, ts_6) = \text{MAX}(\ 0,\ maxSoPer + surPer(\{b\}, ts_6)\ ) = \text{MAX}(\ 0,\ 2 + 0\ ) = 2$, $soPer(\{b\}, ts_7) = 4$, $soPer(\{b\}, ts_9) = 3$, and $soPer(\{b\}, ts_{10}) = 0$. A summary of this example is provided in Table 2.

Having explained how spillovers are calculated, the next paragraph explains how the concept of spillover is used to identify end points of time-intervals in a database.

Definition 8 (Time-intervals of an itemset). Let ts_{g_j} be the first start point of an itemset X, where $j \in [1, sup(X)]$. A time-interval is composed of a start point and an end point. To identify the start and end points of time-intervals, a transaction database is scanned from the start point by moving forward in time. A point is considered as an end point of the current time-interval if the spillover of its period is larger than $maxSoPer$ at that moment. That is, starting from ts_{g_j}, if $soPer(X, ts_{g_{j+k}}) > maxSoPer$, then $ts_{g_{j+k}}$ is an end point. The first time-interval of X is defined as $[ts_{g_j}, ts_{g_{j+k}}]$. Using an upper-bound on the start point and a minimum duration constraint, it is possible to search for other start points occurring after the last end point (see Lemma 1). This process is repeated to obtain a series of time-intervals ordered by timestamps. For the last time-interval, if the last timestamp ts_{max} is reached and no end point is found, then ts_{max} is considered as the end point of the last time-interval. An interval closed in this way should be interpreted as indicating that the pattern was periodic in that time-interval until the end of the database. The reason for closing the time-interval in this way is that the input is a static database, and thus no data is available about the future (it is not know when the real end point will occur). Note that an alternative way of handling unclosed time-intervals is to ignore them, but this results in losing information.

Example 8. Continuing the previous example, the first start point of item $\{b\}$ is ts_1. From this point, there is only one time-interval for item $\{b\}$, which is $[ts_1,ts_{10}]$ (from the 6^{th} June 2018 to the 25^{th} June 2018). For item $\{a\}$, ts_1 is also the first start point. Starting from ts_1, it is found that $soPer(\{a\}, ts_1) = 0$, $soPer(\{a\}, ts_2) = 0$, $soPer(\{a\}, ts_3) = 10 > 2$. Thus, the first time-interval of $\{a\}$ is $[ts_1, ts_3]$. After ts_3, ts_8 meet the requirements to be a new start point for $\{a\}$. Starting from ts_8, $soPer(\{a\}, ts_8) = 0$, $soPer(\{a\}, ts_9) = 0$ and $ts_{max}(ts_{10})$ is reached, obtaining $[ts_8, ts_{10}]$ as the second time-interval of $\{a\}$.

There exists an interesting relationship between the spillovers of an itemset for a given timestamp, when using different start points. This relationship is explained by the following lemma. It defines an upper-bound on spillover values at a timestamp for an itemset.

Lemma 1 (Upper-bound of a start point). Let $[ts_i, ts_j]$ be a time-interval of X and ts_k be a timestamp that meets the requirements for being a start point, where $i < k \leq j$. Starting from ts_i, $soPer(X, ts_k) = \alpha$. If ts_k is a start point, $soPer(X, ts_k) = \beta$. It follows that $\alpha \leq \beta$.

Proof 1. If we use ts_i as start point, $soPer(X, ts_k)$ = MAX(0, $soPer$(X, ts_{k-1}) + $surPer(X, ts_k)$) = α. If we use ts_k as start point, $soPer(X, ts_k)$ = MAX(0, $maxSoPer$ + $surPer(X, ts_k)$) = β. By definition, $soPer(X, ts_{k-1}) \leq maxSoPer$. Hence $\alpha \leq \beta$.

Based on the above lemma, it is unecessary to consider all start points in a time-interval because the oldest start point will always provide the smallest spillover for a given timestamp. This allows to avoid considering multiple overlapping time-intervals, and thus to reduce the search space by always considering the oldest start points to identify time-intervals.

Another important consideration is that very short time-intervals may not be meaningful to the user. To avoid finding such time-intervals, the concept of duration is introduced in the proposed model. The user can specify a minimum duration constraint to eliminate short time-intervals.

Definition 9 (Duration of a time-interval). Let $[ts_i, ts_j]$ be a time-interval of an itemset X. The span length between ts_i and ts_j is defined as the duration of this time-interval, that is $dur(X, [ts_i, ts_j]) = ts_j - ts_i$. A time-interval of X is said to be periodic (or interesting) if its duration is no less than a user-specified minimum duration threshold, denoted as $minDur$. Formally, an itemset X is said to be periodic in a time-interval $[ts_i, ts_j]$ of X if $dur(X, [ts_i, ts_j]) \geq minDur$.

Example 9. The time-interval of item $\{b\}$ is $[ts_1,ts_{10}]$. Thus, $dur(\{b\},[ts_1,ts_{10}]) = ts_{10}$ - $ts_1 = 19$. The time-intervals of item $\{a\}$ are $[ts_1,ts_3]$ and $[ts_8,ts_{10}]$. The durations of these time-intervals are $dur(\{a\},[ts_1,ts_3])$ = ts_3 - ts_1 = 3 and $dur(\{c\},[ts_8,ts_{10}])$ = 3. Assume that $minDur$ = 7. Then, the item $\{b\}$ has a periodic time-interval $[ts_1,ts_{10}]$, while item $\{a\}$ has none. A summary of the periods, surplus, spillover, and time-interval(s) of itemset $\{a\}$ is provided in Table 3.

The equation of Definition 7 and Lemma 1 are important as they provide an efficient way of updating the spillover of a period and identifying periodic time-intervals. Using the equation of Definition 7, $O(1)$ time is required to update $soPer$ for each new item and to detect a change by testing $soPer > maxSoPer$. This allows to efficiently detect time-intervals of variable lengths. According to Definition 9, time-intervals having a duration smaller than the user-defined $minDur$ threshold can be discarded. The process of discarding time-intervals is dynamic. After discarding a time-interval, the timestamps that is included in this time-interval is not considered. According to Lemma 1, it is hence only necessary to consider the timestamps following this time-interval. This allows to avoid performing unecessary work when searching for time-intervals, and will lead to more efficient algorithms.

Based on the concept of time-interval, this paper proposes a novel type of periodic patterns.

Definition 10 (Local periodic pattern). An itemset X is a Local Periodic Pattern (LPP) if it has at least one periodic time-interval $[ts_x, ts_y]$ such that $dur(X, [ts_x, ts_y]) \geq minDur$.

Example 10. Continuing the previous example, the item $\{b\}$ is a LPP as it has a periodic time-interval $[ts_1,ts_{10}]$, while the item $\{a\}$ is not a LPP as it does not have a periodic time-interval.

Table 3: Gaps, periods, surplus, spillover, and time-interval(s) of itemset $\{a\}$.

Transactions T_x where $\{a\}$ appears, i.e. $GS^{\{a\}}$	T_1	T_2	T_3	T_8	T_9
Gaps of $\{a\}$, i.e. $gap(\{a\})$	1	1	1	5	1, 1
Timestamps t_x where $\{a\}$ appears, i.e. $TS^{\{a\}}$	t_1 6	t_2 7	t_3 9	t_8 22	t_9 23
Periods of $\{a\}$, i.e. $per(\{a\}, t_x)$	1	2	13	1	2
Surplus of $\{a\}$, i.e. $surPer(\{a\}, t_x)$	-2	-1	10	-2	-1
Spillover of $\{a\}$, i.e. $soPer(\{a\}, t_x)$	0	0	10	0	0
Time-interval(s) of $\{a\}$	$[t_1, t_3]$ duration: 3, $[t_8, t_{10}]$ duration: 3				
Periodic time-interval(s) of $\{a\}$	$\emptyset$				

Based on the above definitions, we define the problem of mining periodic time-intervals of patterns in a discrete sequence.

Definition 11 (Problem definition). Let there be a discrete sequence *TDB*, a maximum period *maxPer* that the user does not expect consecutive periods of a pattern to exceed, a maximum period of spillover (*maxSoPer*) and a minimum duration (*minDur*). The problem of mining periodic time-intervals of patterns in a discrete sequence consists of using the dynamic update mechanism of spillovers of periods, to find all periodic time-intervals having a duration that is no less than the *minDur* threshold. Patterns having at least one periodic time-interval are said to be LPPs and are presented to the user with their periodic time-intervals.

Example 11. The set of all LPPs found in the temporal database of Table 1 for $maxPer = 3$, $maxSoPer = 2$ and $minDur = 7$ are shown in Table 4, with their periodic time-intervals. Note that although all LPPs in this example have a single periodic time-interval, the proposed model allows each LPP to have more than one.

Table 4: LPPs found in the database of Table 1 for $maxPer = 3$, $maxSoPer = 2$ and $minDur = 7$.

LPP	Periodic time-interval
$\{b\}$	[6^{th} *June* 2018, 25^{th} *June* 2018]
$\{b, e\}$	[6^{th} *June* 2018, 18^{th} *June* 2018]
$\{c\}$	[6^{th} *June* 2018, 18^{th} *June* 2018]
$\{e\}$	[6^{th} *June* 2018, 18^{th} *June* 2018]

The proposed problem has three parameters, namely *maxPer*, *maxSoPer* and *minDur*. The user can utilize the *maxPer* parameter to specify the expected maximum time between consecutive occurrences of periodic patterns. For example, setting $maxPer = 2$ *days* means that a pattern would be considered as periodic in a time-interval if the pattern appears about every two days or less. The *maxSoPer* parameter is used to allow patterns to temporarily exceed the *maxPer* threshold if the spillover (cumulative sum of surplus) is smaller than *maxSoPer*. Thanks to this parameter, the proposed problem is more flexible than the traditional maximum periodicity constraint used in PFP mining. The *minDur* parameter indicates the minimum length for time-intervals and is used to discard short time-intervals. All these parameters are

dataset dependent and must be set by the user. For example, to analyze shopping data, one may decide to set $minDur = 1$ *week* to analyze sales on a weekly basis, while other data may be analyzed by considering months.

The key difference between the traditional problem of PFP mining and the proposed problem is that the former aims at finding patterns that are periodic in the whole database by considering a strict maximum gap constraint, while the latter aims at finding non-predefined time-intervals where patterns are periodic by taking the dynamic periodic behavior of patterns into account. The concept of time is handled differently. Whereas traditional PFP mining evaluates each gap separately (if a single gap is greater than $maxGap$, a pattern is discarded), the proposed problem applies the cumulative sum to consider the relationship between each period and the previous ones. This definition of periodicity has the advantage of being more noise-tolerant since a pattern may have a value greater than $maxPer$ at a time point and not be discarded because of the contribution of the preceding periods to the cumulative sum. Because of these differences, it is not possible to transform the proposed problem into the traditional PFP mining problem.

Discovering the proposed LPPs is thus desirable to identify patterns that are locally periodic. But efficiently finding these LPPs with their time-intervals is an important challenge as the number of possible patterns can be very large and many potential time-intervals may have to be considered. If a database contains z items, then up to $2^z - 1$ itemsets may have to be considered. Moreover, if a database contains x timestamps, up to $\lfloor \frac{x}{minDur} \rfloor \times (2^z - 1)$ time-intervals may have to be considered for each itemset. Next section presents three algorithms to efficiently find all LPPs in a discrete sequence.

4. Algorithms

This section presents the designed algorithms to identify local periodic patterns. The first subsection presents some theoretical results that are the foundation of these algorithms. Then, the following subsections presents the three algorithms. They use different approaches to find patterns: a breadth-first search using a vertical database, a depth-first search using a vertical database and a pattern-growth approach using a tree-based database representation. These algorithms are extending the basic search procedures of the *Apriori-TID* [1], *Eclat* [40] and *FP-Growth* [19] algorithms, respectively, for mining local periodic patterns. The *Apriori-TID*, *Eclat* and *FP-Growth* algorithms are designed for mining frequent itemsets and do not consider the periods of items nor their timestamps. Some of these algorithms have previously been extended for PFP mining. For example, Eclat has been extended in the PFPM [9] algorithm, and FP-Growth has been extended in the PF-Tree algorithm [36]. However, such algorithms [9, 22, 33, 36] are traditional PFP mining algorithms, which do not consider the temporal relationships between periods, and are not designed to identify time-intervals of interest where patterns may be periodic. Thus, these algorithms cannot be directly applied to the problem of mining LPPs. Hence, we designed some novel data structures to detect and save information about time and time-intervals, and designed unique pruning strategies to reduce the search space for mining LPPs. As it will be shown in the experimental evaluation of this paper, each proposed algorithm has some advantages in some situations.

4.1. Theoretical results and search space pruning properties

The proposed algorithms explore the search space of itemsets by starting from single items and recursively appending single items to these itemsets to consider larger itemsets. To avoid generating the same itemsets more than once a processing order on items $\succ$ is used. This order can be any total order such as the ascending lexicographical order. In the following, the concept of extension is used to describe how the algorithms explore the search space:

Definition 12 (Extension). Consider an itemset P and an item z such that $z \succ$ k $\forall$k$\in$P. The itemset $P \cup \{z\}$ is called an extension of P. For brevity, the notation Pz is also used to refer to $P \cup \{z\}$. Consider another item y such that $y \succ k \ \forall k \in P$. The two itemsets Pz and Py are said to have the same prefix P.

Example 12. The itemsets $\{a, b\}$ and $\{a, c\}$ are both extensions of the itemset $\{a\}$, which is an extension of the empty set.

To develop efficient algorithms for mining patterns in a database, it is important to design efficient search space pruning strategies to avoid considering all possible patterns [11, 10]. For the proposed problem of mining LPPs, the search space is quite large because not only various combinations of items must be considered but also various time-intervals, and the periodicity of patterns must be evaluated as well. To be able to reduce the search space, the following properties are proposed.

Lemma 2 (Anti-monotonicity of the total duration of periodic time-intervals). Let X and Y be itemsets such that $X \subset Y$. The total duration of periodic time-intervals of X is denoted as $dur(X)$ and the total duration of periodic time-intervals of Y is denoted as $dur(Y)$. It follows that $dur(X) \geq dur(Y)$.

Proof 2. Since $X \subset Y$, Y may only appear when X appears. For any time-interval $[ts_a, ts_b]$ where X appears, Y may appear in it or not. If Y appears in it with a time-interval $[ts_c, ts_d]$, that is $ts_c \geq ts_a$ and $ts_d \leq ts_b$, then $dur(X, [ts_a, ts_b]) \geq dur(Y, [ts_c, ts_d])$. If Y does not appear in it, then the duration of Y is zero in $[ts_a, ts_b]$. Hence, $dur(X) \geq dur(Y)$.

Property 1 (Pruning using the duration of a periodic time-interval). Let X be an itemset appearing in a temporal database TDB, and $[ts_i, ts_j]$ be a time-interval. If $dur(X, [ts_i, ts_j]) < minDur$, then the itemset X and its supersets are not periodic in $[ts_i, ts_j]$. Thus, if this condition is met, the search space consisting of this time-interval for X and all its supersets can be discarded.

Proof 3. By definition 9, for a certain time range $[ts_a, ts_b]$, if $dur(X, [ts_a, ts_b]) < minDur$, $[ts_a, ts_b]$ is not a periodic time-interval for X. By Lemma 2, supersets of X are also not periodic in that time-interval.

Property 2 (Pruning using total duration of periodic time-intervals). Let X be an itemset appearing in a temporal database TDB such that the total duration of periodic time-intervals of X is $dur(X)$. If $dur(X) = 0$, then X is not a LPP as well as all of its supersets. Thus, if this condition is met, the part of the search space consisting of X and all its supersets can be discarded.

Proof 4. By definition 10, if X has no periodic time-interval, then X is not a LPP. By Lemma 2, supersets of X are also not LPPs.

4.2. $LPPM_{breadth}$ algorithm

This subsection introduces the first proposed algorithm, named $LPPM_{breadth}$, to efficiently discover LPPs in a temporal database (a discrete sequence). The algorithm is based on the *Apriori-TID* algorithm, and relies on a novel data structure called a *timestamp-list* (*ts-list*).

Definition 13 (Timestamp list). The *ts-list* of an itemset X in a temporal database TDB is the ordered list of timestamps TS^X where X occurred. Moreover, the ts-list of an itemset X is annotated with two values, namely $soPer(X)$ and $dur(X)$.

Example 13. In the database TDB of Table 1, the itemset $\{c, e\}$ appears in transactions T_1, T_4, T_6 and T_7. Thus, its ts-list contains the timestamps $\{6^{th}June$ 2018, $10^{th}June$ 2018, $15^{th}June$ 2018, $18^{th}June$ 2018$\}$.

A useful property of ts-lists is that the ts-list of an itemset $X \cup Y$ can be obtained without scanning the database by simply intersecting the ts-list of X with that of Y. For instance, the ts-list of $\{a, c\}$ can be calculated as the intersection of the ts-list of $\{a\}$ with that of $\{c\}$, that is $\{6^{th}June$ 2018, $7^{th}June$ 2018, $9^{th}June$ 2018, $22^{nd}June$ 2018, $23^{rd}June$ 2018$\} \cap \{7^{th}June$ 2018, $12^{nd}June$ 2018, $18^{th}June$ 2018, $23^{rd}June$ 2018$\} = \{7^{th}June$ 2018, $23^{rd}June$ 2018$\}$. This process is illustrated in Fig. 1 (a).

In terms of implementation, the list of timestamps of a ts-list can be represented as an array of numbers, as shown in Fig.1 (a). However, this can require a considerable amount of memory for dense datasets where an itemset X appears at many timestamps. An alternative representation is to encode each ts-list as a bit vector, where the i-th bit indicates if the itemset appears or not (1 or 0) at the i-th timestamp. For instance, the list of timestamps of the itemset $\{a, c\}$ can be represented as the bit vector 1100000100 indicating that

(a) Array representation (b) Bit vector representation

Figure 1: The ts-lists of $\{a\}$, $\{c\}$ and $\{a, c\}$ as arrays or bitvectors.

$\{a, c\}$ appears at the first, second and eighth timestamps of the database, as depicted in Fig. 1 (b). To be able to translate bit vector positions into real timestamps an array is also created where the i-th position stores the i-th timestamp, as illustrated in Table. 5. The bit vector representation can reduce the required memory for storing ts-lists for dense databases but also has the advantage of allowing to quickly calculate the intersection of the ts-lists of two itemsets using the bitwise AND operator. For instance, Fig. 1 (b) shows how the AND operator is applied on the ts-lists of $\{a\}$ and $\{c\}$ to obtain the ts-list of $\{a, c\}$. The bit vector representation is used in the implementations of $LPPM_{breadth}$ and $LPPM_{depth}$.

Table 5: Array storing the mapping between bit vector positions and timestamps.

bit vector position	1	2	3	4	5	6	7	8	9	10
timestamp	6	7	9	10	12	15	18	22	23	25

The key difference between the proposed ts-list structure and the vertical structure of *Apriori-TID* is that the ts-list structure stores timestamps instead of transaction identifiers so that periodic time-intervals can be easily computed.

The pseudocode of $LPPM_{breadth}$ is shown in Algorithm 1. It takes as input a temporal database (a discrete sequence), and the *maxPer*, *maxSoPer* and *minDur* thresholds, and outputs all LPPs. As soon as the algorithm finds an LPP, the algorithm outputs it (e.g. by printing the pattern in the console, saving it to a file or keeping it in main memory). The algorithm first converts the database into a vertical database *VD*, consisting of the ts-list of each item (represented as a bit vector). Then, for each item $k \in VD$, the algorithm initializes the periodic time-interval list (PTL) of $\{k\}$, denoted as $PTL^{\{k\}}$, as the empty set. Then, the algorithm scans the *ts-list* of k in order of increasing timestamps while dynamically calculating $soPer(\{k\})$ to identify each time-interval $[ts_i, ts_j]$, where $ts_i, ts_j \in TS^{\{k\}}$. Note that because timestamps that are before the first time-interval of $\{k\}$ cannot appear in any periodic time-interval of $\{k\}$ or its supersets, those timestamps are removed from the *ts-list* of $\{k\}$ during the scan of its *ts-list*. Then, for each identified time-interval $[ts_i,ts_j]$, the algorithm calculates $dur(\{k\}, [ts_i,ts_j])$. If $dur(\{k\}, [ts_i,ts_j]) < minDur$, all timestamps in this time-interval can be removed from the *ts-list* of k (or the corresponding bits can be set to zero), based on Property 1. Otherwise, if $dur(\{k\}, [ts_i,ts_j]) \geq minDur$, then this time-interval is a periodic time-interval of $\{k\}$ and it is thus added to $PTL^{\{k\}}$. After finishing scanning the *ts-list* of $\{k\}$, the whole periodic time-interval list (PTL) of $\{k\}$ has been obtained. If $PTL^{\{k\}}$ contains at least a periodic time-interval, then $\{k\}$ is output as a LPP with $PTL^{\{k\}}$ Moreover, $\{k\}$ is added to a set I^* of all items that are LPPs. Only these items will then be considered by the algorithm to find larger LPPs because other items cannot be part of a LPP according to Property 2. After processing all items, items in I^* are sorted in ascending lexicographical order $\succ$, which will be thereafter used as processing order to avoid generating the same itemsets multiple times. Then, the breadth-first search exploration of itemsets starts by calling the recursive $LPPM_{breadth}Search$ procedure with a *map* having an entry with key $= \emptyset$ and value $= k$ for each item $k \in I^*$. Moreover, the *maxPer*, *maxSoPer* and *minDur* thresholds, and ts_{max}, are also passed as parameters.

The $LPPM_{breadth}Search$ procedure (Algorithm 2) explores the search space by combining pairs of w-LPPs (LPPs having w items) to generate LPPs having $(w + 1)$ items. The procedure takes as input

Algorithm 1: The $LPPM_{breadth}$ algorithm

input : TDB: a temporal database,
$maxPer$, $maxSoPer$, $minDur$: the user-specified thresholds
output: the set of LPPs and their periodic time-intervals

```
1  Convert TDB into a vertical database VD and calculate ts_max;
2  I* ← ∅;
3  foreach item k ∈ VD do
4  |  PTL^{k} ← ∅;
5  |  foreach time-interval [ts_i, ts_j] obtained by scanning the ts-list of {k} ∈ VD while calculating
   |   soPer({k}) do
6  |  |  if dur({k}, [ts_i, ts_j]) ≥ minDur then PTL^{k} ← PTL^{k} ∪ [ts_i, ts_j];
7  |  |  else Remove timestamps [ts_i, ts_j] from the ts-list of {k};
8  |  end
9  |  if |PTL^{k}| > 0 then
10 |  |  I* ← I*∪{k};
11 |  |  Output {k} with its PTL^{k};
12 |  end
13 end
14 Sort I* in ascending lexicographical order;
15 Create a map having an entry key = ∅ and value = k for each item k ∈ I*;
16 while map.size > 0 do
17 |  map ← LPPM_breadthSearch (map, maxPer, maxSoPer, minDur, ts_max);
18 end
```

the $maxPer$, $maxSoPer$ and $minDur$ thresholds, and the largest timestamp ts_{max} of the input database. Moreover, it takes as input a w-map where each entry represents a w-LPP of the form $P \cup \{z\}$ where P is a $(w-1)$-LPP and $z \succ k \ \forall k \in P$. More precisely, all entries representing itemsets having a same prefix P (the first w-1 items) are grouped using the same key, and the list of the last items *z-list* of these itemsets are the corresponding values. Initially, the $LPPM_{breadth}Search$ procedure is called with the 1-*map* containing all LPPs of the form $P \cup \{z\}$ where $P = \emptyset$ and $z \in I^*$.

The $LPPM_{breadth}Search$ procedure is applied as follows. First an empty $(w+1)$-*map* is created for storing $(w+1)$-LPPs. Then, a loop is performed to process each key P of the w-*map* to find larger itemsets having P as prefix. For a key P, the timestamps of P are first calculated by intersecting the ts-lists of all items in P. Then, another loop is performed to combine each pair of LPPs of the form Px and Py from the w-*map* to form an itemset Pxy. Note that two itemsets Px and Py are only combined if $y \succ x$ to avoid generating the same itemset twice. Finding all pairs of itemsets of the form Px and Py that have the same prefix P is fast because Px and Py are both stored in the same key P of the w-*map* (with values x and y, respectively). For each itemset Pxy obtained in this way, the timestamps TS^{Pxy} are calculated by intersecting the ts-lists of P, x and y. Then, the periodic time-interval list (PTL) of Pxy are calculated by calling the *time2interval* procedure (Algorithm 3) with the *ts-list* of Pxy. This procedure applies the equations presented in the previous section. Then, if PTL^{Pxy} contains at least a periodic time-interval, Pxy is output as a LPP with PTL^{Pxy}. Moreover, Pxy is added to the $(w+1)$-*map* as an entry with Px as key and y is added to value. When the two loops terminate, all LPPs having $(w+1)$ items have been output and stored in the $(w+1)$-*map*. Then, the recursive $LPPM_{breadth}Search$ procedure is called with the $(w+1)$-*map* to continue the breadth-first search and find LPPs having $(w+2)$ items. This process continues in that way until no more itemsets can be generated. Then, all LPPs have been output by the algorithm.

To provide more details about how $LPPM_{breadth}$ is applied, a brief example is given. Consider the temporal database of Table 1 and that $maxPer$, $maxSoPer$ and $minDur$ are set to 3, 2 and 7, respectively. $LPPM_{breadth}$ first converts the database into a vertical database. The resulting *ts-list* of each item is shown

Algorithm 2: The $LPPM_{breadth}$Search procedure

input : *w-map*: a map containing w-LPPs,
$maxPer$, $maxSoPer$, $minDur$: the user-specified thresholds,
ts_{max}: the largest timestamp of the input database
output: the (w+1)-*map* containing the set of LPPs having $w+1$ items, and their periodic time-intervals

```
Create the (w+1)-map as an empty map;
foreach key P of the w-map do
    Calculate TS^P by intersecting the ts-lists of items ∈ P;
    foreach item x and item y that are values associated to the key P in the w-map, and such that y ≻ x do
        TS^Pxy = TS^P ∩ TS^{x} ∩ TS^{y};
        PTL^Pxy = time2interval(TS^Pxy, maxPer, maxSoPer, minDur, ts_max);
        if |PTL^Pxy| > 0 then
            Output the itemset Pxy and PTL^Pxy;
            Add an entry to the (w+1)-map such that key = Px and y is added to value;
        end
    end
end
return (w+1)-map;
```

in Fig. 2 (a). For each item, the periodic time-interval (PTL) is calculated using its *ts-list* by calling the *time2interval* algorithm. The result is shown in Fig. 2 (b). It is found that item d has no periodic time-interval, and that the time-interval $[t_8, t_{10}]$ of a does not satisfy the $minDur$ constraint, while items b, c and e have PTLs and are thus LPPs. Then, the algorithm discards the items a and d from the vertical database because they are not LPPs. Moreover, timestamps that are not part of the $PTLs$ of 1-LPPs are also removed. For instance, because t_8 is not in the PTL of $\{c\}$, it is removed from the *ts-list* of $\{c\}$ (the corresponding bit is set to zero in the *ts-list* of $\{c\}$), as shown in Fig. 2 (c). Then, b, c and e are sorted in ascending lexicographical order ($b \prec c \prec d$). These items are then inserted in the 1-*map* with $\emptyset$ as key. The 1-*map* is illustrated in Fig. 2 (d). After that, the algorithm tries to generate itemsets having two items. It considers $\{b, c\}$ and $\{b, e\}$ as potential LPPs having $\{b\}$ as prefix. The *ts-list* of $\{b, c\}$ is obtained by applying the AND logical operator on the *ts-list* of b and the *ts-list* of c, which results in 1100011000. Similarly, it is found that the *ts-list* of $\{b, e\}$ is 1010111000. Then, by applying the *time2interval* algorithm, the PTL $[t_1, t_7]$ is obtained for $\{b, e\}$ while no PTL is found for $\{b, c\}$. Hence, the 2-*map* is built having an entry with $\{b\}$ as key and only $[e]$ as value. Then, the algorithm consider $\{c, e\}$ as potential LPP having $\{c\}$ as prefix. However, it is found that $\{c, e\}$ has no PTL. Thus, $\{c, e\}$ is not inserted in the 2-*map*. The final 2-*map* is shown in Fig. 2 (d). Because the algorithm cannot generate larger itemset from the 2-*map*, the algorithm terminates. Totally, four LPPs have been found. They are listed with their PTLs in Table 4.

The $LPPM_{breadth}$ algorithm initially finds LPPs containing single items and then applies the procedure named $LPPM_{breadth}Search$ to generate all other LPPs using a breadth-first search. When the search procedure receives items having w items as parameter, it combines them to generate itemsets having $w+1$ items. This type of breadth-first search exploration, originally introduced in the Apriori algorithm for frequent itemset mining [1], guarantees that all possible itemsets can be considered. And because the algorithm only prunes the search space using Property 1 and 2, it is guaranteed that no LPPs will be missed. Thus, it can be easily seen that this procedure is correct and complete to discover all LPPs.

To ensure the high performance of $LPPM_{breadth}$ for mining LPPs, two strategies have been used in the algorithm's design, which are explained in more details below.

Strategy 1. Only using the Timestamp list of Single items (OTS). To calculate the *ts-list* of a w-itemset (w>1), a solution is to maintain the ts-lists of (w-1)-itemsets in memory and intersect them to

Algorithm 3: The *time2interval* algorithm

input : TS^X: the ts-list of the itemset X,
$maxPer$, $maxSoPer$, $minDur$: the user-specified thresholds,
ts_{max}: the maximum timestamp of the TDB
output: PTL^X, the periodic time-interval list of the itemset X

$ts_{pre} \leftarrow TS^X[0]$, $start \leftarrow -1$;
foreach *timestamp* $ts \in TS^X[1:]$ **do**
  $per(X, ts_{pre}) \leftarrow ts - ts_{pre}$;
  if *per(X, ts)* $\leq maxPer \wedge start = -1$ **then**
    $start \leftarrow ts_{pre}$, $soPer(X) \leftarrow maxSoPer$;
  end
  if $start \neq -1$ **then**
    $soPer(X) \leftarrow MAX(0, soPer(X) + per(X, ts) - maxPer)$;
    if *soPer(X)* $> maxSoPer$ **then**
      if $dur(X, [start, ts_{pre}]) \geq minDur$ **then**
        $PTL^X \leftarrow PTL^X \cup [start, ts_{pre}]$;
      end
      $start \leftarrow -1$;
    end
  end
  $ts_{pre} \leftarrow ts$;
end
if $start \neq -1$ **then**
  $soPer(X) \leftarrow MAX(0, soPer(X) + ts_{max} - ts_{pre} - maxPer)$;
  if $soPer(X) > maxSoPer \wedge dur(X, [start, ts_{pre}]) \geq minDur$ **then**
    $PTL^X \leftarrow PTL^X \cup [start, ts_{pre}]$;
  end
  if $soPer(X) \leq maxSoPer \wedge dur(X, [start, ts_{max}]) \geq minDur$ **then**
    $PTL^X \leftarrow PTL^X \cup [start, ts_{max}]$;
  end
end
return PTL^X;

obtain those of the w-itemset. Although this approach has the benefit of avoiding database scans, it can require a considerable amount of memory to store ts-lists. To avoid keeping the ts-lists of (w-1)-itemsets in memory, the strategy adopted by $LPPM_{breadth}$ is to calculate the *ts-list* of a w-itemset by intersecting the ts-lists of its single items. Since the intersection operation is very fast on bit vectors, this strategy does not increase much the runtime for calculating ts-lists but can considerably reduce memory usage.

Strategy 2. Sharing Prefix using the Map structure (SPM). To generate $(w+1)$-itemsets from w-itemsets, a naive solution is to compare all w-itemsets with each other. However, this would be very costly. To avoid this problem, the strategy used in this paper is to store all w-itemsets having a same prefix P in a map using a same key and where the value representing an itemset Px is the item x. Then, all w-itemsets having a same prefix can be directly found and combined to generate $(w+1)$-itemsets without performing unecessary comparisons. This strategy is combined with the OTS strategy in $LPPM_{breadth}$. In the experimental evaluation section of this paper, it will be shown that this strategy can considerably reduce runtime and memory usage.

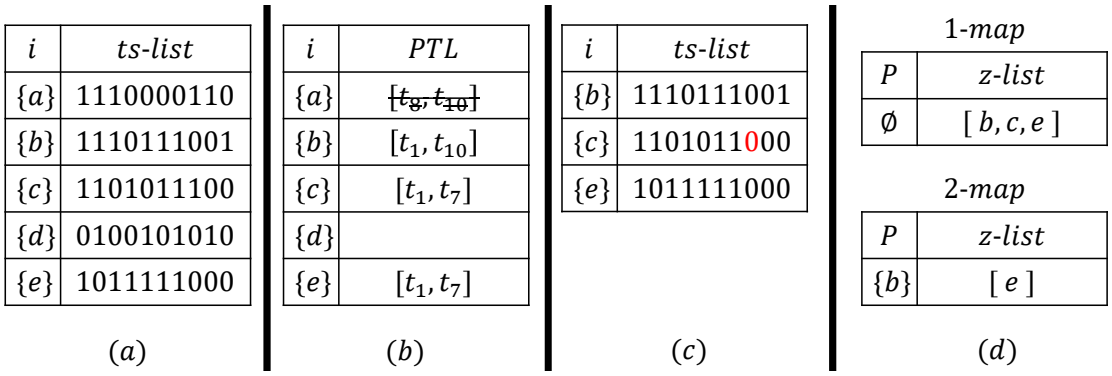

Figure 2: Illustration of the mining process of $LPPM_{breadth}$ on the database of Table 1. The (a) vertical database (*ts-list* of each item), (b) the PTL of each item, (c) the *ts-list* of 1-LPPs, (d) the 1-*map* and 2-*map*.

4.3. $LPPM_{depth}$ algorithm

This subsection introduces the second proposed algorithm, named $LPPM_{depth}$. It relies on the same *ts-list* structure as $LPPM_{breadth}$ but employs a depth-first search instead of a breadth-first search.

Algorithm 4: The $LPPM_{depth}$ algorithm

input : TDB: a temporal database,
$maxPer$, $maxSoPer$, $minDur$: the user-specified thresholds
output: the set of LPPs and their periodic time-intervals

1 Convert TDB into a vertical database VD and calculate ts_{max};
2 $I^* \leftarrow \emptyset$;
3 **foreach** *item* $k \in VD$ **do**
4 $\quad PTL^{\{k\}} \leftarrow \emptyset$;
5 $\quad$ **foreach** *time-interval* $[ts_i, ts_j]$ *obtained by scanning the ts-list of* $\{k\} \in VD$ *while calculating* $soPer(\{k\})$ **do**
6 $\quad\quad$ **if** $dur(\{k\}, [ts_i, ts_j]) \geq minDur$ **then** $PTL^{\{k\}} \leftarrow PTL^{\{k\}} \cup [ts_i, ts_j]$;
7 $\quad\quad$ **else** Remove timestamps $[ts_i, ts_j]$ from the ts-list of {k};
8 $\quad$ **end**
9 $\quad$ **if** $|PTL^{\{k\}}| > 0$ **then**
10 $\quad\quad$ $I^* \leftarrow I^* \cup \{k\}$;
11 $\quad\quad$ Output $\{k\}$ with its $PTL^{\{k\}}$;
12 $\quad$ **end**
13 **end**
14 Sort I^* in ascending lexicographical order;
15 $LPPM_{depth}Search$ (I^*,$maxPer$, $maxSoPer$, $minDur$, ts_{max});

The pseudocode of $LPPM_{depth}$ is presented in Algorithm 4. The main procedure is the same as $LPPM_{breadth}$ for identifying LPPs containing a single item. But then the algorithm performs a depth-first search exploration of itemsets by calling the recursive $LPPM_{depth}Search$ procedure with the set of single items I^*, $maxPer$, $maxSoPer$ and $minDur$ instead of performing a breadth-first search.

The $LPPM_{depth}Search$ procedure (Algorithm 5) takes as input the user-defined thresholds, the timestamp ts_{max}, and a set of LPPs called $ExtensionsOfP$. This set contains LPPs that extend an itemset P and have the form Pz, where z is any item such that $z \succ k \ \forall k \in P$. When the search procedure is called for the first time, $ExpensionsOfP$ contains LPPs having a single item, which extend the itemset $P = \emptyset$.

The $LPPM_{depth}Search$ procedure performs a loop on each extension Px of P. Because Px is a LPP, it

Algorithm 5: The $LPPM_{depth}Search$ algorithm

input : $ExtensionsOfP$: a set of extensions of an itemset P,
$maxPer$, $maxSoPer$, $minDur$: the user-specified thresholds,
ts_{max}: the largest timestamp in the input database
output: the set of LPPs and their periodic time-intervals

foreach *itemset* $Px \in ExtensionsOfP$ **do**
  $ExtensionsOfPx \leftarrow \emptyset$;
  foreach *itemset* $Py \in ExtensionsOfP$ *such that* $y \succ x$ **do**
    $TS^{Pxy} = TS^{Px} \cap TS^{Py}$;
    $PTL^{Pxy} = time2interval(TS^{Pxy}, maxPer, maxSoPer, minDur, ts_{max})$;
    if $|PTL^{Pxy}| > 0$ **then**
      Output the itemset Pxy and its PTL;
      $ExtensionsOfPx \leftarrow ExtensionsOfPx \cup \{Pxy\}$;
    end
  end
  $LPPM_{depth}Search$ ($ExtensionsOfPx$,$maxPer$, $maxSoPer$, $minDur$, ts_{max});
end

means that extensions of Px should be explored by Property 2. This is performed by merging Px with all extensions Py of P such that $y \succ x$ to form extensions of the form Pxy containing $|Px|+1$ items. The ts-list of Pxy is then constructed by intersecting the ts-list of Px and Py, which have been previously calculated. Then, the procedure scans the ts-list of Pxy to dynamically calculate $soPer(Pxy)$ and obtain the periodic time-intervals of Pxy (PTL^{Pxy}) by applying the *time2interval* procedure. If PTL^{Pxy} is not empty, Pxy is a LPP and Pxy is output with its PTL^{Pxy}. Then, the $LPPM_{depth}Search$ procedure is recursively called with Pxy to consider extensions of Pxy in the same way. This depth-first search process is continued until no more itemsets can be generated. Then, the set of all LPPs has been output.

The depth-first search exploration ensures that all itemsets can be visited. Moreover, the pruning strategies used by $LPPM_{depth}$ are the same as those used by $LPPM_{breadth}$ and have been proved formally to only prune itemsets that are not LPPs. Thus, it can be easily seen that $LPPM_{depth}$ is also correct and complete to discover all LPPs.

In terms of implementation, the OTS strategy is not applied in $LPPM_{depth}$. The reason is that a prefix P can be considered multiple times by the $LPPM_{depth}Search$ procedure, and if the *ts-list* of P is reconstructed every time, it would greatly increase the runtime. The concept of processing itemsets using a depth-first search by considering sets of items having a same prefix was initially proposed in the *Eclat* algorithm in frequent itemset mining.

4.4. LPP-Growth algorithm

This subsection proposes the third algorithm for efficiently mining LPPs, which is named *LPP-Growth*. It is applied in two steps: (i) it first compresses the input database into a time-interval periodic frequent tree (LPP-tree) structure and (ii) then recursively mines the LPP-tree to find all LPPs. The *LPP-Growth* algorithm is inspired by the tree-based pattern-growth approach of the FP-Growth algorithm but adapted for the task of mining LPPs.

This section first describes the LPP-tree structure used by *LPP-Growth* and how it is constructed. Then, it explains how the LPP-tree is mined to find all LPPs.

The LPP-tree structure. It consists of a prefix-tree and a LPP-list. The LPP-list contains entries having two fields: (i) an item name and (ii) a periodic time-interval list (PTL). The LPP-tree is a tree structure inspired by the prefix-tree structure used by FP-Growth for storing transactions (called FP-tree). As the FP-tree, the LPP-tree stores transactions as paths in a tree where each node represent an item. However, a key difference between the FP-ree and the LPP-tree is that LPP-tree nodes explicitly maintain

occurrence information about items in transactions to efficiently calculate the PTL of patterns. More specifically, an occurrence timestamp list, called *ts-list*, is stored in the last node of every transaction in the LPP-tree. Hence, two types of nodes are maintained in a LPP-tree: **ordinary** nodes and **tail** nodes. The ordinary nodes are similar to FP-tree nodes, whereas a tail node represents the last item of a sorted transaction. The structure of a tail node is $i[ts_a, ts_b, \cdots, ts_c]$, where i is the node's item name and each t_j, $j \in [1, m]$ is a timestamp where item i is the last item of the transaction. To facilitate tree traversal, each node in the prefix-tree maintains parent, children and node traversal pointers. Unlike an FP-tree, LPP-tree nodes do not store support count values. Thus, ordinary nodes only contain an item name. To obtain a highly compact tree, items in the prefix-tree are arranged in descending order of their **total duration** of periodic time-interval in PTL, and the ts-list of a node only stores the timestamps in its PTL. The next paragraph explains how a LPP-tree is constructed and mined.

Construction of a LPP-tree. To build an LPP-tree, there are two main steps: building the LPP-list and construcing the prefix-tree.

Algorithm 6 describes the steps for constructing the LPP-list by reading a temporal database. The algorithm takes as input a temporal database and the user-specified $maxPer$, $maxSoPer$ and $minDur$ thresholds. The algorithm reads the database once and outputs the LPP-list, which is a sorted list of items with their periodic time-intervals. The algorithm uses two temporary arrays named ts and p to respectively store the current timestamp and $soPer$ value of each item while scanning the database. The algorithm performs a loop over timestamps. At a given timestamp, the algorithm performs a second loop to process each appearing item. The PTL-list of the item is updated by detecting start points and end points of its interval, and if an item has a periodic time-interval it is inserted in the LPP-list. Finally, when the database scan is completed, it is possible that some time-intervals have no end points. In those case, the timestamp ts_{max} is used as endpoint. Then, the algorithm returns the LPP-list, sorted in descending order of total duration. The list contains all LPPs having a single item.

This LPP-list construction process is illustrated with an example. Consider the temporal database of Table 1 and that $maxPer$, $maxSoPer$ and $minDur$ have been set to 3, 2 and 7, respectively. Figures 3(a)-(d) show the construction of the LPP-list after scanning the first, second, and eighth transactions, and after scanning the whole database, respectively. Figures 3 (e) shows the result of adding ts_{max} to close time-intervals that have no end points (here, $ts_{max} = ts_{10}$). Finally, Figures 3 (f) shows the final LPP-list containing LPPs having a single item, sorted in descending order of their total duration.

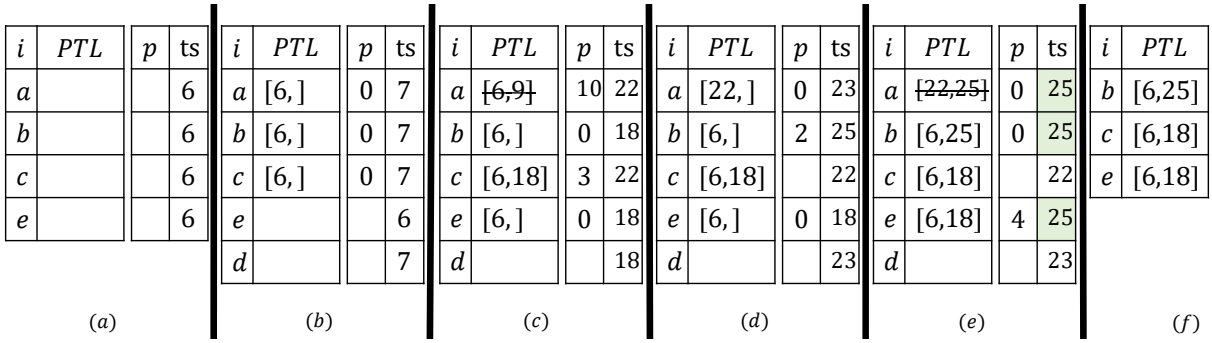

Figure 3: Construction of the LPP-list for the database of Table 1. after (a) scanning the first transaction, (b) the second transaction, (c) the eighth transaction, (d) the entire database, and after (e) adding ts_{max} to the time-intervals that had no end points, and (f) the final LPP-list sorted by descending order of total duration

After the LPP-list has been build, the next step is to build the prefix-tree of the LPP-tree by scanning the database a second time. Algorithm 7 and 8 describe the steps for constructing the prefix-tree. The algorithm takes as input a temporal database and the LPP-list containing LPPs and their PTLs. The construction of the prefix-tree is similar to that of the FP-tree [19]. However, as previously mentionned, the information stored is different. In a LPP-tree, there is no support values and **tail** nodes of a LPP-tree store information about timestamps of periodic time-intervals. The algorithm loops over each transaction and

Algorithm 6: Construction of a LPP-list

input : TDB: a temporal database,
$maxPer$, $maxSoPer$, $minDur$: the user-specified thresholds
output: the LPP-list containing the sorted list of items

Let s^i denotes the start point of the current time-interval.
foreach *itemset* X *and current timestamp* $ts \in TDB$ **do**
  foreach *item* $i \in X$ **do**
    if *item* i *is not encountered for the first time* **then**
      $per(i, ts_{pre}) = ts - ts_{pre}$;
      if $per(i, ts_{pre}) \leq maxPer \wedge s^i = -1$ **then**
        $s^i \leftarrow ts_{pre}$;
        $soPer(i) \leftarrow maxsoPer$;
      end
      if $s^i \neq$ *-1* **then**
        $soPer(i) \leftarrow MAX(0, soPer(i) + per(i, ts_{pre}) - maxPer)$;
        if $soPer(i) > maxSoPer$ **then**
          if $dur(i, [s^i, pre_{ts}]) \leq minDur$ **then**
            $PTL^i \leftarrow PTL^i \cup [s^i, pre_{ts}]$;
            LPP-list $\leftarrow$ (i, PTL^i);
          end
          $s^i \leftarrow -1$;
        end
      end
    end
    else $ts_{pre} \leftarrow ts$; $s^i \leftarrow -1$; LPP-list $\leftarrow$ (i, PTL^i);
  end
end
Add ts_{max} to PTL^i for each item i such that $s^i \neq -1$;
Prune all items from the LPP-list that have $|PTL^i| \not> 0$;
Consider the remaining items in LPP-list as LPPs and sort them in descending order of their total duration;
Return the LPP-list;

timestamp. For a given transaction X, items that are not LPPs are ignored, and the transaction is sorted using the same order as the LPP-list. Then, the *addTransaction* procedure is called to insert the resulting transaction X^* into the prefix-tree. That latter procedure inserts X^* as a path in the tree. Moreover, it updates pointers (called *node links*) between each item and other items having the same name in the tree.

This prefix-tree construction process is illustrated with an example. Consider the temporal database of Table 1 and that $maxPer$, $maxSoPer$ and $minDur$ have been set to 3, 2 and 7, respectively. Figure 4(a)-(e) shows the construction of the LPP-tree after scanning the first, second, eighth and the whole temporal database. Notice that the timestamp of the eighth transaction is not added to the node c because this timestamp is part of any periodic time-interval of c (as shown in Figure 4 (c)). In a LPP-tree, an item header table is built so that each item points to its occurrences in the tree via a chain of node-links, to facilitate tree traversal. For simplicity, these node-links are not shown in Figure 4. These links are constructed and maintained as in the original FP-tree.

Mining the LPP-tree. After the proposed algorithm has built the LPP-tree, it does not need the original database anymore to find LPPs because all the important information for mining LPPs is stored in the LPP-tree. Starting from the LPPs having a single item, which have been stored in the LPP-list, the LPP-Growth algorithm explores the search space of itemsets using a depth-first search that has been

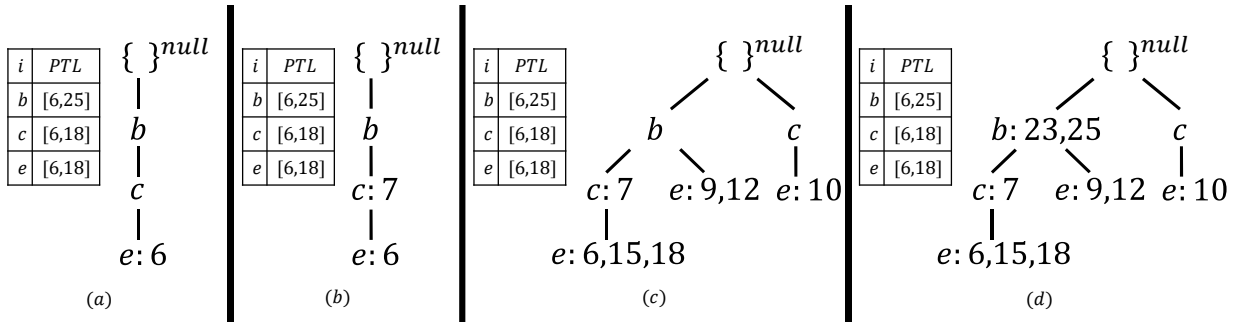

Figure 4: Construction of the LPP-tree for the database of Table 1. (a) After scanning the first transaction, (b) the second transaction, (c) the eighth transaction, and (d) the whole database.

Algorithm 7: Construction of a LPP-tree

input : TDB: a temporal database,
LPP-list: contains LPPs and their PTL

1 Create the root of the LPP-tree T, and label the root as "null";
2 **foreach** *transaction* X *and current timestamp* $ts \in TDB$ **do**
3 Sort single-item LPPs appearing in X using the same order as the *LPP-list*. Let this sorted set of items be called X^*;
4 $addTransaction(X^*, ts, T)$;
5 **end**

adapted from the FP-Growth algorithm to find LPPs.

The depth first search exploration is conducted by Algorithm 9. This algorithm takes as input the initial LPP-tree T, the user-defined thresholds, and an itemset α to be extended to find LPPs. Initially, the LPP-Growth procedure is called with $\alpha = \emptyset$ to search for all LPPs. The output of the algorithm is the set of all LPPs having α has prefix and their periodic time-intervals. As soon as the algorithm finds an LPP, the algorithm outputs it (e.g. by printing the pattern in the console, saving it to a file or keeping it in main memory).

The algorithm performs a loop on items stored in the header table of the LPP-tree in reverse order to process them one by one. For each item i, the algorithm considers the itemset $\beta = \alpha \cup \{i\}$, which was previously identified as a LPP. Thus, the algorithm outputs β and its periodic time-intervals stored in the LPP-tree. Then, the algorithm tries to find all items that could extend β to generate larger itemsets. Those items called γ are the ancestors of β in the LPP-tree. The algorithm traverses the node-links of i to collect the timestamps of all items in γ, and create a conditional pattern base of β. The conditional pattern base of β is the set of paths in the current LPP-tree that lead to i (excluding the nodes representing i). Then, the PTL of each item j in γ is calculated by calling the *time2interval* procedure to determine if $\beta \cup \{j\}$ is a LPP (has at least one periodic time-interval). The set of all such items is called γ'. If this set is not empty, the LPP-Growth algorithm recursively calls itself to output all LPPs of the form $\beta \cup \{j\}$ where $j \in \gamma'$ and to look for other LPPs that extend these patterns. The recursive call is done using a new LPP-tree T_β, which is created by inserting all paths of the conditional pattern base of β as transactions, and using γ' as items for its header table. Then, the item i is removed from the LPP-tree T and its timestamps are pushed to its parent node. The item i can be removed because it will not be needed when exploring the rest of the search space according to the processing order of items. Since the algorithm starts from single item LPPs and recursively explores the search space, all itemsets can be visited. And since the algorithm only prunes itemsets that have no periodic time-intervals from the search space (based on Property 2), the algorithm can find all LPPs.

To provide more details about the process of mining LPPs using the LPP-tree, a brief example is given.

Algorithm 8: addTransaction

input : X^*: sorted list of single–item LPPs,
ts: the timestamp, T: LPP-tree

```
1 N ← T.root;
2 foreach item i ∈ X* do
3 |   if N do not have a child N^i with name i then
4 |   |   Create a new node N^i. Let N^i be a child of N. Add a node-link pointer to the next node
  |   |     having the same item name i via the node-link structure;
5 |   end
6 |   N ← N^i;
7 end
8 add ts to N;
```

Consider the temporal database of Table 1 and that $maxPer$, $maxSoPer$ and $minDur$ are set to 3, 2 and 7, respectively. The prefix-tree T is initially built by scanning the database. The resulting LPP-tree is shown in Fig. 4, where the header table contains the items $\{b, c, e\}$, sorted in that order, and their PTLs. The LPP-Growth procedure is then called with this tree T and $\alpha = \emptyset$. The items from the header table are processed in reverse order. Thus, the LPP $\beta = \{e\}$ is first processed. It is output as a LPP with its PTL. Then, the next step is to evaluate whether extensions of β may be LPPs. For this purpose, the node-links of e are followed to quickly find all occurrences of e in the LPP-tree. The timestamps of all ancestors of e are collected. Those ancestors are $\gamma = \{b, c\}$. The conditional pattern base of $\{e\}$ is created, which is shown in the second line / first column of Table 6, and illustrated as a tree in Fig. 5 (b). Then, the *time2interval* function is called to find the PTLs of $\beta \cup \{b\}$ and $\beta \cup \{c\}$. These PTLs are shown in the second line / fourth column of Table 6. Because the PTL of $\beta \cup \{c\}$ is empty, that extension of β is not a LPP and does not need to be considered or extended. On the other hand, $\beta \cup \{b\}$ has a periodic time-interval $[6, 18]$, and is thus a LPP and its extensions must be considered. Thus, $\gamma' = \{b\}$. Then, the LPP-tree of β is built by applying the LPP-tree construction procedure on the paths from the conditional pattern base of β. The result is the LPP-tree $T^{\{b\}}$ shown in Fig. 5 (c) where the header table contains the item $\gamma' = b$. This tree is then recursively mined by calling the LPP-Growth procedure to output $\{b, e\}$ with its periodic time-interval $[6, 18]$ and explore its extensions. This recursion is not described for the sake of brevity. After the recursive call for mining the conditional-tree of $\{e\}$ returns, each node representing $\{e\}$ is deleted from the LPP-tree and its timestamps are pushed to its parent node. The result is shown in Fig. 5 (a). Then, the algorithm considers $\beta = \{c\}$. The conditional pattern base of $\{c\}$ is calculated, which is depicted in the third line / first column of Table 6. Here, $\gamma = \{b\}$. It is found that $\beta \cup \{b\}$ has no PTL and thus $\beta \cup \{b\}$ and its extensions do not need to be considered. Then, the algorithm considers $\beta = \{b\}$. The conditional pattern base of $\{b\}$ is empty and it has no ancestors. Thus, extensions of this itemset are also not considered. When the algorithm terminates, all LPPs with their PTLs have been found, which are shown in Table 4.

Pattern-growth algorithms are generally quite popular in the pattern mining literature [11, 10]. The main reason is that they only consider patterns that actually appear in the database because they scan the database (or a representation of the database such as a tree) to find itemsets. This is different from Apriori and Eclat based algorithms, which generate candidates by combining previously found itemsets, and can thus generate many candidates that do not exist in the database, thus waisting time to evaluate such patterns. However, pattern-growth algorithms such as those based on prefix-tree can be more complicated to implement, and it is not given that they perform better in all situations [15]. Moreover, in the context of LPP mining, these three types of approaches may have different behavior. Thus, the next section provides a detailed performance comparison of the three proposed algorithms.

Algorithm 9: The *LPP-Growth* algorithm

input : T: a LPP-tree, α: an itemset (initial value is $\emptyset$),
$maxPer$, $maxSoPer$, $minDur$: the user-specified thresholds
output: the set of LPPs and their periodic time-intervals

foreach *item i in T's header table T^{ht} starting from the last one* **do**
  $\beta \leftarrow \alpha \cup \{i\}$;
  Output β and its corresponding periodic time-intervals $T.PTL^{\beta}$;
  Let γ be the ancestor items of β in T;
  Traverse the node-link of i in T to construct β's conditional pattern base and collect the timestamps of each item in γ;
  Calculate the PTL of each item in γ by calling the *time2interval* procedure;
  $\gamma' = \{j | j \in \gamma \wedge |PTL^j| > 0\}$;
  if $\gamma' \neq \emptyset$ **then**
    Construct β's conditional LPP-tree T_β where $T_\beta^{ht} = \gamma'$;
    Call LPP-Growth(T_β, β);
  end
  Remove i from T;
end

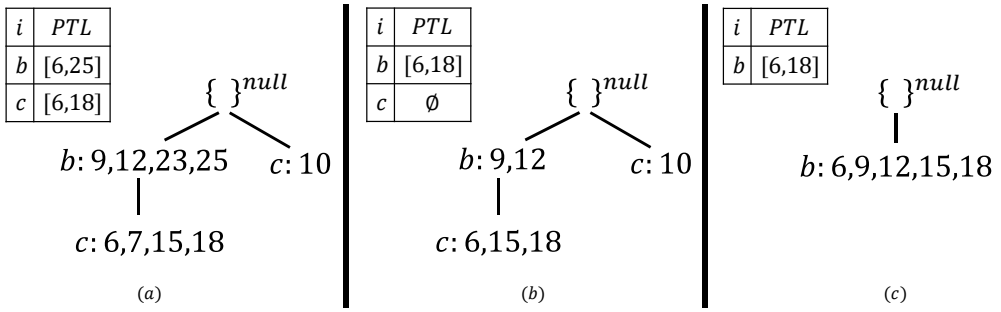

Figure 5: Mining LPPs using $\beta = \{e\}$. (a) LPP-tree after removing the item 'e', (b) Prefix-tree of $\beta = \{e\}$, (c) Conditional tree of $\beta = \{e\}$.

5. Experimental evaluation

This section describes the evaluation of the proposed algorithms to discover LPPs. It is divided in three parts. First, a performance evaluation is described in terms of runtime, number of patterns, memory consumption and scalability, when the algorithms are applied on benchmark datasets with different parameter settings. Second, an analysis of patterns found in real shopping data is presented, which shows that interesting patterns are found. Third, a performance comparison with the traditional PF-Growth [36] and PF-Growth++ [22] algorithms is presented.

The three proposed algorithms (LPPM$_{breadth}$, LPPM$_{depth}$ and LPP-Growth) and the compared algorithms are implemented in Java, and produce the same output for the same parameter settings and datasets. The performance of the three algorithms is however generally different because they utilize different data structures and search procedures. Briefly, LPPM$_{breadth}$ and LPPM$_{depth}$ both use a vertical database representation based on bit vectors, and a candidate generation approach, but the former applies a breadth first search while the latter performs a depth-first search. On the other hand, to avoid generating too many candidates, LPP-Growth applies a depth-first search on a horizontal database using a pattern-growth approach. Experiments were performed on a computer having a 64 bit Xeon E3-1270 3.6 Ghz CPU, running the Windows 10 operating system and 64 GB of RAM. Memory usage was measured using the standard Java API.

Table 6: Calculations when mining an LPP-tree with $\beta = \{e\}, \{c\}$, and $\{b\}$.

β	γ	Conditional Pattern Base	PTL	γ'	Conditional LPP-tree	LPPs
$\{e\}$	$\{b, c\}$	$\{bc$:6,15,18$\}$, $\{b$:9,12$\}$, $\{c$:10$\}$	$\{b$:[6,18]$\}$, $\{$c:$\emptyset\}$	$\{b\}$	$\{b$:6,9,12,15,18$\}$	$\{eb$:[6,18]$\}$
$\{c\}$	$\{b\}$	$\{b$:6,7,15,18$\}$	$\{b$:$\emptyset\}$	–	–	–
$\{b\}$	–	–	–	–	–	–

The algorithms were evaluated in terms of performance on benchmark synthetic (*T10I4D100K*) and real-world (*mushroom*, *kosarak* and *online retail*) datasets commonly used in the pattern mining literature, which have varied characteristics. They are available on the SPMF software website [6]. The transactions in the *T10I4D100K*, *mushroom* and *kosarak* datasets do not have timestamps. Therefore, the timestamp of each transaction was set to its transaction ID. For example, the database *T10I4D100K* contains 100,000 transactions, where the first transaction has the ID 1 and the timestamp 1, the second transaction has the ID 2 and the timestamp 2, ..., and the 100,000th transaction has the ID 100,000 and the timestamp 100,000. Thus, all consecutive transactions have a fixed time gap of one time unit. While *T10I4D100K*, *mushroom* and *kosarak* have synthetic timestamps, the *online retail* dataset is a shopping dataset with real timestamps. The time span is from 2010-12-1 8:26 to 2011-12-9 12:50. The smallest time unit in this database is minutes.

These datasets have been chosen because they are benchmark datasets that represent the main types of data encountered in real-life scenarios (dense, sparse and long records). The characteristics of all datasets are presented in Table 7. The notations $|D|$, $|I|$, T_{min}, T_{max} and T_{avg} represent the number of transactions, number of distinct items, minimum transaction length, maximum transaction length and the average transaction length, respectively. Moreover, for the convenience of the reader, Table 8 shows the first four records of each dataset, where items are represented as positive integers. For instance, the second transaction of the *online retail* dataset has the ID 536366. It indicates that at 8:28 AM on 2010/12/1 some customer has bought two items having the IDs 1650 and 1653. Those codes represents two products named HAND WARMER UNION JACK and HAND WARMER RED POLKA DOT, respectively. As it can be seen from this table, these datasets have very different characteristics. For example, while Kosarak has very short transactions, mushroom has very long and similar transactions. Testing the algorithms on these different types of data allow to assess their behavior in different situations.

In the following experiments, *maxSoPer*, *maxPer* and *minDur* are expressed as percentage of the size of the dataset. To ensure reproducibility of the experiments, the source code and datasets used in the experiments are available at: `http://www.philippe-fournier-viger.com/spmf/LPPGG/`.

Table 7: Characteristics of the datasets.

Dataset	$\|D\|$	$\|I\|$	T_{min}	T_{max}	T_{avg}	Type
T10I4D100K	100,000	870	1	29	10	sparse, many records
mushroom	8,124	119	23	23	23	dense, long records
kosarak	990,002	41,270	1	2,498	8	sparse, many items and records
online retail	23,260	4,224	1	1,108	22.7	sparse, long records

5.1. Evaluation and comparison of proposed algorithms

The first set of experiments was carried to compare the proposed algorithms in terms of runtime, memory usage, scalability, and also to study the influence of parameters on the number of patterns found.

Runtime performance. The runtime performance was first evaluated. Algorithms were run on each dataset, while varying parameter values. The goal is to compare the performance of algorithms in different situations and see the influence of parameters on performance. Generally, as *maxSoPer* and *maxPer* are

Table 8: The first four records of each dataset.

Dataset	TID	Transaction	Timestamp
T10I4D100K	1	25 52 164 240 274 328 368 448 538 561 630 687 730 775 825 834	1
T10I4D100K	2	39 120 124 205 401 581 704 814 825 834	2
T10I4D100K	3	35 249 674 712 733 759 854 950	3
T10I4D100K	4	39 422 449 704 825 857 895 937 954 964	4
mushroom	1	1 5 12 21 23 25 36 39 42 53 56 57 67 71 79 88 90 94 97 104 113 120 128	1
mushroom	2	1 5 12 21 23 25 36 39 42 53 56 57 67 71 79 88 90 94 97 104 108 120 128	2
mushroom	3	1 5 12 21 23 25 36 39 42 50 56 57 67 71 79 88 90 94 97 104 113 120 128	3
mushroom	4	1 5 12 21 23 25 36 39 42 50 56 57 67 71 79 88 90 94 97 104 108 120 128	4
kosarak	1	1 2 3	1
kosarak	2	1	2
kosarak	3	4 5 6 7	3
kosarak	4	1 8	4
online retail	536365	3918 913 3926 1910 3158 1529 2911	2010/12/1 8:26
online retail	536366	1650 1653	2010/12/1 8:28
online retail	536367	454 1833 1290 1803 568 2814 2833 1746 2034 244 4062 2743 2744 1082 573 574	2010/12/1 8:34
online retail	536368	319	2010/12/1 8:34

increased, and *minDur* is decreased, the size of the search space is expected to increase and more patterns may be found and runtimes may be longer. However, it is to be noted that since datasets have different characteristics, the performance of a same algorithm can vary greatly from one dataset to another for the same parameter values. For example, it is possible that millions of patterns are found for some parameter values on one dataset, while few patterns are found for the same parameter values on another dataset. For this reason, to be able to clearly see the behavior of the algorithms and the trends as parameters are varied, the parameters have been set differently on each dataset. The methodology for choosing the parameter values was the following. We tried various parameter values and found a set of values that would produce some patterns on each dataset. Then, we varied each parameter individually to evaluate its influence on performance, by decreasing *maxSoPer*, decreasing *maxPer* and increasing *minDur*, respectively. We stopped varying a parameter when algorithms became too long to execute, ran out of memory or a clear trend was observed. This methodology of using different parameters on different datasets to evaluate pattern mining algorithms has been used in numerous studies [8, 9, 12, 13, 16, 17, 19, 30, 33].

Figure 6 presents the runtimes of the three algorithms for different *maxSoPer*, *maxPer* and *minDur* values on the *T10I4D100K*, *mushroom* and *online retail* datasets, respectively. It can be observed that for the *T10I4D100K* and *online retail* datasets, LPP-Growth is faster than LPPM$_{breadth}$ and LPPM$_{depth}$, while on the *mushroom* dataset, LPP-Growth is slower. The reason is that the mushroom dataset is very dense and each item appears in many transactions. Thus, many LPPs have large time-intervals. Therefore, it takes a considerable amount of time to build conditional trees, which increases the overall runtime of LPP-Growth. A second reason is that the candidate generation approaches of LPPM$_{depth}$ and LPPM$_{breadth}$ are more efficient on dense datasets than on sparse datasets. This can be observed by calculating the percentage of candidate patterns that are LPPs. For the moderately sparse dataset $T10I4D100K$, up to 4% of candidates

generated by LPPM$_{depth}$ and LPPM$_{breadth}$ are LPPs, while on the dense Mushrooms dataset, up to 80% are LPPs. Thus, on dense datasets, LPPM$_{depth}$ and LPPM$_{breadth}$ are very efficient as most candidates that they generate are LPPs, while on sparse datasets, these algorithms spend a huge amount of time considering patterns that are not LPPs and may even not exist in the database. The LPP-Growth algorithms does not have this problem as it uses a pattern-growth approach that only consider patterns that exist in the database.

The second observation is that LPPM$_{depth}$ is faster than LPPM$_{breadth}$ in some cases, while in other cases, the runtimes are similar. Although, two strategies were integrated in LPPM$_{breadth}$ to improve its performance, LPPM$_{depth}$ is faster for low $minDur$ and $maxSoPer$ values, or high $maxPer$ values, that is when the search space becomes larger, and mainly for the sparse datasets such as *online retail*. On the other hand, on dense datasets such as *mushroom*, their performance is similar. A reason why LPPM$_{depth}$ is sometimes faster than LPPM$_{breadth}$ is that the former can group itemsets that have the same prefix together without using a map. Thus, it avoids the cost of inserting itemsets in or accessing itemsets in the map. Another reason is that the OTS strategy is used by LPPM$_{breadth}$ but not by LPPM$_{depth}$. This strategy can reduce memory as it will be explained because multiple ts-lists do not need to be kept in memory but it can increase the runtime for calculating ts-lists of long itemsets. Because the performance difference occurs mainly for very low/large threshold values for these two algorithms, it can be expected that these two algorithms have similar performance for a normal use case where selected threshold values are not too small or large. For the influence of different parameters, it can be observed in Figure 6 that increasing $maxSoPer$, $maxPer$ or decreasing $minDur$ can increase runtimes. The reason is that increasing $maxSoPer$, $maxPer$ or decreasing $minDur$, increases the size of the search space.

On overall, it can be observed that the runtimes of the three algorithms increase more or less quickly when a parameter is varied and this depends on the dataset. Generally, LPP-Growth performs best on sparse datasets, on which its runtime increases more slowly than those of LPPM$_{breadth}$ and LPPM$_{depth}$ as the search space becomes larger. But on dense datasets, LPPM$_{breadth}$ and LPPM$_{depth}$ perfoms best and their runtimes increase more slowly than that of LPP-growth. These results are somewhat in accordance with results reported in Section 2 about frequent itemset mining, where FP-growth was found to be generally faster on sparse datasets and low minimum support threshold than Apriori-TID and Eclat [15].

Pattern count. In a second experiment, the number of LPPs was evaluated when the algorithms' parameters were varied using the same values as in the first experiment. Results are shown in Figure 7 for the $T10I4D100K$, *mushroom* and *online retail* datasets, respectively. It can be observed that increasing $maxSoPer$, $maxPer$ or decreasing $minDur$ can increase the number of LPPs. It is also observed that these measures have a similar influence on the number of LPPs found than on the algorithms' runtimes. This is because varying these parameters influences the size of the search space, which grows more or less proportionally to the number of patterns.

Scalability. In a third experiment, we assessed the scalability of the three proposed algorithms in terms of execution time and number of patterns found when varying the number of transactions in the database. For this experiment, we used the real *kosarak* dataset since it is a huge sparse dataset with a large number of distinct items and transactions (see Table 7). The dataset was divided into five parts, and algorithms were ran on the first part, the first two parts, the three first parts, the four first parts, and the whole database. For these experiments, the parameters were fixed as $maxPer = 0.2\%$, $minDur = 20\%$ and $maxSoPer = 0.2\%$ of $|kosarak|$ to only evaluate the influence of the database size. Figure 8 shows the experimental results. The lines represent the execution times of the three algorithms, while the bars indicates the number of patterns found. It can be observed that the runtime and number of patterns increase in a similar way with the database size. Because when the $maxPer$ and $maxSoPer$ thresholds are increased, the constraints are relaxed, the search space is larger, the algorithms can find more patterns, and thus take more time.

Memory consumption. Memory consumption was evaluated next. Table 9 shows the peak memory consumption of the algorithms for different parameter values on the *online retail* dataset. Results on other datasets are similar. In this experiment, thresholds are expressed as numbers rather than percentage. Several observations can be made. First, the LPP-Growth algorithm is the most memory efficient, consuming always at least three times less memory than the other algorithms. This is because it uses a compact tree structure for representing the database, and timestamps are only saved in **tail** nodes. Second, LPPM$_{breadth}$ often

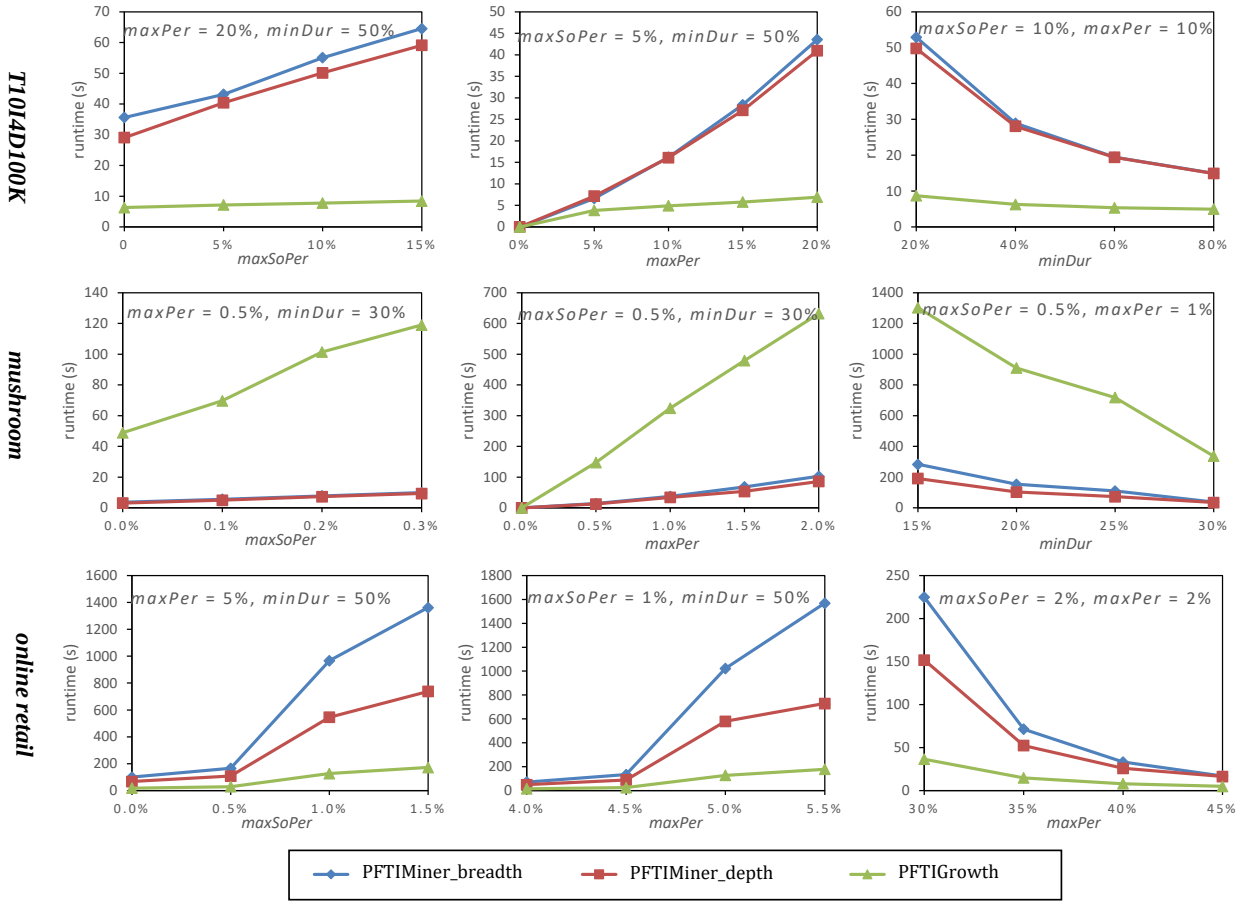

Figure 6: Runtime for different parameter values.

consumes less memory than LPPM$_{depth}$ when the two strategies (OTS and SPM) are applied. Thus, this shows that although LPPM$_{depth}$ is usually faster than LPPM$_{breadth}$, the latter has the advantage of using less memory. Third, when the *minDur* parameter is increased, memory consumption of these three algorithms decreases. Four, when the *maxPer* or *maxSoPer* thresholds are increased, the memory consumption of these three algorithms generally increases but may also slightly decrease in some cases. The reason why it can decrease is that the total number of periodic time-intervals may decrease when many short time-intervals can be combined into a long time-interval by increasing *maxPer* or *maxSoPer*, which can reduce the memory consumption.

Most of the memory usage of LPPM$_{depth}$ and LPPM$_{breadth}$ is used for storing the bit vectors. And these bit vectors may have a large size for datasets containing many transactions. To further reduce the memory usage of these algorithms, several strategies could be employed in future work. Inspiration could be taken from the DCI_Closed [26] algorithm, which proposed several optimizations for manipulating bit vectors such as (1) integrating a mechanism to reuse results of previous bitwise comparisons, (2) reconstructing the bitvectors during a depth-first search by eliminating all columns (transactions) that only contain 0s, and (3) reordering transactions to ensure that 1s are typically close to each other in bit vectors. Another technique used by the DBV-Miner algorithm is to apply bit vector compression techniques to remove long strings of consecutive 0s [37]. Such optimization can be very effective on sparse datasets. Techniques could also be considered to perform memory buffering (reusing memory space for bit vectors rather than allocating new memory) [7]. However, implementing bit vectors and memory optimizations can be complex. Thus, it will be considered in future work.

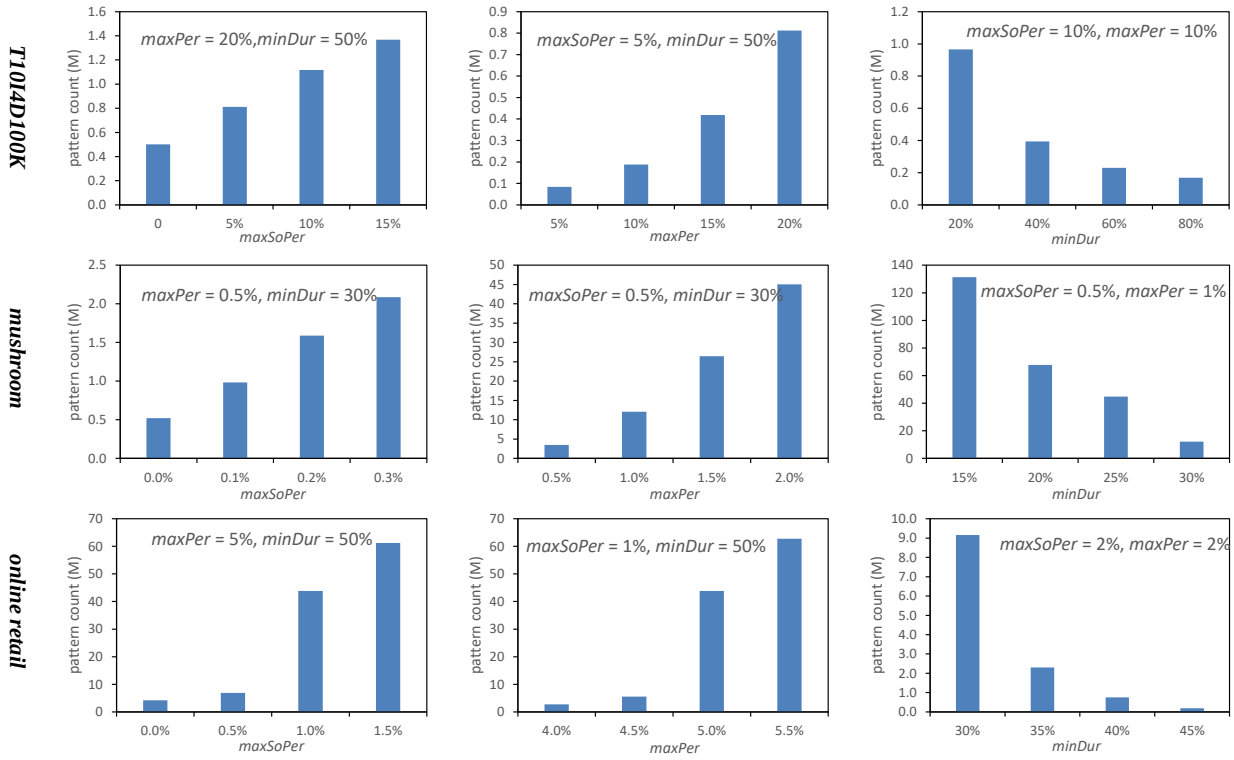

Figure 7: Number of patterns for different parameter values.

Influence of the SPM strategy. To evaluate the influence of the SPM strategy, another experiment was carried out where the runtime and memory usage of $LPPM_{breadth}$ was compared for various $maxSoPer$ values with that of $LPPM_{breadth}$ without the SPM strategy. This version is denoted as $LPPM_{breadth}$*. Figure 9 shows the runtime and peak memory usage of $LPPM_{breadth}$ with and without the SPM strategy, for the *kosarak* and *T10I4D100K* datasets. In these charts, lines represent runtimes and bars indicate peak memory usage. It can be observed that runtimes and peak memory usage of $LPPM_{breadth}$ are considerably improved when the SPM strategy is utilized. In particular, $LPPM_{breadth}$ consumed more than three times less memory than $LPPM_{breadth}$* on *kosarak*, and was more than five time faster on *T10I4D100K*. The reason for this better performance is that all itemsets having a same prefix are stored together, and hence, itemsets can be directly combined to generate candidates without performing unecessary comparisons between itemsets. If the SPM strategy is not used, all itemsets must be compared to identify those having a same prefix. The above experimental results are in line with our expectation when designing the SPM strategy. Results are similar on other datasets.

5.2. Pattern analysis.

To evaluate the usefulness of the proposed model, patterns found in the real *online retail* dataset of customer transactions were analyzed. The goal of this experiment is to assess if interesting patterns are found that provide insights on the shopping behavior of customers. To find patterns, parameters were adjusted and it was found that for $maxPer = 1$ (day), $maxSoPer = 2$ (days) and $minDur = 30$ (days) yield several interesting patterns. Some of those are depicted in Table 10. The first pattern indicates that the sale of a paper chain kit and vintage sewing kit becomes periodic from the 27th October until the 9th December, which correspond to the time period for buying Christmas gifts. It is interesting that this pattern is periodic during only a short part of the whole year, and thus this pattern would be ignored by traditional periodic frequent pattern mining algorithms that focus on finding patterns that are periodic during the whole year. The second pattern is about the sale of party items from the 12nd July until the 12nd August, which

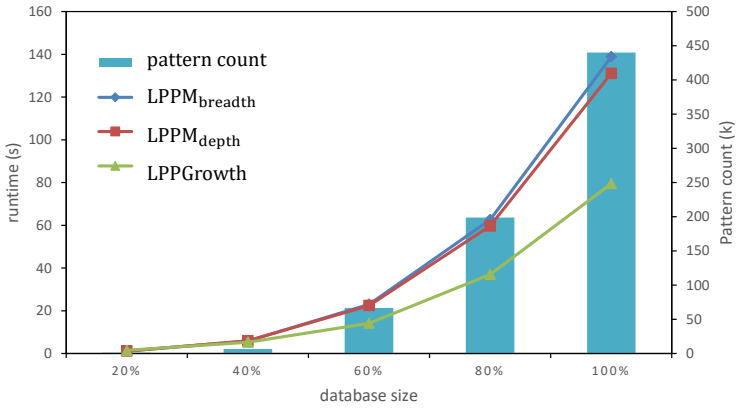

Figure 8: Runtime and number of patterns found when varying the database size.

Table 9: Comparison of peak memory usage (MB).

Thresholds			Algorithms		
maxPer	*maxSoPer*	*minDur*	LPPM$_{breadth}$	LPPM$_{depth}$	LPP-Growth
1	1	40	943	729	215
1	1	60	627	639	174
1	1	90	528	536	163
1	2	40	1,618	2,071	250
1	3	40	1,576	4,534	280
2	1	40	4,190	4,537	277
3	1	40	4,331	5,437	545

is reasonable given that many parties are held during the summer in western countries, while the pattern is not periodic during the rest of the year. The third pattern is cake pantries, which becomes periodic before Christmas and one month before Easter. This pattern has multiple different perodic time-intervals.

To further analyse these results, Fig. 10 shows the *soPer* values of the third pattern over time. The X-axis indicates the real timestamps while the Y-axis indicates the *soPer* value of the pattern at each timestamp where it appears. Furthermore, the start and end points of the two periodic time-intervals are indicated with stars. Note that to reduce the range of values of *soPer* and make the figure easier to visualize, *soPer* values larger than 2.5 (days) have been set to 2.5 (days) and the end point of a time-interval is also set to 2.5 (days). From Fig. 10, it can seen that although there are several time-intervals where *soPer* values are less than *maxSoPer* (2 days), only two periodic time-intervals have durations larger than *minDur* (30 days). This chart also illustrates that the periodic behavior of customers can vary considerably in real data over time, and it is thus worthy to identify these time-intervals where patterns are locally periodic.

On overall, it is observed that the algorithms have reasonable performance and can find interesting patterns in real-life data that cannot be found using traditional models, which focus on periodicity in the whole database. The proposed algorithms can discover patterns in non-predefined time-intervals, and each algorithm was found to perform best in some situations. Several possibilities could be considered for future work such as considering other types of local periodic patterns, streaming data, and design methods to help select parameters automatically.

5.3. Comparison with a traditional PFP mining algorithm.

Lastly, additional experiments were carried to compare the performance of LPP-Growth with traditional PFP mining algorithms. It should be noted that the proposed algorithms are designed for a new problem of mining local periodic patterns, which is different from the traditional problem of PFP mining. Thus, the

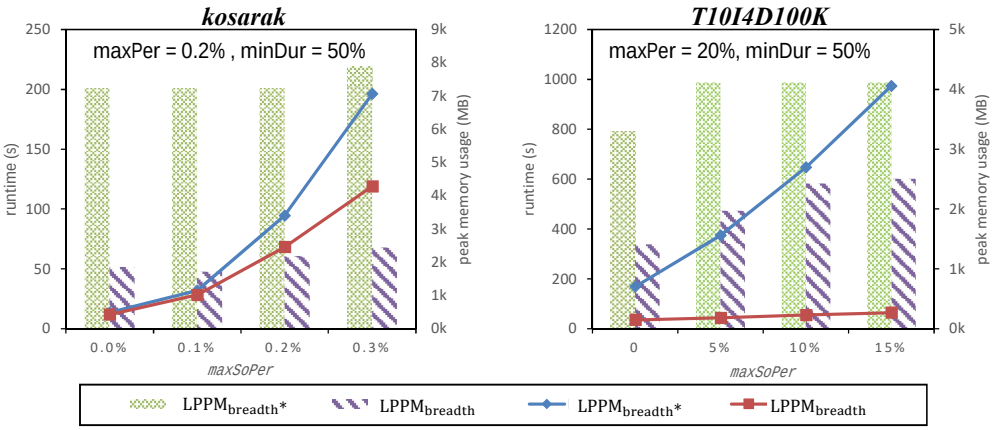

Figure 9: Runtime and peak memory usage of $LPPM_{breadth}$ with and without the SPM strategy.

Table 10: Some interesting local periodic patterns found in the *online retail* dataset.

Patterns	Time-intervals
PAPER CHAIN KIT 50'S CHRISTMAS, VINTAGE DOILY TRAVEL SEWING KIT	[2011-10-27 13:33, 2011-12-09 12:50]
PARTY BUNTING, SET OF 3 REGENCY CAKE TINS	[2011-07-12 12:24, 2011-08-12 16:30]
RECIPE BOX PANTRY YELLOW DESIGN, SET OF 3 CAKE TINS PANTRY DESIGN	[2011-02-13 12:03, 2011-04-03 14:42], [2011-10-24 16:17, 2011-12-09 12:50]

size of the search space of the two problems is not the same. In particular, the proposed algorithms must consider and detect multiple time-intervals, which increases the size of the search space.

To be able to compare the proposed algorithms with traditional PFP mining algorithms [36, 22], we modified the measures used to find local periodic patterns as follows. First, we set *maxDur* to 100% to consider the size of the whole database. Second, we set the start point as the timestamp of the first transaction for detecting patterns in the whole database. Third, because we do not need to detect multiple time-intervals, we modified the way of detecting time-intervals, and changed the initial value of *soPer* from *maxPer* to 0. We use the name LPP*-Growth algorithm to refer to the modified version of LPP-Growth, and the name LPP* to refer to the patterns found by this modified version. The compared algorithms from the literature for mining periodic frequent patterns are PF-Growth [36] and PF-Growth++ [22]. The algorithms have been compared in terms of runtime, number of patterns found, memory consumption, and patterns were analyzed.

Runtime. Figure 11 shows the runtime requirements of the LPP*-Growth algorithm for different *minSup* and *maxPer* values on *mushroom* and $T10I4D100K$, respectively. In these charts, there are two lines representing LPP*-Growth algorithms with different *maxSoPer* values. The following observations are drawn from Figure 11. Increasing *minSup* or decreasing *maxPer* often decreases the runtime. The reason is that increasing *minSup* or decreasing *maxPer* increases the search space to mine patterns. The second observation is that the two algorithms to mine PFP are faster than algorithms to mine LPP*. This is reasonable because the *maxSoPer* measure relaxes the constraint set by *maxPer*, which dynamically increases the range of *maxPer* values accepted for patterns.

Pattern count. Figure 12 shows the number of patterns generated by the algorithms for different *minSup* and *maxPer* values, respectively. In the left chart, the *maxPer* threshold is set to 10% while in the right chart, the *minSup* threshold is set to 10%. The notation PFP refer to using the PF-Growth or PF-Growth++ algorithms to mine periodic frequent patterns. The following observations are made. Increasing *minSup* may decrease the number of patterns found. The reason is that some patterns will then fail to

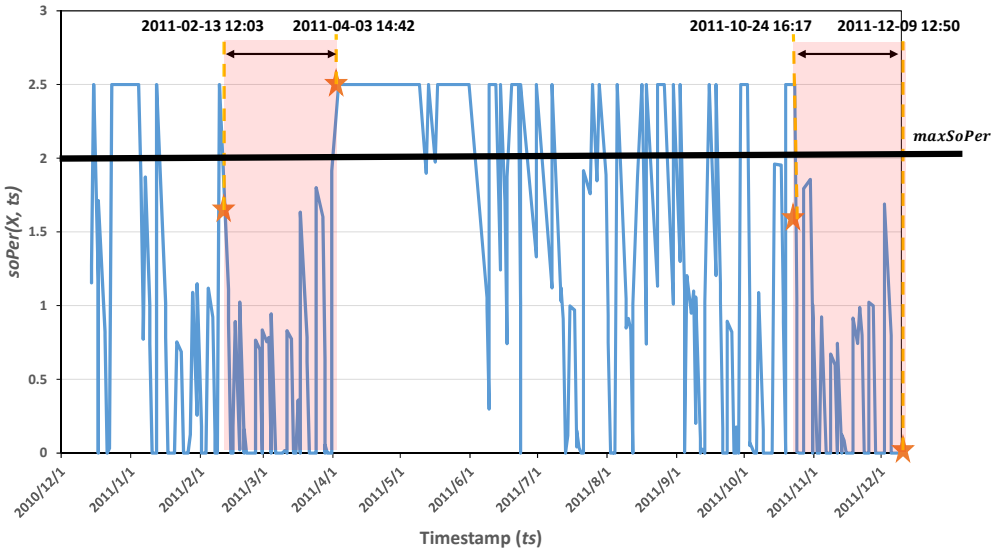

Figure 10: The periodic time-intervals of the local periodic pattern {RECIPE BOX PANTRY YELLOW DESIGN, SET OF 3 CAKE TINS PANTRY DESIGN} found in the *online retail* dataset.

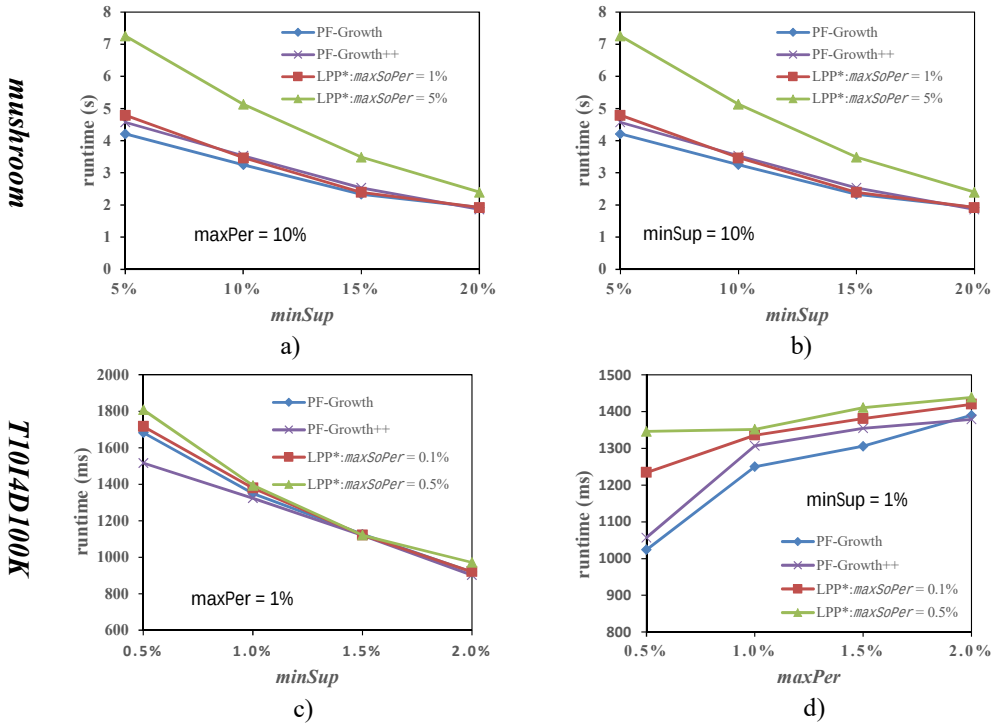

Figure 11: Runtime for different parameter values. a) various *minSup* values on *mushroom*, b) various *maxPer* values on *mushroom*, c) various *minSup* values on $T10I4D100K$, d) various *maxPer* values on $T10I4D100K$.

satisfy the higher *minSup* threshold constraint. Similary, increasing *maxPer* may increase the number of patterns found. The reason is that as *maxPer* is increased, frequent patterns having longer periods may become periodic frequent patterns. And LPP*-Growth will find more patterns, because the traditional PFP algorithms eliminate many patterns due to the strict *maxPer* constraint.

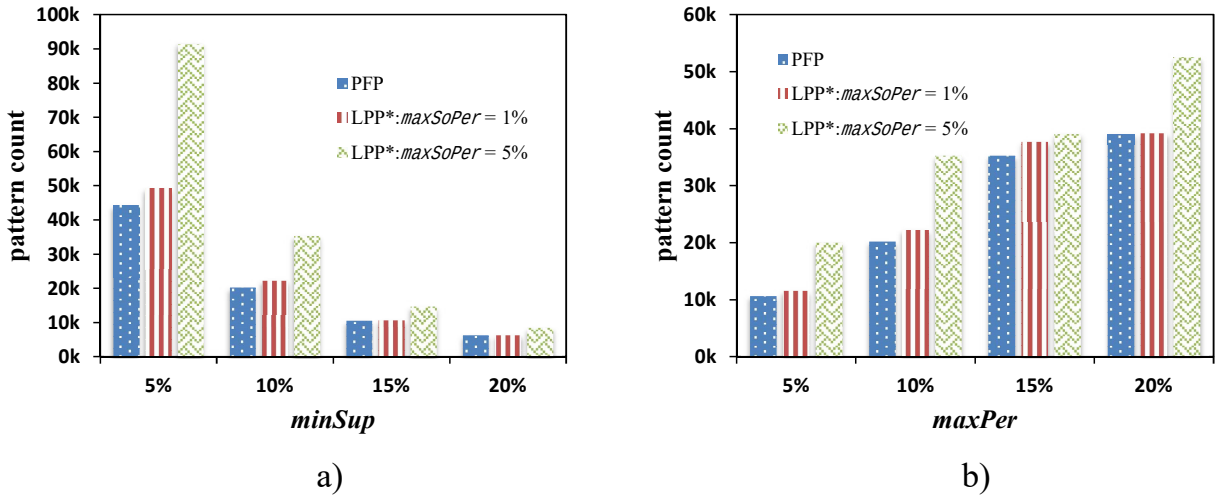

Figure 12: Number of patterns for different parameter values on *mushroom* dataset. a) various *minSup* values, b) various *maxPer* values.

Memory consumption. In Table 11, the peak memory usage of LPP*-Growth and PF-Growth were recorded for different parameter values on $T10I4D100K$. Note that we fixed $maxSoPer = 0.5\%$ to run LPP*-Growth. Results show that the algorithms consumes less memory for high *minSup* and low *maxPer* values. This is reasonable as more patterns can be pruned. To find patterns, LPP*-Growth creates more nodes in its tree structure than PF-Growth, which also means that more memory is needed.

Table 11: Comparison of peak memory usage.

Thresholds		Peak memory usage (MB)		Node count	
minSup	*maxPer*	LPP*-Growth	PFP-Growth	LPP*-Growth	PFP-Growth
1%	1%	296	296	515,122	514,153
0.5%	1%	396	324	641,214	596,351
1%	0.5%	295	268	514,153	334,153

6. Conclusion

This paper has presented a novel problem of mining local periodic patterns. Two novel measures named *maxSoPer* and *minDur* have been defined to assess the periodic interestingness of an itemset in a discrete sequence (a temporal database). Properties of the problem have been studied and it was shown that LPPs satisfy an anti-monotonic property (Lemma 2). Three algorithms, named $LPPM_{breadth}$, $LPPM_{depth}$ and *LPP-Growth*, have been proposed to efficiently find LPPs. They respectively adopt a breadth-first search, depth-first search and pattern-growth approach by extending the *Apriori-TID*, *Eclat* and *FP-Growth* algorithms. We have performed an experimental evaluation on both real and synthetic datasets to evaluate the performance of these algorithms. Results show that these algorithms have their own advantages and limitations, and the proposed model can discover useful information in real customer transaction data. Moreover, a comparison was made with a traditional PFP mining algorithms although the proposed problem is different from the traditional one.

This is the first paper on mining LPPs with their periodic time-intervals. For future work, one could design more efficient algorithms in terms of runtime and memory consumption. Moreover, it would be interesting to adapt the proposed algorithms for big data frameworks, or using GPUs, multi-thread, and

other technologies for high performance computing. This could allow processing larger databases in less time. Another interesting possibility is to extend the proposed model to process non static databases such as incremental databases and data streams. Moreover, another possibility is to extend the proposed model to other types of patterns such as sequential patterns and rules [10], or to design methods to set parameters automatically.

References

[1] Agrawal, R., Imielinski, T., Swami, A.N.: Mining Association Rules Between Sets of Items in Large Databases. In: Proc. 1993 ACM SIGMOD Int. Conf. on Management of Data, pp. 207–216 (1993)

[2] Agrawal, R., Srikant, R.: Fast algorithms for mining association rules. In: Proc. 20th Int. Conf. Very Large Data Bases, pp. 487–499 (1994)

[3] Agrawal, R., Srikant, R.: Mining sequential patterns. In: Proc. 11th Int. Conf. on Data Engineering, pp. 3–14 (1995)

[4] Bailey, J., Manoukian, T., Ramamohanarao, K.: Fast algorithms for mining emerging patterns. In: Proc. 6th European Conf. on Principles of Data Mining and Knowledge Discovery, pp. 39–50 (2002)

[5] Dong, G. , Li, J.: Efficient mining of emerging patterns: Discovering trends and differences. In: Proc. 5th ACM SIGKDD Int. Conf. on Knowledge Discovery and Data Mining, pp. 43–52 (1999)

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

[7] Fournier-Viger, P., Li, J., Lin, J.C., Chi, T.T., Kiran, R.U.: Mining Cost-Effective Patterns in Event Logs. Knowledge-Based Systems, Elsevier, 191: 105241 (2019)

[8] Fournier-Viger, P., Lin, J.C.W., Duong, Q.H., Dam T.L.: PHM: Mining Periodic High-Utility Itemsets. In: Proc. 16th Industrial Conf. on Data Mining, pp. 64–79 (2016)

[9] Fournier-Viger, P., Lin, J.C.W., Duong, Q.H., Dam, T.L., Ševčík, L., Uhrin, D., Voznak, M.: PFPM: discovering periodic frequent patterns with novel periodicity measures. In: Proc. 2nd Czech-China Scientific Conf. pp. 27–38 (2017)

[10] Fournier-Viger, P., Lin, J.C.W., Kiran, R.U., Koh, Y.-S.: A Survey of Sequential Pattern Mining, Data Science and Pattern Recognition, 1(1), pp. 54–77 (2017)

[11] Fournier-Viger, P., Lin, J.C.W., Vo, B., Chi, T. T., Zhang, J., Le, H.B.: A Survey of Itemset Mining. WIREs Data Mining and Knowledge Discovery, Wiley, e1207 doi: 10.1002/widm.1207 (2017)

[12] Fournier-Viger, P., Wu, C.-W., Zida, S., Tseng, V.S.: FHM: Faster high-utility itemset mining using estimated utility co-occurrence pruning. In: Proc. 21st Int. Symp. on Methodologies for Intelligent System, pp. 83–92 (2014)

[13] Fournier-Viger, P., Zhang, Y., Lin, J.C.W., Fujita, H., Koh, Y.S.: Mining Local and Peak High Utility Itemsets. Information Sciences, Elsevier, 481, pp. 344–367 (2019)

[14] Gama, J., Zliobaite, I., Bifet, A., Pechenizkiy, M., Bouchachia, H.: A Survey on Concept Drift Adaptation. Journal of ACM Computing Surveys, ACM, 46(4), pp. 1–37 (2014)

[15] Goethals, B., Zaki, M.J.: Advances in frequent itemset mining implementations: report on FIMI'03. ACM SIGKDD Explorations Newsletter, ACM, 6(1), pp.109–117 (2004)

[16] Gouda, K., Zaki, M.J.: Efficiently mining maximal frequent itemsets. In: Proc. 1st IEEE Int. Conf. on Data Mining, pp. 163–170 (2001)

[17] Hackman, A., Huang, Y., Tseng, V.S.: Mining Trending High Utility Itemsets from Temporal Transaction Databases. In: Proc. 29th Int. Conf. on Database and Expert Systems Applications, pp. 461–470 (2018)

[18] Han, J., Cheng, H., Xin, D.: Frequent pattern mining: current status and future directions. Data Mining and Knowledge Discovery, 15 (1), pp. 55–86 (2007)

[19] Han, J., Pei, J., Yin, Y.: Mining Frequent Patterns without Candidate Generation. In: Proc. 2000 ACM SIGMOD Int. Conf. on Management of Data, pp. 1–12 (2000)

[20] Hipp, J., Güntzer, U., Nakhaeizadeh, G.: Algorithms for association rule mining – a general survey and comparison. ACM sigkdd explorations newsletter, ACM, 2(1), pp.58–64 (2000)

[21] Kiran, R.U., Kitsuregawa, M.: Discovering Quasi-Periodic-Frequent Patterns in Transactional Databases. In: Proc. 2nd Int. Conf. on Big Data Analytics. pp. 97–115 (2013)

[22] Kiran, R.U. , Kitsuregawa, M., Reddy, P.K.: Efficient discovery of periodic-fre- quent patterns in very large databases. In: Journal of Systems and Software, 112, pp. 110–121 (2016)

[23] Kiran, R.U., Shang, H., Toyoda, M., Kitsuregawa, M.: Discovering Recurring Patterns in Time Series. In: Proc. 18th Int. Conf. on Extending Database Technology, pp. 97–108 (2015)

[24] Kiran, R.U., Venkatesh, J.N., Fournier-Viger, P., Toyoda, M., Reddy, P.K., Kitsuregawa, M.: Discovering Periodic Patterns in Non-uniform Temporal Databases. In: Proc. 21st Pacific-Asia Conf. on Knowledge Discovery and Data Mining, pp. 604–617 (2017)

[25] Liu, Y., Liao, W., Choudhary, A.: A two-phase algorithm for fast discovery of high utility itemsets. In: Proc. 9th Pacific-Asia Conf. on Knowl. Discovery and Data Mining, pp. 689–695 (2005)

[26] Lucchese, C., Orlando, S., Perego, R.: Fast and memory efficient mining of frequent closed itemsets. IEEE Transactions on Knowledge and Data Engineering, IEEE, 18(1), pp. 21–36 (2005)

[27] Minato, S.I., Uno, T., Arimura, H.: Lcm over ZBDDs: Fast generation of very large-scale frequent itemsets using a compact graph-based representation. In: Proc. 12th Pacific-Asia Conf. Knowledge Discovery and Data Mining, pp. 234–246 (2008)

[28] Muthukrishnan, S., Berg, E.V.D., Wu, Y.: Sequential Change Detection on Data Streams. In: Proc. 7th IEEE Int. Conf. on Data Mining Workshops, pp. 551–550 (2007)
[29] Pasquier, N., Bastide, Y., Taouil, R., Lakhal, L.: Discovering frequent closed itemsets for association rules. In: Proc. 7th Int. Conf. on Database Theory, pp. 398–416 (1999)
[30] Pei, J., Han J., Lu, H., Nishio, S., Tang, S., Yang, D.: H-Mine: fast and space-preserving frequent pattern mining in large databases. IIE Transactions, 39 (6), pp. 593–605 (2007)
[31] Pei, J., Han, J., Mortazavi-Asl, B., Pinto, H., Chen, Q., Dayal, U., Hsu, M.: PrefixSpan: Mining Sequential Patterns by Prefix-Projected Growth. In: Proc. 17th Int. Conf. on Data Engineering, pp. 215–224 (2001)
[32] Rasheed, F., Alhajj, R.: Framework for Periodic Outlier Pattern Detection in Time-Series Sequences. IEEE Transactions on Cybernetics, 44(5), pp. 569–582 (2014)
[33] Rashid, M.M., Karim, M.R., Jeong, B.S., Choi, H.J.: Efficient Mining Regularly Frequent Patterns in Transactional Databases. In: Proc. 17th Int. Conf. on Database Systems for Advanced Applications, pp. 258–271 (2012)
[34] Stormer, H.: Improving e-commerce recommender systems by the identification of seasonal products. In: Proc. 22nd Conf. on Artificial Intelligence, pp. 92–99 (2007)
[35] Surana, A., Kiran, R.U., Reddy, P.K.: An Efficient Approach to Mine Periodic-Frequent Patterns in Transactional Databases. In: Proc. 16th Pacific-Asia Conf. on Knowledge Discovery and Data Mining, pp. 254–266 (2012)
[36] Tanbeer, S.K., Ahmed, C.F., Jeong, B.S., Lee, Y.K.: Discovering periodic-frequent patterns in transactional databases. In: Proc. 13th Pacific-Asia Conf. on Knowledge Discovery and Data Mining, pp. 242–253 (2009)
[37] Vo, B., Hong, T.P., Le, B.: DBV-Miner: A Dynamic Bit-Vector approach for fast mining frequent closed itemsets. Expert Systems with Applications, Elsevier, 39(8), pp.7196–7206 (2012)
[38] Wan, Q., An, A.: Discovering Transitional Patterns and Their Significant Milestones in Transaction Databases. IEEE Transactions on Knowledge and Data Engineering, 21(12), pp. 1692–1707 (2009)
[39] Weiss, G.M.: Mining with rarity: a unifying framework. In: SIGKDD Exploration, 6 (1), pp. 7–19 (2004)
[40] Zaki, M.J., Gouda, K.: Fast vertical mining using diffsets. In: Proc. 9th ACM SIGKDD Intern. Conf. Knowledge Discovery and Data Mining, pp. 326–335 (2003)
[41] Zhang, C., Liu, C., Zhang, X., Almpanidis, G.: An up-to-date comparison of state-of-the-art classification algorithms. Expert Systems with Applications, 82, 128–150 (2017)
[42] Zhang, W., Yoshida, T., Tang, X., Wang, Q.: Text clustering using frequent itemsets. Knowl. Based Systems, 23(5), pp. 379–388 (2010)
[43] Zhang, M., Kao, B., Cheng, D.W., Yip, K.Y.: Mining periodic patterns with gap requirement from sequences. ACM Transactions on Knowledge Discovery from Data. 1(2), pp. 7–46 (2007)