Noname manuscript No.
(will be inserted by the editor)

TSPIN: Mining Top-k Stable Periodic Patterns

Philippe Fournier-Viger · Ying Wang · Peng Yang · Jerry Chun-Wei Lin · Unil Yun · Rage Uday Kiran

Received: date / Accepted: date

Abstract Discovering periodic patterns consists of identifying all sets of items (values) that periodically co-occur in a discrete sequence. Although traditional periodic pattern mining algorithms have multiple applications, they have two key limitations. First, they consider that a pattern is not periodic if the time difference between two of its successive occurrences is greater than a *maxPer* threshold. But this constraint is too strict, as a pattern may be discarded based on only two of its occurrences, although it may be usually periodic. Second, traditional algorithms use a constraint that the support (occurrence frequency) of a pattern must be no less than a *minSup* threshold. But setting that parameter is not intuitive. Hence, it is usually set by trial and error, which is time-consuming. This paper addresses the first limitation by introducing a concept of stability to find periodic patterns that have a stable periodic behavior. Then, the second limitation is addressed by proposing an algorithm named TSPIN (Top-k Stable Periodic pattern mINer) to find the top-k stable periodic patterns, where the user can directly specify the number of patterns k to be found rather than using the *minSup* threshold. Several experiments have been performed to assess TSPIN's performance, and it was found that it is efficient and can discover patterns that reveal interesting insights in real data.

Keywords Periodic patterns · Stability · Top-k patterns · Sequence

1 Introduction

A popular sub-field of data mining is pattern mining [34]. It consists of applying algorithms to discover patterns in data that satisfy some user requirements. Pattern mining is generally an unsupervised process that is done to reveal patterns that may help to understand the data and/or support decision-making. One of the most popular pattern mining tasks is Frequent Itemset Mining (FIM) [2,5,6,15,22,23,35,51,52,34], which aims at identifying all itemsets (sets of values or symbols) that appear frequently in

Philippe Fournier-Viger
School of Humanities and Social Sciences, Harbin Institute of Technology (Shenzhen), Shenzhen, China
E-mail: philfv8@yahoo.com

Y. Wang
School of Computer Science and Technology, Harbin Institute of Technology (Shenzhen), Shenzhen, China
E-mail: iwangying_919@163.com

P. Yang
School of Computer Science and Technology, Harbin Institute of Technology (Shenzhen), Shenzhen, China
E-mail: pengyeung@163.com

J. C. W. Lin
Department of Computing, Mathematics and Physics, Western Norway University of Applied Sciences (HVL), Bergen, Norway
E-mail: jerrylin@ieee.org

U. Yun
Department of Computer Engineering, Sejong University, Seoul, Republic of Korea
E-mail: unilyun@gmail.com

R. U. Kiran
The University of Tokyo, Tokyo, Japan
E-mail: uday rage@tkl.iis.u-tokyo.ac.jp

data. The input is a transaction database and a minimum support (*minSup*) threshold. The output is the set of all frequent itemsets, that is sets of values having a support (occurrence frequency) that is no less than the *minSup* threshold. FIM has been applied in many domains such as image classification, bioinformatics, network traffic analysis, e-learning, activity monitoring and malware detection [15,34]. However, selecting itemsets based on their frequency is useful but it may still reveal many patterns that are uninteresting to users. For example, analyzing a customer transaction database may reveal that some customers have bought milk and bread together many times but it does not say much about the context of these purchases. To find patterns that are more tailored to the needs of users for various applications, additional criteria have been considered and other types of patterns were studied. For example, it was proposed to mine condensed representations of patterns such as maximal frequent itemsets [21] and closed frequent itemsets [41] to obtain a summary of all frequent itemsets. Moreover, other variations of the FIM task were considered such as to mine the most profitable patterns [14,18, 47,49,50], itemsets when considering uncertainty [33,46], rare patterns [31], frequent episodes [24,19, 36], sequential patterns [13] and periodic frequent patterns [3,4,12,16,27,28,30,32,39,40,42,43,45].

Itemsets that are frequent and periodically appear in a database are called *periodic-frequent patterns* (PFP). For instance, a periodic frequent pattern found in transactions made by a customer may be $\{wine, cheese, bread\}$, indicating that he periodically buys wine, cheese and bread together, every four transactions or less. Such periodic pattern can be very interesting as it reveals some habit that a customer repeats over time. This pattern could be used for marketing [12,45]. For instance, based on this pattern, a retailer could send a text message every week to the customer to remind him of buying these items and offer him a discount. Besides market basket analysis, there are many other domains where it is interesting to find events that periodically re-appear in a sequence of events, or more generally in a sequence of symbols. For instance, periodic patterns have been discovered in activity sequences to perform activity monitoring [26]. A periodic pattern found in an activity sequence can be for example that someone always prepare coffee, eat breakfast, and brush his teeth. Other examples of applications are to analyse GPS trajectories [53], finding patterns in patient data for health management [25], in location sequences to perform user location prediction [48]. in spatial sensor data to understand how pollution is spreading [29], and to improve the performance of recommender systems [9]. While some studies have considered discovering PFPs in a database of sequences [8,16], most studies have considered analyzing a single sequence of events (sets of items), called a transaction database [3,4,11,12,27,30,45].

The goal of PFP mining in a sequence of events is to find some sets of events (itemsets) that repeatedly appear. This is done by looking at the periods of patterns to find patterns that have a small periodicity. The *periods* of a pattern are the time gaps or number of events between each consecutive occurrences of that pattern. The *periodicity* of a pattern is traditionally defined as its maximum period. A pattern is then considered periodic if its periodicity is not greater than a user-defined $maxPer$ threshold [27,45]. For example, a pattern $\{wine, cheese, bread\}$ may be considered to have a periodicity of 4 transactions if is purchased at least every four transactions by a customer (all the pattern's periods are not greater than 4). And if $maxper \geq 4$, this pattern is periodic. This model sometimes called *full* PFP mining has been well studied but has an important drawback. It is that it is too strict as a pattern is discarded if it has only one period exceeding $maxPer$. For instance, consider that $maxPer = 1$ week, and that a customer buys $\{bread, cheese\}$ every week-end. This pattern will be periodic, but if the customer skips a single week, then it will be considered as not periodic.

To provide more flexibility, the model of *partial* PFP mining was defined [30]. It relaxes the $maxPer$ constraint to allow up to a maximum number of periods to exceed $maxPer$. In other words, a pattern is deemed periodic if no more than x of its periods are greater than $maxPer$, where x is a user-defined threshold. Thus, full PFP mining is the special case where $x = 0$. Partial PFP mining is more flexible than full PFP mining but still has an important issue, which is that it only checks if each period exceeds the $maxPer$ threshold but it does not consider by how much a period exceeds that threshold. Thus, if $maxPer$ is set to three days, this model considers that if a customer stops buying some products for a period of one week, it is the same as if he had stopped for a period of one year. Besides, another limitation of this model and previous ones is that they do not check whether the periods satisfying the $maxPer$ constraint are close to each other or not. Thus, a pattern may be considered as periodic even if it frequently alternates between periods that are longer and shorter than $maxPer$. For applications such as the analysis of customer behavior, it is desirable to find stable periodic-frequent patterns, i.e. patterns that have consecutive periods that are more or less stable in terms of length over time. The reason is that a stable pattern will be more predictable than an unstable one as its periods will not vary greatly. Thus, taking decisions based on a stable pattern can be viewed as less risky than based on unstable patterns. Discovering stable patterns can reveal useful information for various applications such as marketing, predicting product demand and managing inventories. For example, knowing that

customers of a store periodically purchase some products about every week and that these patterns are very stable, can help plan orders to refill inventories.

Besides, a second problem with traditional PFP mining algorithms is that the user must set a constraint on the support (occurrence frequency) of patterns. This constraint, inspired by FIM, states that a PFP must have a support that is no less than a *minSup* threshold [45]. Though this constraint is useful to focus on popular trends in the data, setting the *minSup* parameter is not intuitive. If *minSup* is set too high, few PFPs may be found, while if *minSup* is set too low, too many PFPs are found and algorithms may have very long runtimes or even run out of memory or storage space. Because how *minSup* influences the number of PFPs depends on datasets characteristics, a user will typically set this parameter by trial and error until just enough patterns are found, which is time-consuming.

This paper addresses the two above limitations of traditional PFP mining. First, a function to measure the stability is introduced to find periodic patterns that may not always be periodic but generally have a stable periodic behavior. Second, to avoid relying on a *minSup* threshold, it is proposed to find the top-k most frequent patterns that have a stable periodicity. In this novel problem, the user can directly specify the number of patterns k to be found rather than having to set a *minSup* threshold. The parameter k is more intuitive to set than *minSup* because choosing an appropriate *minsup* value depends on datasets characteristics that are initially unknown to users, while k simply represents the number of patterns that the user wants to analyze. The contributions of this paper are threefold:

1. To be able to identify stable periodic patterns, a novel function to measure stability is proposed named *lability*. It is designed to assess how stable the periodic behavior of an itemset is in a sequence of transactions. This function is based on the concept of cumulative sum and evaluates by how much each period exceeds a *maxPer* threshold, and how close these periods are. Properties of the *lability* function are studied and based on that function a formal definition of the concept of stable periodic pattern is proposed. Furthermore, a search space pruning property is designed to discover stable periodic patterns without considering all possible patterns.
2. An algorithm, named TSPIN (Top-k Stable Periodic pattern mINer) is designed to efficiently discover the complete set of top-k stable periodic itemsets in a sequence of transactions. Those are the k most frequent itemsets that have a stable periodicity. The algorithm is correct and complete and its complexity is discussed.
3. Extensive experiments have been done on various benchmark datasets to evaluate TSPIN's performance. Moreover, stable periodic patterns found in real transactions from an online retail store using the designed stability function have been analyzed. Results show that TSPIN is efficient for finding the top-k stable periodic patterns. Moreover, insightful patterns have been found in the real shopping data, which cannot be discovered using traditional PFP mining algorithms.

It is to be noted that this paper extends the paper *Discovering Stable Periodic-Frequent Patterns in Transactional Data*, published in the proceedings of the IEA AIE 2019 conference [17]. In that early version of this work, an algorithm named SPP-Growth was presented for discovering stable frequent periodic patterns using the *minSup* threshold. In the current paper, SPP-Growth is revised as the TSPIN algorithm for mining the k most frequent stable periodic patterns. All sections of the current paper are much more detailed than the conference paper and new experiments have been done.

The rest of the paper is organized as follows. Related work is reviewed in Section 2. The proposed model is presented in Section 3. Then, the TSPIN algorithm is introduced in Section 4. Finally, Section 5 describes experimental results and Section 6 draws a conclusion.

2 Related work

Frequent itemset mining [2] is a popular data mining task that was originally proposed for analyzing customer transaction data to identify sets of frequently purchased items. Frequent itemset mining is applied to a type of data called a *transaction database*. A transaction database, also called a sequence of events, is defined as follows [2,15,23].

Definition 1 (Transaction database) Consider a set of items (e.g. events, symbols or products), denoted as I. An *itemset* is a subset $X \subseteq I$. An itemset containing k items is said to be of size k, or to be a k-itemset. A *transaction database* $D = \{T_1, T_2, ..., T_n\}$ is a multi-set of transactions, where each transaction T_c $(1 \leq c \leq n)$ is an itemset $(T_c \subseteq I)$. Moreover, the number c in subscript of a transaction T_c is said to be its unique Transaction IDentifier (TID). In the following, the notation $tid(T_c)$ will be also used to refer to the transaction identifer of a transaction T_C.

Example 1 A transaction database is shown in Table 1, which will be used as running example. It contains ten transactions denoted as T_1 to T_{10}. The transaction identifier of T_3 is $tid(T_3) = 3$. The transaction T_3 is a 3-itemset $\{a, b, e\}$. In the context of market basket analysis, this transaction may indicate that a customer has purchased some items a, b and e together.

A transaction database can be viewed as a table of records described according to binary attributes. Each transaction is a record and the presence or absence of an item in a transaction indicates if an attribute is true or false. Because this representation is quite general, frequent itemset mining has been applied in many domains [34]. Moreover, if transactions are ordered, this representation can also model a sequence of events such as the sequence of purchases made by a customer, the sequence of treatments received by a hospital patient, or the sequence of learning activities done in an e-learning environment.

The goal of frequent itemset mining is to find all sets of items (values) that frequently appear in a transaction database. FIM is defined based on a frequency measure, called the *support*.

Definition 2 (Support) Let there be a database D and an itemset X. The set of transactions containing X is defined as $trans(X) = \{T | T \in D \wedge X \subseteq T\}$, and the support of X is defined as $sup(X) = |trans(X)|$. In other words, the support of an itemset is the number of transactions that contains it.

Definition 3 (Frequent itemset mining) Let there be a threshold $minSup > 0$, set by the user. Frequent itemset mining is the task of enumerating all frequent itemsets. An itemset X is called infrequent if $sup(X) < minSup$ and frequent if $sup(X) \geq minSup$ [2,34].

Example 2 The itemset $\{a, b, c\}$ appears in the transactions $trans(\{a, b, c\}) = \{T_1, T_2\}$. Thus, its support is $sup(\{a, b, c\}) = |trans(\{a, b, c\})| = 2$. If $minSup = 5$, the result of FIM is the frequent itemsets $\{a\}$, $\{b\}$, $\{c\}$, $\{e\}$ and $\{b, e\}$, which have support values of 5, 8, 6, 6 and 5, respectively.

Table 1 A transaction database containing ten transactions

TID	Itemset	TID	Itemset
1	$\{a, b, c, e\}$	6	$\{b, c, e\}$
2	$\{a, b, c, d\}$	7	$\{b, c, d, e\}$
3	$\{a, b, e\}$	8	$\{a, c\}$
4	$\{c, e\}$	9	$\{a, b, d\}$
5	$\{b, d, e\}$	10	$\{b\}$

The problem of itemset mining is not an easy problem because up to $2^{|I|} - 1$ itemsets may have to be evaluated (excluding the empty set) for a set I of items. For large databases containing many distinct items, this search space can be very large. To avoid considering all possibilities and still find all frequent itemsets, several efficient FIM algorithms have been designed. The first algorithm, named Apriori [2], adopts a breadth-first search. It initially counts the support of single items and then combines pairs of frequent itemsets to generate candidate 2-itemsets. Apriori then scans the database to count the support of these 2-itemsets and then combine frequent 2-itemsets to generate candidate 3-itemsets. Then, this process is repeated to find all larger frequent itemsets. Apriori reduces the search space using a property of the support measure called *Apriori property* or *downward-closure property*, which states that the support of an itemset cannot be greater than the support of its subsets [2]. Thus, if an itemset is infrequent, all its supersets can be ignored. Though Apriori can greatly reduce the search space using that property, a major drawback of Apriori is that it performs many database scans to count the support of itemsets.

To address this issue, other FIM algorithms have been proposed. For example, the Apriori-TID [2] and Eclat [51] algorithms have adopted a vertical database representation where each itemset is annotated with the list of transactions where it appears. This representation can be built using only two database scans, and then allows to calculate the support of any itemset. However, this list-based representation can consume a large amount of memory for databases having many transactions or dense databases. Besides, all the above algorithms have the drawback that they can generate candidate patterns that do not exist in the database, and thus waste a considerable amount of time evaluating them.

To find all frequent itemsets without generating candidate patterns, the FP-Growth [23] algorithm was proposed, which adopts a *pattern-growth* approach. It first scans the database to build a tree based representation. Then, it mines frequent itemsets from the tree and projected trees using a depth-first

search. This approach was shown to be faster than previous approaches. Then, many other FIM algorithms were designed, mainly by extending the Apriori, Eclat and FP-Growth algorithms or some of their concepts, and several variations of the FIM problem have been studied [6,15,34,35,52].

To find patterns that periodically appear in a transaction database where transactions are ordered by time, it was proposed to mine periodic frequent patterns (PFPs) [3,12,30,44]. The traditional model of PFP mining, called (*full*) PFP mining [45], is defined based on the following definitions.

Definition 4 (Transaction database ordered by time) A transaction database $D = \{T_1, T_2, \ldots, T_n\}$ is said to ordered by time if for any two transactions $T_i, T_j \in D$ such that $i < j$, then T_i occurred before T_j.

Definition 5 (List of transactions containing an itemset) Let there be a transaction database $D = \{T_1, T_2, \ldots, T_n\}$ ordered by time and an itemset X. The *list of transactions containing* X is denoted as $g(X)$ and defined as the list of transactions from $trans(X)$, ordered by time.

Example 3 Consider the database of Table 1. For the rest of the paper, it will be assumed that this database is ordered by time. The list of transactions containing the itemset $\{b, c\}$ is $g(X) = \langle T_1, T_2, T_6, T_7 \rangle$.

Definition 6 (i-th transaction containing an itemset X) Let there be a transaction database D ordered by time. The *i-th transaction containing* X is denoted as T_i^X and is the i-th transaction in $g(X)$. Thus, $g(X)$ can be written as $g(X) = \langle T_1^X, T_2^X \ldots T_{|sup(X)|}^X \rangle$.

Example 4 Let $X = \{b, c\}$. The first transaction of Table 1 containing the itemset X is $T_1^X = T_1$. The second transaction containing X is $T_2^X = T_2$. The third transaction containing X is $T_3^X = T_6$. The fourth transaction containing X is $T_4^X = T_7$.

Definition 7 (Extended list of transactions containing an itemset) To simplify some calculations, two transactions are added to the list of transactions $g(X)$ of an itemset X. The result is called the *extended list of transactions* of X, which is defined as $\overline{g}(X) = \langle T_0^X, T_1^X, T_2^X \ldots T_{sup(X)}^X, T_{sup(X)+1}^X \rangle$. where T_0^X and $T_{sup(X)+1}^X$ have the identifiers $tid(T_0^X) = 0$ and $tid(T_{sup(X)+1}^X) = |D|$, respectively. In other word, $T_{sup(X)+1}^X$ is the last transaction of the database, whereas T_0^X is a virtual transaction that would have occurred before the first transaction.

Example 5 The extended list of transactions containing the itemset $\{b, c\}$ is $\overline{g}(X) = \langle T_0, T_1, T_2, T_6, T_7, T_{10} \rangle$.

Definition 8 (Period of an itemset) Let there be a database D ordered by time and an itemset X. Consider any two transactions $T_i^X, T_{i+1}^X \in \overline{g}(X)$. The *period* between T_i^X and T_{i+1}^X is the number of transactions between T_i^X and T_{i+1}^X. It is formally defined as $per(X, i) = tid(T_{i+1}^X) - tid(T_i^X)$, and is said to be a *period* of X.

Definition 9 (List of periods of an itemset) The *list of periods* of an itemset X is defined as $per(X) = \langle per(X, 0), per(X, 1), \ldots per(X, sup(X)) \rangle$.

Example 6 Let $X = \{b, c\}$. The period between T_0^X and T_1^X is $per(X, 0) = tid(T_1^X) - tid(T_0^X) = 1 - 0 = 1$. The period between T_1^X and T_2^X is $per(X, 1) = tid(T_2^X) - tid(T_1^X) = 2 - 1 = 1$. The period between T_2^X and T_3^X is $per(X, 2) = tid(T_3^X) - tid(T_2^X) = 6 - 2 = 4$. The period between T_3^X and T_4^X is $per(X, 3) = tid(T_4^X) - tid(T_3^X) = 7 - 6 = 1$. The period between T_5^X and T_4^X is $per(X, 4) = tid(T_5^X) - tid(T_4^X) = 10 - 7 = 3$. Hence, the list of periods of X is: $per(X) = \langle 1, 1, 4, 1, 3 \rangle$.

Definition 10 (Periodic frequent pattern mining) Let there be two user-defined thresholds $maxPer > 0$ and $maxSup > 0$. The problem of (full) periodic frequent pattern mining consists of identifying all periodic frequent patterns. An itemset X is a *periodic frequent pattern (PFP)* if $maxper(X) \leq maxPer$ and $sup(X) \geq minSup$, where the *periodicity* of X is denoted and defined as $maxper(X) = max(per(X))$ [45].

Example 7 Assume that $maxPer = 2$ and $minSup = 5$. The PFPs found in the database of the running examples are $\{b\}$ and $\{c\}$, where $sup(\{b\}) = 8$, $maxper(\{b\}) = 2$, $sup(\{c\}) = 6$, and $maxper(\{c\}) = 2$.

It is important to note that the problem of PFP mining can be easily adapted to handle a database where each transaction has a unique timestamp. This can be done by simply replacing transaction identifiers by timestamps in the database. The result will be that all periods will be calculated in terms of time duration instead of number of transactions, which can be more meaningful for some applications.

Example 8 Assume that transactions T_1, T_2 $\ldots T_{10}$ were recorded at time 10, 20 $\ldots$ 100. Then, the list of periods of $\{b, c\}$ is $per(X) = \langle 10, 10, 40, 10, 30 \rangle$, assuming that $T_0^X = 0$.

The problem of mining PFPs was introduced by Tanbeer et al. [45]. They designed the PF-growth algorithm to enumerate all PFPs by extending the FP-Growth algorithm. Thereafter, an algorithm inspired by Eclat was proposed, named MTKPP [3]. It relies on a depth-first search and a vertical database representation to enumerate all PFPs. However, a problem with PF-tree, MKTPP and other traditional PFP mining algorithms is that they discard a pattern if only one of its periods is larger than $maxPer$. For example, if $maxPer = 2$, the itemset $\{e\}$ is not considered as periodic because it has one period of 3 exceeding $maxPer$, although it is otherwise always periodic.

To provide more flexibility, it was proposed to set a $minSup$ threshold and a $maxPer$ threshold for each item [44] so that each item can be assessed differently. Although this can be useful for some applications, a problem is that the user needs to set $2 \times |I|$ parameters in an appropriate way, which can be difficult and time-consuming in practice.

To relax the maximum periodicity constraint, an alternative solution designed by Kiran et al. [30] was to use a function called *periodic-frequency*. The periodic frequency of an itemset X is defined as $perFreq(X) = \frac{|\{i|per(X,i) \leq maxPer \wedge i \in [0, sup(X)]\}|}{|per(X)|}$, that is the percentage of periods where X is periodic. Then, an algorithm was designed to mine all (*partial*) periodic patterns, that is those having a periodic frequency that is no less than a user-defined threshold. Though this definition is more flexible than full PFP mining, a major problem is that patterns having some very long periods can still be considered as periodic. For instance, the item $\{a\}$ could be considered as partial-periodic for $maxPer = 2$ even if it is periodic only in the first few transactions. This problem occurs because the amount by which $maxPer$ is exceeded in a period is not taken into account.

In another study, Fournier-Viger et al. [11] proposed to measure the average periodicity of a pattern as $avg(X) = avg(per(X))$ and find patterns having an average periodicity that is between a minimum and maximum value. They proposed an algorithm inspired by Eclat to find these patterns, named PFPM [11], and an extension for high utility itemset mining, named PHM [12]. Though using the average allows to consider by how much a period is exceeded, this algorithm does not consider whether periods exceeding the threshold are close to each other or not. In other words, each period is evaluated independently of the periods that occurred before and after. As a result, a pattern could have dozens of very long consecutive periods where it is not periodic followed by dozens of periods where it is periodic, and still be considered as periodic for the whole sequence in terms of average periodicity. But such patterns clearly do not have a stable periodicity. A similar approach has been to use the variance [32, 42, 43] and the standard deviation [39, 40, 1] to evaluate the periodicity of patterns. But these studies have the same limitation of not considering whether periods are close to each other or not.

Some approximate algorithms were also proposed for periodic pattern mining. For instance, the ITL-tree algorithm [4] searches for PFPs while using an approximate calculation of the periodicities of patterns. Another approximate algorithm for PFP mining was proposed by Kiran and Reddy [28]. A drawback of these algorithms is that they cannot guarantee a complete set of results. This paper focus on designing an exact algorithm.

Some other variations of the above models were also designed to address specific needs. For instance, Kiran et al. designed an Eclat-based algorithm to mine partial periodic patterns in spatial data [29], Afriyie proposed a method to find a summary of periodic frequent patterns [1], Islam et al. designed a model for mining periodic patterns in RF-tag data [26], and Huang et al. integrated the PFPM algorithm in a system for health management [25].

To summarize, many studies on PFP mining evaluate the periodic behavior of a pattern by only counting the number of periods that are greater than $maxPer$ and ignore by how much these periods exceed $maxPer$ [3, 45]. Though a few studies have proposed solutions such as using the average periodicity [11, 12], variance [32, 42, 43] and standard deviation [39, 40, 1], they process each period independently, that is they do not consider whether periods are close to each other or not. Moreover, another limitation is that most studies require to set a $minSup$ threshold that is hard to set [40, 42, 43, 45]. If it is set too low, few patterns are found. And if it is set too high, too many patterns may be found and algorithms may have long runtimes. To find patterns that have a stable periodic behavior over time, this study propose a novel problem of mining the top-k stable periodic-frequent patterns by considering not only by how much the periods of each pattern exceed $maxPer$ but also whether these periods are close in time. Moreover, rather than using a $minSup$ threshold, the user can directly set k, the number of patterns to be found, and the proposed algorithm returns the top-k most frequent patterns that have a stable periodic behavior.

3 The Proposed Model

This section presents the proposed model of top-k stable periodic-frequent pattern mining. It can be viewed as an extension of periodic-frequent pattern mining that captures frequent patterns having a stable periodic behavior.

Identifying stable periodic patterns requires to measure how the periodic behavior of a pattern is changing over time and to check if the patterns always remain more or less periodic. For assessing how a pattern's periodic behavior varies, this paper introduces a novel model based on the concept of cumulative sum. It is a technique commonly used to find changes in time series. The cumulative sum for the first data point of a time series is the difference between that point and a fixed number ρ. Then, the cumulative sum for the i-th point of a time series is the sum of the differences of each of the first i data points and ρ, or zero if the total is negative. Then, if the cumulative sum at the i-th point exceeds a fixed number α that is greater than ρ, then a change is said to have happened in the time series at that point [20,38]. To explain this more formally, consider a sequence of w numbers $w_1, w_2, \ldots, w_r$. The cumulative sum for the i-th data point ($0 < i \leq w$) is defined as $C_i = \max(0, C_{i-1} + w_i - \rho)$ where $C_0 = 0$. Although the cumulative sum is useful to detect a change in a time series by looking at how values change over time, it is not designed for assessing the periodicity of patterns using their periods and for identifying stable periodic patterns.

As a solution, this paper proposes to assess the stability of a pattern by calculating the cumulative sum of the difference between each of its periods and $maxPer$. This allows to accumulate the amount by which $maxPer$ is exceeded for consecutive periods. Then, a change detected using this sum is interpreted as a display of instability. The novel function for assessing the stable periodic behavior of patterns based on the cumulative sum is called *lability*, and is defined as follows.

Definition 11 (Lability of an itemset) The lability of an itemset X is a list of values denoted as $la(X) = \langle la(X,0), la(X,1), ..., la(X, sup(X))\rangle$ that contains $sup(X)+1$ values. In other words, $|la(X)| = sup(X)+1 = |per(X)|$. Each lability value in $la(X)$ is no less than zero. The first lability value of X is defined as $la(X,0) = max(0, per(X,0) - maxPer)$. It calculates the difference between the first period of X and $maxPer$. Then, the i-th lability value of X for $i > 0$ is defined based on the the previous lability value as $la(X,i) = max(0, la(X,i-1) + per(X,i) - maxPer)$. Thus, lability values are calculated as a cumulative sum. Note that the above definition of lability can also be rewritten more concisely as follows: $la(X,i) = max(0, la(X,i-1) + tid(T^X_{i+1}) - tid(T^X_i) - maxPer)$ where $la(X,-1)$ is defined as $la(X,-1) = 0$.

The main idea behind that definition is the following. For an itemset X, a list of lability values is calculated. The i-th lability value of X corresponds to the i-th period of X. The first lability value is the difference between the first period of X and $maxPer$. This is to evaluate by how much the first period exceeded the $maxPer$ threshold. If that value is negative, then it is set to zero. Then, the following lability values are a cumulative sum of the lability values to accumulate the differences between each period and the $maxPer$ threshold. This definition is interesting as it allows to accumulate the amounts by which $maxPer$ is exceeded over time. If an itemset exceeds $maxPer$ for several periods that are close to each others, the exceeding amounts will be accumulated in the lability values. Conversely, if the periods of a pattern are less than $maxPer$ for many periods that are close to each other, the accumulated lability values will decrease, until it reaches a minimum value of zero. A lability value close to zero means that a pattern has a stable periodic behavior while a large lability value means that a pattern has an unstable periodic behavior (its periods often exceeds $maxPer$ or exceed $maxPer$ by large values).

Example 9 For the database of the running example and $maxPer = 2$, the periods of itemset $\{d\}$ are $per(\{d\}) = \{2,3,2,2,1\}$. Since the itemset $\{d\}$ has five periods, it also has five lability values. The first lability value of $\{d\}$ is $la(\{d\},0) = max(0, per(\{d\},0) - maxPer) = max(0, 2-2) = 0$. Then, the following lability values are $la(\{d\},1) = 1$, $la(\{d\},2) = 1$, $la(\{d\},3) = 1$ and $la(\{d\},4) = 0$. Thus, the lability of itemset $\{d\}$ is $la(\{d\}) = \{0,1,1,1,0\}$.

The proposed lability function has the interesting property that if a pattern always has periods smaller than $maxPer$, its lability will be zero, and if a pattern has many periods larger/smaller than $maxPer$, the amounts above/below $maxPer$ will be accumulated by increasing/decreasing the lability values. Hence, the *lability* of an itemset is changing over time depending on its recent periodic behavior. Because low lability values indicate a stable behavior and high values indicate an unstable one, the lability function can be used to find stable patterns by setting a maximum constraint on the lability. Based on this idea, the concept of stable periodic frequent pattern is defined as follows.

Definition 12 (Stability) The *maximum lability* of an itemset X is defined as $maxla(X) = max(la(X))$, and is also called the *stability* of X.

Example 10 As the lability values of itemset $\{d\}$ are $la(\{d\}) = \{0, 1, 1, 1, 0\}$, then $maxla(\{d\}) = 1$.

Definition 13 (Stable periodic pattern) Let there be an itemset X and a threshold $maxLa \geq 0$, called maximum lability threshold. An itemset X having a lability no greater than $maxLa$ is said to be a stable periodic pattern (SPP), i.e. $maxla(X) \leq maxLa$. Moreover, to avoid finding infrequent patterns, a support constraint can be added. An SPP that has a support that is no less than a user-defined threshold $minSup > 0$ is said to be a stable periodic-frequent pattern, i.e. $sup(X) \geq minSup$.

Based on the above definition, the problem of stable periodic-frequent pattern mining is defined.

Definition 14 (Stable periodic-frequent pattern mining) Let there be a transaction database D, a set of items I, three user-defined thresholds $minSup > 0$, $maxPer > 0$ and $maxLa \geq 0$. The problem of mining the stable periodic-frequent patterns in D consists of enumerating each itemset X in D such that $maxla(X) \leq maxLa$ and $sup(X) \geq minSup$.

Example 11 If $maxLa = 1$ and $minSup = 4$, the periodic-frequent patterns in the database of the running example are $\{b\}$, $\{c\}$, $\{e\}$ and $\{b, e\}$. The support and stability of $\{b\}$ are $sup(\{b\}) = 8$ and $maxla(\{b\}) = 0$, respectively. The itemset $\{c\}$ has a support of 6 and a stability of 0. The itemset $\{e\}$ has a support of 6 and a stability of 1. Finally, the itemset $\{b, e\}$ has a support of 5 and a stability of 1.

It can be observed that the above problem of stable periodic-frequent pattern mining generalizes the problem of full PFP mining. Indeed, if $maxLa = 0$, the former becomes equivalent to the latter. The problem of SPP mining is more flexible than that of traditional PFP mining, since SPP mining allows periods to exceed the $maxPer$ threshold as long as the accumulated sum (lability) remains below $maxLa$. But a limitation of the problem of SPP mining is that the user still need to set a $minSup$ threshold, and finding an appropriate value for this parameter is not obvious in practice as it is dataset dependent.

To address this issue, a variation of the SPP mining problem is proposed called *top-k stable periodic-frequent pattern mining*, where $minSup$ is replaced by a parameter k. This let the user directly specify the number k of patterns to be discovered.

Definition 15 (Top-k stable periodic-frequent pattern mining) Let there be a set of items I, a transaction database D and three user-defined parameters $minSup > 0$, $maxPer > 0$ and $k \geq 1$. The problem of top-k stable periodic-frequent pattern mining consists of discovering the set of the k most frequent itemsets that are stable periodic patterns, that is a set Z of k itemsets such that $\forall X \in Z, maxla(X) \leq maxLa \wedge \nexists Y \in Z | sup(Y) > sup(X)$.

Example 12 If $maxPer = 5$, $maxLa = 1$ and $k = 3$, the top-3 stable periodic-frequent patterns are $\{b\}$, $\{c\}$ and $\{e\}$ because no other stable periodic patterns have a higher support. The support and lability values of these patterns are (8,0), (6, 0) and (6,1), respectively.

It can be observed that for some databases and values of k, it is possible that more than k patterns could be included in the set Z. For instance, this can happen if more than k patterns have exactly the same support. In that case, the above problem definition only requires to find k of those patterns. Besides, it is also possible that the set Z contains less than k patterns. For instance, this can happen if the number of possible patterns in a database is less than k.

In terms of applications, the proposed problem of top-k SPP mining can be applied in many domains where data is modeled as a sequence of events or symbols, with or without timestamps. Because the proposed problem generalizes the problem of full PFP mining, it can be used for the same applications such as to perform activity monitoring [26], to analyse GPS trajectories [53], for health management [25], to perform user location prediction [48], to improve the performance of recommender systems [9], for market basket analysis [11], and to analyze pollution data [29]. For these applications, the benefits of finding stable periodic patterns instead of traditional PFPs is to allow more flexibility when searching for periodic patterns by allowing $maxPer$ to be temporarily exceeded, and to ensure that patterns are stable by having a periodic behavior that remains more or less below $maxPer$ in consecutive periods). A use case of SPP mining in market basket analysis is to analyze the purchases of a customer over time to find his most steady habits such as to buy a newspaper, bread and coffee every morning. Finding steady habits (stable patterns) is more interesting for marketers than studying irregular habits because

the former are more predictable. Stable patterns can be used to offer personalized marketing such as offering discounts on newspaper with bread and coffee. A second interesting use case is to follow the activities of an elderly person to find stable patterns representing the typical activities that the person performs everyday. These patterns can then be used to define a model of the normal behavior of that person. Then, if the person's behavior eventually deviates from that model (e.g. because of a fall), an alarm could be raised.

Creating an algorithm to efficiently enumerate all SPPs or top-k SPPs in a transaction database requires to design efficient search space pruning strategies. In FIM, there exists the well-known *Apriori* property for reducing the search space using the support measure [2]. To be able to also reduce the search space using the lability function, the following lemma and theorem are presented.

Lemma 1 *(Monotonicity of the maximum lability) For any two itemsets $X \subset Y \subseteq I$, the relationship $maxla(Y) \geq maxla(X)$ holds.*

Proof Because $X \subset Y$, it follows that $\overline{g}(Y) \subseteq \overline{g}(X)$. In the case where $\overline{g}(Y) = \overline{g}(X)$, the periods of X and Y are the same. Hence $la(Y) = la(X)$ and $maxla(Y) = maxla(X)$. In the case where $\overline{g}(Y) \subset \overline{g}(X)$, then for each transaction $\{T_z | T_z \in \overline{g}(X) \wedge T_z \notin \overline{g}(Y)\}$, the corresponding period $per(X, z)$ will be replaced by a larger period $per(Y, z)$. Hence, any period in $per(Y)$ cannot be smaller than a period in $per(X)$. Thus, $maxla(Y) \geq maxla(X)$, and the lemma holds.

Theorem 1 *(Maximum lability pruning) For a database D, if $maxla(X) > maxLa$ for an itemset X, then X and its supersets are not SPPs. Hence, the part of the search space containing X and its supersets can be ignored.*

Proof According to the definition of SPP, if $maxla(X) > maxLa$, then X is not a SPP. Then, any superset Y of X is also not a superset based on Lemma 1.

Besides, it can be observed that the problem of top-k stable periodic-frequent pattern mining is more difficult than that of SPP mining since the minimum support value to obtain the k most stable periodic-frequent patterns is not known in advance. Consequently, all itemsets having a support greater than zero may have to be evaluated to select the top-k SPPs. Because of this, the search space of top-k stable periodic-frequent pattern mining is always greater or equal to that of SPP mining when the minimum support threshold is set to the optimal value to obtain k patterns. To efficiently discover top-k stable periodic-frequent patterns, the next section describes the proposed TSPIN algorithm.

4 The TSPIN Algorithm

This section presents the proposed TSPIN algorithm, which performs two main steps. First, it scans the input database to create a stable periodic-frequent tree (SPP-tree) structure. Then, TSPIN mines the top-k stable periodic-frequent patterns directly from that tree using a depth-first search.

To avoid generating candidate itemsets that do not exist in the database, TSPIN adopts a pattern-growth approach similar to that of FP-Growth [23] for FIM. But there are several differences between FP-Growth and TSPIN. First, TSPIN stores the IDs of transactions (or timestamps) in its tree structure to calculate periods and the lability function. Second, TSPIN utilizes the lability function to identify SPPs and for search space pruning. Third, TSPIN is also adapted to find the top-k patterns rather than using a fixed *minSup* threshold.

This section first presents the SPP-tree structure and how it is built. Then, the process for mining the SPP-tree to discover the top-k SPPs is described, and a brief example of how the algorithm is applied is presented.

4.1 The SPP-tree structure

The main data structure used by TSPIN is a structure called SPP-tree, which contains two sub-structures:

- The *prefix-tree* is a tree structure that stores transactions from the input database. Each tree node represents an item and each tree path represents a transaction or a part of a transaction. Initially, the algorithm scans the database to build this tree, and then the database is not required anymore as all the relevant information for mining SPPs is stored in that tree.

– The *SPP-list* is a structure that is used for quickly finding items in the prefix-tree. The SPP-list contains a tuple of the form (i, S, ML, pt) for each item i appearing in the prefix-tree. In that tuple, S is the support $sup(i)$ of i and ML is the maximum lability $maxla(i)$ of item i for transactions in the prefix-tree. Moreover, pt is a pointer to a prefix-tree node that contains item i.

The prefix-tree is similar to the FP-tree of FP-Growth [23] since they both stores transactions or parts of transactions as tree paths. However, a difference is that while each FP-tree node stores an item i and a support value, an SPP-tree node does not store the support but can store additional information. More precisely, there are two types of SPP-tree nodes:

– An *ordinary node* stores an item i, similarly to an FP-tree node.
– A *tail node* is a node that contains the last item i of a path representing a transaction. A tail node stores the item i but also a list called *TID-list* indicating the transactions (or timestamps) ending at that node. Keeping this extra information is useful for calculating the maximum lability and support of itemsets, as it will be explained. Formally, the content of a tail node is denoted as $i[t_a, t_b, \ldots, t_c]$, where i is the node's item and t_j $(j \in [1, n])$ is the IDs of transactions where i is the last item.

After building the tree, the designed TSPIN algorithm traverses the tree to mine SPPs. For this purpose, each prefix-tree node has pointers to its parent and childs. Moreover, as previously explained, each tuple (i, S, ML, pt) of an SPP-list contains a pointer to a prefix-tree node containing the item i in the prefix-tree. This node has a pointer called *nodelink* which points to the next node having the item i in the prefix-tree, and this latter points to another one, and so on, such that all nodes having item i are linked by these pointers. Thus, these *nodelink* pointers allows to quickly traverse all nodes having a given item i in the prefix-tree.

The process for building the SPP-Tree structure is detailed in the next sub-section. It consist of inserting all transactions from the input database as a branch in the tree. But before this process starts, items in transactions are sorted according to a total order $\prec$ on I (e.g. the lexicographical order). By ensuring that items in transactions are sorted, all transactions that share a same prefix will overlap in the prefix-tree (have some common tree nodes), which will reduce the space required by the tree. To have a high likelihood that transactions overlap, the $\prec$ order used in the implementation of the proposed algorithm is the descending order of item support as used in FP-Growth [23].

4.2 Building an SPP-tree

Two main tasks must be carried out to build an SPP-tree: building the SPP-list and constructing the prefix-tree.

Constructing an SPP-list. The process for building an SPP-list is described in Algorithm 1. The input is a transaction database D and the user-specified $maxPer$ and $maxLa$ parameters. The output is an SPP-list, which contains a tuple (i, S, ML) for each item i found in the database that is an SPP, and where $S = sup(i)$ and $ML = maxla(i)$.

The algorithm first initializes a temporary array t (line 1). This array will store for each item i, the TID of the last transaction containing i, denoted as $t_{last}(i)$, the maximum lability $ML(i)$ of i, the lability $la(i)$ of i and the support $S(i)$ of i. The values are initialized as $ML(i) = 0$, $la(i) = 0$ and $S(i) = 0$. Then the algorithm reads each transaction T having a TID (or timestamp) t_{cur} (line 2). For each item i in the transaction T, the algorithm increases $S(i)$ by 1, updates the lability $la(i)$ using the current period length $t_{cur} - t_{last}(i)$, updates the maximum lability $ML(i)$, and updates $t_{last}(i)$ (line 3 to 7). After the last transaction has been read, t_{cur} is set to the database size $|D|$ and the information of each item i is updated in t (line 10). Then, the SPP-list is created, containing a tuple (i, S, ML) for each item i such that $ML(i) < maxLa$ according to t, and where items are sorted in descending order of support (line 11). Note that at this stage, the pt field of each tuple of the SPP-list is not created yet.

An example is provided to illustrate the process of SPP-list construction. Consider the database of Table 2, $maxPer = 2$ and $maxLa = 1$. Figures 1(a) illustrates the content of the t array after reading the first transaction. Figure 1(b) shows the t array after reading the second transaction. Figure 1(c) depicts the t array after inserting all transactions. Figures 1 (d) shows the t array after adding the TID $g(sup(X) + 1)$ to every item (line 10 of Algorithm 1). Figures 1 (e) shows the final SPP-list containing the stable itemsets, sorted by descending order of *support* (line 11 of Algorithm 1).

Constructing a prefix-tree. After building an SPP-list, TSPIN scans the database again to build the prefix-tree. This is done by Algorithm 2, which takes as input a transaction database and the SPP-list. The output is a prefix-tree and an updated SPP-list. The prefix-tree is constructed in a way that is

i	S	ML	$t_{last(i)}$
b	1	0	1
c	1	0	1
e	1	0	1

(a)

i	S	ML	$t_{last(i)}$
b	2	0	2
c	2	0	2
e	1	0	1

(b)

i	S	ML	$t_{last(i)}$
b	8	0	10
c	6	0	8
e	6	0	7

(c)

i	S	ML
b	8	0
c	6	0
e	6	1

(d)

i	S	ML
b	8	0
c	6	0
e	6	1

(e)

Fig. 1 The t array after scanning (a) the first transaction of Table 2, (b) the second transaction, (c) the entire database, (d) after adding $t = gX(sup(X) + 1)$ to each item, and (e) the final SPP-list containing the sorted list of items.

Algorithm 1 Construction of an SPP-list

Input:
D: a transaction database,
$maxPer$, $maxLa$: the user-specified parameters
Output: the SPP-list containing the sorted list of items

1: Create a temporary array t to store for each item i, the TID $t_{last}(i)$ of the last transaction containing i, the maximum lability $ML(i)$ of i, the lability $la(i)$ of i and the support $S(i)$ of i. The values are initialized as $ML(i) \leftarrow 0$, $la(i) \leftarrow 0$ and $S(i) \leftarrow 0$.
2: **for each** transaction $T \in D$ with TID t_{cur} **do**
3: **for each item** $i \in T$ **do**
4: $S(i) \leftarrow S(i) + 1$;
5: $la(i) \leftarrow max(0, la(i) + t_{cur} - t_{last}(i) - maxPer)$;
6: $ML(i) \leftarrow max(ML(i), la(i))$;
7: $t_{last}(i) \leftarrow t_{cur}$;
8: **end for**
9: **end for**
10: Set $t_{cur} = |D|$ and update the information of each item in t.
11: Create a SPP-list containing a tuple (i, S, ML) for each item i such that $ML(i) < maxLa$ according to t, and where items are sorted in descending order of support.
12: Return the SPP-list.

Table 2 A transaction database with three items b, c, e

TID	Itemset	TID	Itemset
1	$\{b, c, e\}$	6	$\{b, c, e\}$
2	$\{b, c\}$	7	$\{b, c, e\}$
3	$\{b, e\}$	8	$\{c\}$
4	$\{c, e\}$	9	$\{b\}$
5	$\{b, e\}$	10	$\{b\}$

similar to how FP-Growth constructs an FP-tree [23]. However, there are some differences because an SPP-tree stores a list of TIDs (or timestamps) in each tail node, and support values are not stored in nodes.

The algorithm first creates the tree root T with $i = null$ as item. Then, each transaction T is read, sorted using the descending order of item support (as in the SPP-list), and inserted in the prefix-tree. The insertion is done by calling the *insert_tree* procedure. It creates a path in the prefix-tree representing the transaction. During the tree creation, the *pt* pointers in the SPP-list, as well as the parent, child and *nodelink* pointers of each node are updated.

An example of prefix-tree construction for the transaction database of Table 2 is described next. Figure 2(a)-(e) illustrates the SPP-tree that is built after scanning the first, second, eighth, and all database transactions. An SPP-list contains a pointer *pt* from each item i in the SPP-list to an item i in the tree. Moreover, all nodes representing an item i in the tree are linked by the *nodelink* pointers. This allows to quickly traverse all occurrences of each item i in the tree when mining patterns from the tree. For the sake of simplicity, these pointers are not shown in the illustrations of Figure 2. But they are created in the same way as for an FP-tree.

Algorithm 2 Construction of an SPP-tree

Input:
D: a transaction database,
SPP-list: contains stable periodic-frequent items, their support S and maximum lability ML

1: Create the root of the SPP-tree, R, and label it with "null";
2: **for each** transaction $T \in D$ with TID t_{cur} **do**
3: Sort stable periodic-frequent items in T according to the descending order of support.
4: Let the sorted candidate item list be $[p|P]$, where p is the first item and P is the remaining items.
5: Call $insert\_tree([p|P], t_{cur}, R)$, which is performed as follows. If R has a child N such that $N.item\text{-}name \neq p.item\text{-}name$, then create a new node N. Link its parent to R. Let its nodelink be linked to nodes with the same *item-name* via the nodelink structure. Remove p from $[p|P]$. If P is empty, add t_{cur} to the leaf node; else, call $insert\_tree(P, ts_{cur}, N)$ recursively.
6: **end for**
7: Return the SPP-tree;

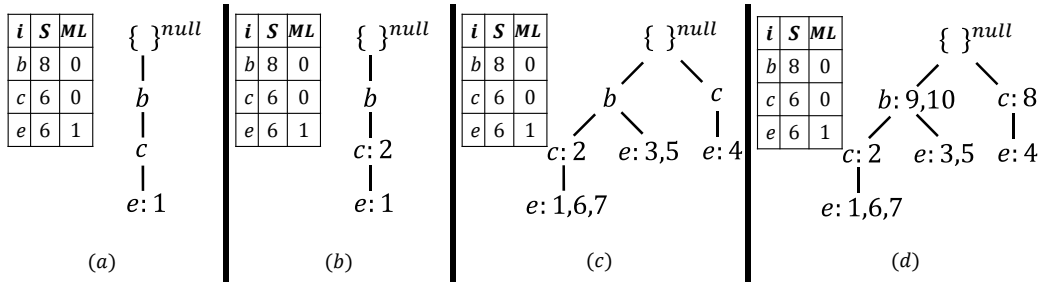

Fig. 2 The SPP-tree built after scanning the (a) first transaction, (b) second transaction, (c) eighth transaction, and (d) all transactions.

4.3 Mining an SPP-tree

After the proposed algorithm has built the SPP-tree, it does not need the original database anymore to find SPPs because all the important information for mining SPPs is stored in the SPP-tree.

The basic idea of TSPIN for finding the top-k patterns is the following. TSPIN first sets an internal *minSup* variable to 1. Then, TSPIN starts searching for stable periodic-frequent patterns by applying a recursive depth-first search procedure. As soon as a pattern is found, it is added to a priority queue of itemsets Q_k ordered by the support. This queue is used to maintain the top-k patterns found until now. Once k SPPs have been found, the internal *minSup* variable is raised to the support of the least frequent itemset in Q_k. Raising the *minSup* value is done to prune the search space when searching for more SPPs. Then, each subsequent time that an SPP is inserted into Q_k, the least frequent itemset is removed from Q_k, and *minSup* is raised to the support of the least frequent itemset among those remaining in Q_k. Then, TSPIN continues searching for more patterns until no pattern can be generated. Then, it has found the top-k stable periodic patterns.

To explore the search space of itemsets, TSPIN performs a depth-first search, which is conducted by Algorithm 3. This algorithm takes as input an itemset α to be extended to find SPPs (initialy the empty set), the initial SPP-tree P_α and the corresponding SPP-list $PList_\alpha$, the priority queue Q_k (initially empty), a *minSup* threshold (initially set to 1), and the user-defined *maxPer* and *maxLa* thresholds. The TSPIN procedure attempts to find extensions of α of the form $\beta = \alpha \cup \{i\}$ that are top-k SPPs. For each such extension, the procedure then recursively calls itself to find other top-k SPPs having β as prefix. This recursive process of extending patterns starting from single items ensures that all top-k SPPs can be found.

The algorithm performs a loop on items stored in the SPP-list $PList_\alpha$ in reverse order. For each item i such that $ML(i) \leq maxLa$ and $S(i) \geq minSup$, the algorithm saves the itemset $\beta = \alpha \cup \{i\}$ as a top-k SPP in Q_k with its support and maximum lability. If Q_k contains at least k itemsets, the *minSup* threshold is raised to that of the least frequent itemset in Q_k. And if Q_k contains more than k itemsets, the least frequent itemset in Q_k is removed from Q_k. Then, the algorithm tries to find all items that could extend β to generate larger itemsets. The set of these items, called γ, contains the ancestors of β in the SPP-tree P_α. The algorithm traverses the nodelinks of i to collect the TID (or timestamps) of all items in γ, and creates a conditional pattern base of β. The conditional pattern base of β is the set of paths in the current SPP-tree P_α that leads to i (excluding the nodes representing i). Then, an SPP-list is created, denoted as $PList_\beta$, containing an entry $PList_\beta(j)$ for each item j in γ. This entry

will be used to calculate the maximum lability and support of $\beta \cup \{j\}$ to determine if it is an SPP. The set of all items that allows to build a SPP is called $\gamma^{'}$. If this set is not empty, the TSPIN algorithm recursively calls itself to store all SPPs of the form $\beta \cup \{j\}$ where $j \in \gamma^{'}$ and to look for other SPPs that extend these patterns. The recursive call is done using a new SPP-tree P_β, which is created by inserting all paths of the conditional pattern base of β as transactions, and using $\gamma^{'}$ as items for its SPP-list, denoted as $PList_\beta$. Then, the item i is removed from the SPP-tree P_α and its TIDs (or timestamps) are pushed to its parent node. The item i can be removed because it is not needed when exploring the rest of the search space according to the processing order of items. Since the algorithm starts from single item SPPs and recursively explores the search space by appending items, all itemsets can be visited. And since the algorithm only prunes itemsets using Theorem 1, it can be seen that this procedure is correct and complete to discover the top-k SPPs.

Note that to accelerate the search for finding the top-k SPPs, the user can decide to start from an internal $minSup$ value greater than 1. However, if this is done, there is a risk of missing some top-k SPPs. Thus, the implementation of TSPIN uses $minSup = 1$ as default value to guarantee completeness.

Algorithm 3 The TSPIN algorithm

Input:
α: an itemset (initially $\emptyset$),
P_α: an SPP-tree,
$PList_\alpha$; the corresponding SPP-list,
Q_k: a priority queue for storing the current top-k stable periodic patterns (initially empty), where patterns with smaller support have higher priority
$minSup$: an internal minimum support threshold (initially set to 1)
$maxPer$: the user-specified maximum periodicity threshold,
$maxLa$: the user-specified maximum lability threshold,
k: the user-specified number of patterns to be found
Output: a set of top-k stable periodic patterns

1: **for** each item i in $PList_\alpha$ (in reverse order) such that $ML(i) \leq maxLa$ and $S(i) \geq minSup$ **do**
2: $\beta \leftarrow \alpha \cup \{i\}$;
3: Insert β into Q_k;
4: **if** $Q_k.size \geq k$ **then**
5: $minSup \leftarrow sup(Q_k.peek())$;
6: **end if**
7: **if** $Q_k.size > k$ **then**
8: pop the highest priority (least frequent) itemset from Q_k;
9: **end if**
10: Let γ be the ancestor items of β in P_α;
11: Traverse the nodelink of i to construct β's conditional pattern base and collect the TIDs (or timestamps) of each item in γ;
12: Create an *SPP-list*, denoted as $PList_\beta$, containing an entry $PList_\beta(j)$ for each item j in γ;
13: $\gamma^{'} \leftarrow \{j | j \in \gamma \wedge Plist_\alpha(j).size > 0\}$;
14: **if** $\gamma^{'} \neq \emptyset$ **then**
15: Construct β's conditional tree P_β while updating the pt fields of tuples in $PList_\beta$;
16: Call $TSPIN(\beta, P_\beta, PList_\beta, Q_k, minSup, maxPer, maxLa, k)$;
17: **end if**
18: Remove i from P_α and push i's TIDs (or timestamps) to its parent nodes;
19: **end for**
20: Return Q_k;

To provide more details about the process of mining SPPs using the SPP-tree structure, a brief example is given. Consider the transaction database of Table 2 and that $k = 3$, $maxPer = 2$ and $maxLa = 1$, respectively. The prefix-tree P_α for $\alpha = \emptyset$ is initially built by scanning the database. That SPP-tree is shown in Figure 2 (c), where the SPP-list contains the items b, c, and e, sorted in that order. The TSPIN procedure is then called with this tree P_α, the corresponding SPP-List $PList_\alpha$, $\alpha = \emptyset$, the priority queue Q_k initialized as empty, and the $maxLa$, $maxPer$ and k parameters. To make this example shorter, it will be assumed that the internal $minsup$ threshold is initially set to 5 rather than 1. The TSPIN procedure processes each item $i \in PList_\alpha$ in reverse order. For each such item, an itemset of the form $\beta \cup \{i\}$ is created.

First, consider $\beta = \{e\}$. Since $S(e) = 6 \geq minSup$ and $ML(e) = 1 \leq maxLa$ according to $PList_\alpha$, $\{e\}$ is a SPP and it is inserted in Q_k as a current top-k SPP. Then, the next step is to evaluate whether extensions of β may be also top-k SPPs. For this purpose, the nodelinks of e are followed to quickly find all occurrences of e in the SPP-tree. The TIDs (or timestamps) of all ancestors of e are collected.

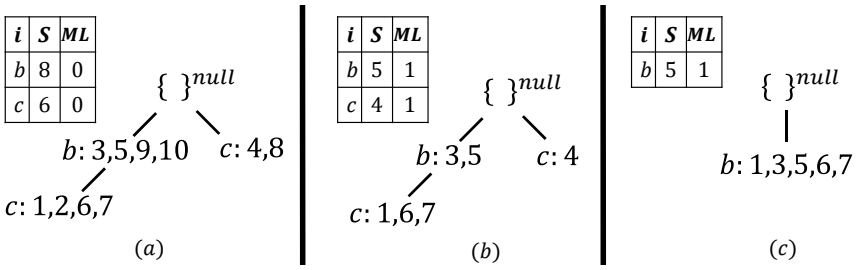

Fig. 3 Mining stable periodic-frequent patterns using suffix item e. (a) The SPP-tree after removing the item e, (b) Prefix-tree of suffix item e, (c) Conditional tree of suffix item e.

Those ancestors are $\gamma = \{b, c\}$. The conditional pattern base of $\{e\}$ is created, which is shown in the second line / third column of Table 3, and illustrated as a tree in Figure 3 b). Then, the procedure calculates the SPP-list of β, denoted as $PList_\beta$, and in particular tuples for $\beta \cup \{b\}$ and $\beta \cup \{c\}$. These tuples are shown in the second line / fourth column of Table 3. Because the support of $\beta \cup \{c\}$ is greater than $minsup = 5$, that extension of β is not a SPP and does not need to be considered or extended. On the other hand, $\beta \cup \{b\}$ has a support of 5 and a *maximum lability* of 1, and is thus a top-k SPP and its extensions must be considered. Thus, $\gamma^{'} = \{b\}$. The SPP-tree of *beta* is built by applying the SPP-tree construction procedure using the paths from the conditional pattern base of β. The result is the SPP-tree $P_{\{b\}}$ shown in Figure 3 c) where the SPP-list contains the item $\gamma^{'} = b$. This tree is then recursively mined by calling the TSPIN procedure. This latter inserts the itemset $\{b, e\}$ with its support of 5 and maximum lability of 1 in Q_k, and explores 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 SPP-tree and its TIDs (or timestamps) are pushed to the parent nodes. The result is shown in Figure 3 a).

Then, the algorithm considers $\beta = \{c\}$. Since $S(c) = 6 \geq minSup$ and $ML(c) = 0 \leq maxLa$ according to $PList_\alpha$, $\{c\}$ is a SPP and it is inserted in Q_k as a current top-k SPP. Since Q_k contains k itemsets, the $minSup$ threshold is set to that of the least frequent itemset in Q_k, which is 5. The conditional pattern base of $\{c\}$ is then calculated, which is depicted in the third line / third column of Table 3. Here, $\gamma = \{b\}$. It is found that the support of $\beta \cup \{b\}$ is less than 5 and the maximum lability of $\beta \cup \{b\}$ is more than 1, so it is not a SPP and thus $\beta \cup \{b\}$ and its extensions do not need to be considered.

Then, the algorithm considers $\beta = \{b\}$. Since $S(b) = 8 \geq minSup$ and $ML(c) = 0 \leq maxLa$ according to $PList_\alpha$, $\{b\}$ is a SPP and it is inserted in Q_k as a current top-k SPP. The queue Q_k now contains four itemsets: $\{e\}$ (support = 6, maximum lability = 1), $\{b, e\}$ (support = 5, maximum lability = 1), $\{b\}$ (support = 8, maximum lability = 0), and $\{c\}$ (support = 6, maximum lability = 0). Because Q_k contains more than $k = 3$ itemsets, the least frequent itemset $\{b, e\}$ is removed from Q_k and the $minSup$ threshold is raised to the support of the least frequent itemset in Q_k, which is now 6. Then, it is found that the conditional pattern base of $\{b\}$ is empty and it has no ancestors. Thus, extensions of this itemset are also not considered.

The algorithm then terminates and the queue Q_k contains the top-k SPPs, which in this example are: $\{e\}$ (support = 6, maximum lability = 1), $\{b\}$ (support = 8, maximum lability = 0), and $\{c\}$ (support = 6, maximum lability = 0).

Table 3 Calculations for mining the example SPP-tree containing items $\beta = \{e\}, \{c\}$ and $\{b\}$

β	γ	Conditional Pattern Base	SPP-list	$\gamma^{'}$	Conditional SPP-tree	SPPs
$\{e\}$	$\{b, c\}$	$\{bc : 1, 6, 7\}$ $\{b : 3, 5\}$ $\{c : 4\}$	$\{\{b, 5, 1, pt\}, \{c, 4, 0, pt\}\}$	$\{b\}$	$\{b : 1, 3, 5, 6, 7\}$	$\{be, 5, 1\}$
$\{c\}$	$\{b\}$	$\{b : 1, 2, 6, 7\}$	$\{\{b, 4, 2, pt\}\}$	-	-	-
$\{b\}$	-	-	-	-	-	-

4.4 Complexity Analysis

The TSPIN algorithm can be viewed as an extension of FP-Growth [23]. Hence, they both have a similar complexity, which is analyzed as follows. Let m be the number of transactions, n be the number

of distinct items, and p be the average length of the transactions. The first operation performed by TSPIN and FP-Growth is to scan the database to build a prefix-tree to store the transactions. Scanning the database and building the prefix-tree is done in linear time, as each transaction is inserted one by one and at most one path is added to the tree per transaction [23]. In terms of memory, the size of a prefix-tree is analyzed as follows. In the worst case, all the transactions will be different, and the prefix-tree will contain a distinct path for each transaction. Thus, the prefix-tree will contain $m \times p$ nodes and no more than $m \times p$ parent-child links between nodes. TSPIN stores an item and a list of up to m TIDs in each prefix-tree node, while FP-Growth only stores an item. Moreover, TSPIN creates a SPP-list for the prefix-tree which is an array containing at most n entries of constant size, where each entry is linked to the tree using up to m node-links. Instead of building an SPP-list, FP-Growth builds a header-table, which contains different information but has the same size complexity. On overall the size of the prefix-tree and SPP-list (or header table) is linear with respect to the database size ($m \times p$). Though the size of a prefix-tree may seems large, generally many transactions are similar in real databases. As a result, many paths of the prefix-tree will overlap, which reduces the number of nodes and links. Thus, the size of a prefix-tree is often much smaller than the original database [23].

After the initial prefix-tree construction, TSPIN and FP-Growth perform a depth-first search to recursively find all the desired itemsets. The number of possible itemsets is $h = 2^{|I|}$-1. But in practice the number of itemsets that is considered depends on the characteristics of the database, and the parameters of the algorithms. If *minSup*, *maxPer* or *maxLa* are increased, or if k is decreased, less itemsets may be considered due the application of the search space pruning strategies.

For each considered itemset β that extends an itemset α, TSPIN traverses the node-links of the SPP-list of α to create the conditional pattern base, SPP-list and prefix-tree of β. This construction is done in linear time as these structures of α are traversed once. The size of the three structures of β are bounded by the size of these structures for α, and can be much smaller. FP-Growth performs a similar process to build a conditional pattern-base, header-table and prefix-tree but those contains less information. The number of SPP-lists, prefix-trees and conditional pattern bases that are built is at most h. And it can be observed that as the depth-first search go deeper, conditional prefix-trees will become smaller [23].

An important difference between TSPIN and FP-Growth is that TSPIN maintains a queue Q_k to keep the top-k best patterns until now. This queue can contains up to h itemsets, and each itemset is inserted and removed at most once from that queue. The three operations performed on the priority queue are insertion, deletion and peek, which have worst-case $\mathcal{O}(\log h)$, $\mathcal{O}(\log h)$, and $\mathcal{O}(1)$ time complexity using a binomial heap [7]. Other implementations can also be considered such as using a Fibonacci heap to obtain an amortized time of $\mathcal{O}(1)$, $\mathcal{O}(\log h)$, and $\mathcal{O}(1)$ for insertion, deletion and peek [7].

Thus, on overall the main factors influencing the time and space complexity of TSPIN and FP-Growth are the number of itemsets that are considered h, and the database size. The time complexity of FP-Growth is roughly $\mathcal{O}(h \times m \times p)$ while that of TSPIN is roughly $\mathcal{O}(h \times m \times p \times log(h))$. The difference is due to the additional effort to manage the priority queue. The space complexity of FP-Growth and TSPIN is roughly the same as $\mathcal{O}(h \times m \times p \times h)$.

As for SPP-Growth, the complexity is roughly the same as same as that of FP-Growth as it does not have to manage a priority queue.

5 Experimental evaluation

To evaluate the proposed algorithm, extensive experiments have been done on a workstation running Windows 10, equipped with an Intel (R) Xeon(R) CPU W-2123 3.60GHz, and 32 GB of RAM. The TSPIN algorithm was implemented in Java, and runtime and peak memory usage were measured using the standard Java API. Four benchmark datasets have been used, which are described in Table 4, in terms of number of transactions ($|D|$), number of distinct items ($|I|$), minimum transaction length (T_{min}), maximum transaction length (T_{max}), and average transaction length (T_{avg}). These datasets were selected as they represent different types of data (long/short transactions, few/many items, and dense/sparse data).

T10I4D100K is a synthetic dataset, which was created using the SPMF library's random transaction generator [10]. The dataset contains 100,000 transactions and 870 distinct items. *Mushroom* is a dataset about mushrooms, often used in benchmarks for itemset mining. It contains 8,124 transactions and 119 distinct items. *Kosarak* is 990,002 transactions of real click-stream data from a Hungarian news portal. It is a very large dataset having 41,270 distinct items. *OnlineRetail* is a real-world shopping dataset where each transaction indicates items purchased together by a customer. *OnlineRetail* contains

Table 4 Description of the datasets

Dataset	$\|D\|$	$\|I\|$	T_{min}	T_{max}	T_{avg}
T10I4D100K	100,000	870	1	29	10
Mushroom	8,124	119	23	23	23
Kosarak	990,002	41,270	1	2,498	8
OnlineRetail	541,909	2,603	1	1,108	23

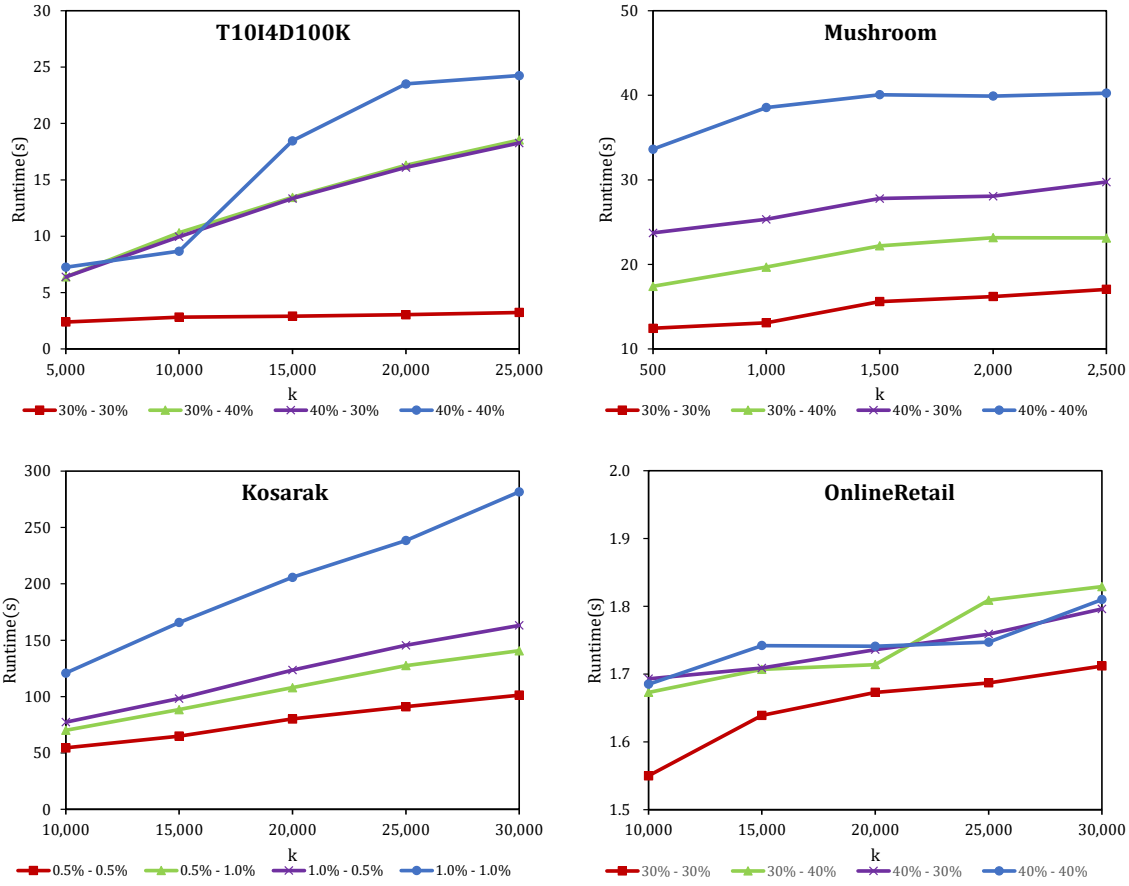

Fig. 4 Execution times for different parameter values

541,909 transactions and 2,603 distinct items. All datasets can be downloaded from the SPMF website (http://www.philippe-fournier-viger.com/spmf/).

To calculate the periodicity of itemsets, two approaches are used. For the $T10I4D100K$, $Mushroom$ and *kosarak* datasets, the periodicity is calculated using the transaction identifiers as in the definitions of Section 2 and 3. The first transaction of a dataset has the identifier 1, the second transaction has the identifier 2, and so on. For the *OnlineRetail* dataset, transaction timestamps are used to calculate the periodicity (other datasets do not have timestamps). To do this, the transaction identifiers are simply replaced by the timestamps (as explained in Section 2). Transactions timestamps in *OnlineRetail* range from 2010-12-1 8:26 to 2011-12-9 12:50, where the time unit is the minute. Using real timestamps instead of transaction identifiers was done because it can lead to discovering more meaningful patterns in customer transactions.

5.1 **Influence of parameters on the performance of TSPIN**

In a first experiment, the parameters $maxPer$, $maxLa$ and k were varied to evaluate their influence on the performance of TSPIN in terms of runtime and peak memory usage on the $Mushroom$ and $T10I4D100K$ datasets. Runtime results are shown in Fig. 4 and peak memory usage is shown in Table 5.

In Figure 4, k values are shown on the x axis, while the y axis denotes excution time. The notation P-L denotes the TSPIN algorithm with $maxPer = P$ and $maxLa = L$. The following observations are drawn from that figure:

- Increasing k often increases the runtime. This is reasonable since as k is increased, more patterns are found, and more itemsets from the search space may have to be considered to find the top-k SPPs. As a result, TSPIN may need to consider more patterns to fill Q_k.
- Increasing $maxPer$ or $maxLa$ often increases the runtime. The reason is that increasing $maxPer$ or $maxLa$ will increase the range of period and lability values accepted for SPPs. Hence, more itemsets may have to be considered from the search space to find the top-k SPPs.
- The TSPIN algorithm has better performance on the sparse dataset than on the dense dataset. The reason is that itemsets in the sparse datasets are more likely to be unstable. Thus, the TSPIN algorithm can eliminate many candidate patterns for such datasets.

Table 5 Comparison of peak memory usage.

$T10I4D100K$				$Mushroom$			
$maxPer$	$maxLa$	k	Peak memory	$maxPer$	$maxLa$	k	Peak memory
30%	30%	10,000	611	30%	30%	1,000	2103
30%	30%	20,000	607	30%	30%	2,000	2658
30%	40%	10,000	2206	30%	40%	1,000	2641
30%	40%	20,000	2699	30%	40%	2,000	2667
40%	30%	10,000	2207	40%	30%	1,000	2633
40%	30%	20,000	2699	40%	30%	2,000	2633
40%	40%	10,000	2191	40%	40%	1,000	2704
40%	40%	20,000	2748	40%	40%	2,000	2704
$Kosarak$				$OnlineRetail$			
$maxPer$	$maxLa$	k	Peak memory	$maxPer$	$maxLa$	k	Peak memory
0.5%	0.5%	10,000	1640	30%	30%	10,000	263
0.5%	0.5%	20,000	1780	30%	30%	20,000	264
0.5%	1.0%	10,000	1985	30%	40%	10,000	264
0.5%	1.0%	20,000	2419	30%	40%	2,000	264
1.0%	0.5%	10,000	1726	40%	30%	10,000	264
1.0%	0.5%	20,000	2317	40%	30%	20,000	268
1.0%	1.0%	10,000	2494	40%	40%	10,000	264
1.0%	1.0%	20,000	2166	40%	40%	20,000	265

The peak memory usage of TSPIN is shown in Table 5, for different parameter values on the four datasets. The following observations are drawn from Table. 5:

- The TSPIN algorithm consumes more memory as k is increased. This is because when k is set to larger values, TSPIN generally need to consider more itemsets to fill Q_k. Moreover, the size of Q_k also contributes to the peak memory usage.
- The TSPIN algorithm consumes more memory when increasing $maxPer$ or $maxLa$. The reason is that when $maxPer$ or $maxLa$ are increased, the range of period and lability values may increase. The number of nodes in the SPP-tree may thus increase, and the size of the SPP-tree also contributes to the peak memory usage.

5.2 **Influence of the number of transactions on TSPIN's performance**

We also evaluated the proposed algorithm's scalability in terms of execution time and number of tree nodes when the number of transactions is varied. For this experiment, the real-world dataset *kosarak* is used, since it has a large number of distinct items and transactions. The dataset was divided into five parts and the performance of the algorithm was measured after adding each part to the previous ones. Figure 5 shows the experiment's results for $maxPer = 1\%$, $maxLa = 1\%$ and $k = 10,000$.

It is clear that the execution time and number of nodes increase along with the database size. This is reasonable because the number of itemsets may be greater in a larger database, and tail nodes may have to store longer lists of transaction IDs (timestamps). Hence, the algorithm may spend time for evaluating additional itemsets, and more time for building the SPP-tree.

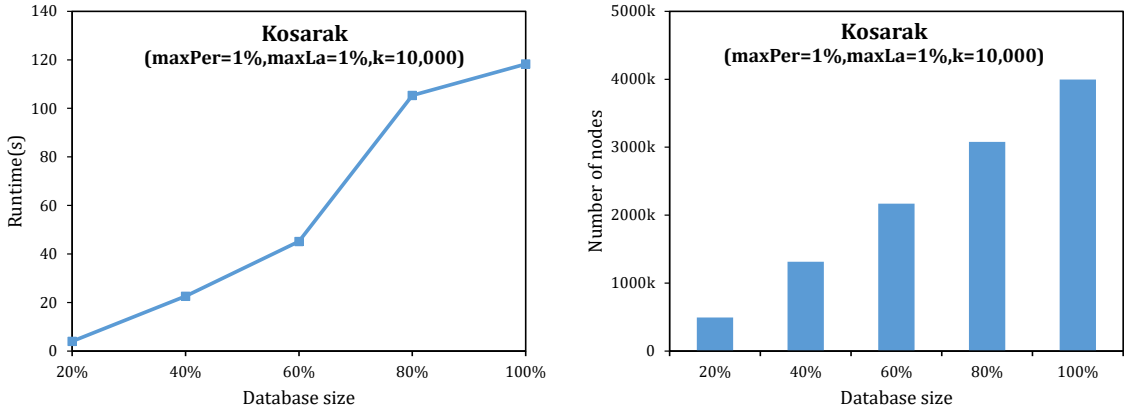

Fig. 5 Scalability of SPP-growth when variying the database size

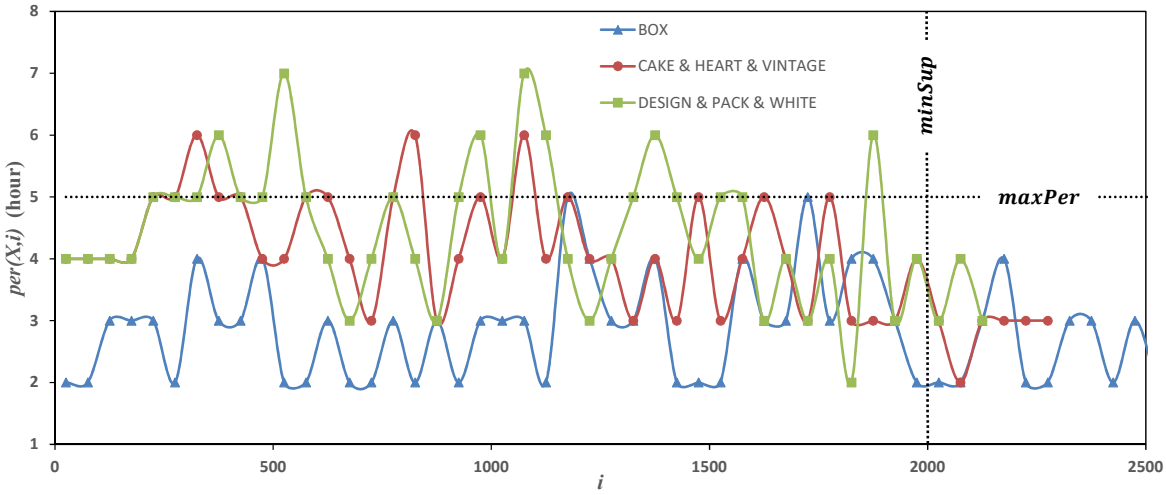

Fig. 6 Periods of some interesting SPPs found in *OnlineRetail*

5.3 Analysis of some interesting SPPs found in customer transaction data

We also analyzed the SPPs found in the real-world *OnlineRetail* dataset to see if the proposed algorithm allows finding interesting patterns. *OnlineRetail* contains transactions of a UK-based online store from 01/12/2010 to 09/12/2011. The data was segmented into hours to obtain 2,975 non empty transactions. Figure 6 shows SPPs that have been found, which are {Box}: (2553, 0h), {Cake, Heart, Vintage}: (2284, 1h) and {Design, Pack, White}: (2131, 2h), where each SPP X is annotated with ($sup(X)$, $maxla(X)$). The X-axis indicates period numbers of patterns, and the Y-axis indicates the value $per(X, i)$ for the i-th period. Note that to reduce the number of points on that chart, only the maximum value for each group of 50 periods is shown in Figure 6. It can be observed that frequent patterns exceeding *maxPer* having a stable periodic behavior are obtained, while such patterns would be ignored by traditional PFP mining algorithms due to the strict *maxPer* constraint. The stable patterns found about the sale of products are also deemed interesting as they indicate stable sale trends. Such information could be used to forecast product sales.

5.4 Performance comparison with SPP-Growth set with an optimal *minSup* threshold

This journal extends a conference paper where an early version of TSPIN was presented, called SPP-Growth [17]. The main difference between TSPIN and SPP-Growth is that TSPIN let the user directly choose the number of patterns k to be found, while SPP-Growth requires that the user sets a *minSup* threshold. Using the parameter k instead of *minsup* was proposed because finding an appropriate value for the *minSup* parameter whithout having background knowledge about a dataset is a process of trial

and error, which can be time-consuming. By letting the user set k, the user can directly specify how many patterns to be found. Though, using k is more intuitive, the resulting problem of top-k SPP mining is much more difficult than that of mining all SPPs using a *minSup* threshold. The reason is that in top-k SPP mining, the search for patterns must start from $minSup = 1$, while in SPP mining the *minSup* threshold is fixed beforehand by the user, and thus SPP-Growth can directly use that threshold to reduce the search space.

Because TSPIN and SPP-Growth have different parameters and are designed for two different problems, it is difficult to compare them. To still do a comparison, a scenario was considered where SPP-Growth was run with an optimal *minsup* value to generate the same number of itemsets as TSPIN. The goal of this experiment is to evaluate if top-k stable periodic pattern mining using TSPIN can have a similar performance to mining stable periodic patterns using SPP-Growth. That question is interesting since top-k stable periodic pattern mining is a more difficult problem than stable periodic pattern mining. For this experiment, both TSPIN and SPP-Growth were implemented in Java, and use the same code for loading datasets, constructing an SPP-tree, mining an SPP-tree and saving patterns.

TSPIN was run with $maxPer = 30\%$ and $maxLa = 30\%$, while k was varied from 5,000 to 25,000 on the $T10I4D100K$ dataset, 500 to 2,500 on the $Mushroom$ dataset, and 10,000 to 30,000 on the $OnlineRetail$ dataset. TSPIN was also run with $maxPer = 0.5\%$ and $maxLa = 0.5\%$, while k was varied from 10,000 to 30,000 on the $Kosarak$ dataset. Then, the SPP-growth algorithm was run with the optimal *minSup* value to obtain the same number of itemsets. Runtime and peak memory usage were measured. Tables 6 to 9 show results for the four datasets, respectively.

From these results, it is found that TSPIN takes more time than SPP-growth on $Mushroom$ and $Kosarak$ while TSPIN and SPP-growth have very similar runtime on $T10I4100K$ and $OnlineRetail$. This is considered a good result since the problem of top-k stable periodic pattern mining is more difficult than the problem of stable periodic pattern mining using *minSup*.

In terms of memory, TSPIN generally consumes more memory than SPP-growth (up to 15 times more on the $Mushroom$ dataset and up to 1.5 times on $T10I4100K$ and $Kosarak$). This is reasonable since TSPIN needs to keep a priority queue Q_k to store the current top-k patterns.

It is important to note that this experiment was done by setting an optimal *minSup* value for SPP-growth to obtain the same number of patterns as TSPIN. But in real-life, the user typically don't know how to set *minSup* threshold. Selecting a value for k is more intuitive than setting *minSup* because the former represents the number of patterns that the user wants to find. If the user sets the *minSup* threshold too low, SPP-growth may find too many patterns and may become very slow, while if the threshold is set too high, the user may need to run the algorithm again until a suitable value is found, which is time-consuming. To avoid such trial-and-error approach to find a suitable *minSup* value, this paper has proposed the TSPIN algorithm, which let the user directly specify the number of patterns to be found. Because the runtime of TSPIN is close to that of SPP-growth on $T10I4D100K$ and $OnlineRetail$, TSPIN can be considered as a valuable alternative to SPP-growth, especially for sparse datasets.

Table 6 Comparison of TSPIN and SPP-Growth with optimal *minSup* threshold on the *T10I4D100K* dataset

k	*minSup*	Runtime (s)		Memory (MB)	
		TSPIN	SPP-Growth	TSPIN	SPP-Growth
5,000	292	2.397	2.020	522	354
10,000	227	2.833	2.185	611	366
15,000	183	2.914	2.194	608	416
20,000	141	3.045	2.287	607	464
25,000	111	3.239	2.366	617	525

Table 7 Comparison of TSPIN and SPP-Growth with optimal *minSup* threshold on the *Mushroom* dataset

k	*minSup*	Runtime (s)		Memory (MB)	
		TSPIN	SPP-Growth	TSPIN	SPP-Growth
500	3312	12.447	0.385	2104	96
1,000	2960	13.109	0.505	2103	127
1,500	2736	15.605	0.628	2661	127
2,000	2592	16.204	0.707	2658	127
2,500	2496	17.059	0.812	2653	127

Table 8 Comparison of TSPIN and SPP-Growth with optimal *minSup* threshold on the *Kosarak* dataset

k	*minSup*	Runtime (s)		Memory (MB)	
		TSPIN	SPP-Growth	TSPIN	SPP-Growth
10,000	2369	54.563	13.519	1640	1646
15,000	2150	65.015	15.524	1755	1295
20,000	2087	80.262	16.016	1780	1185
25,000	2045	91.088	17.446	1845	1202
30,000	2019	101.342	17.880	1853	1191

Table 9 Comparison of TSPIN and SPP-Growth with optimal *minSup* threshold on the *OnlineRetail* dataset

k	*minSup*	Runtime (s)		Memory (MB)	
		TSPIN	SPP-Growth	TSPIN	SPP-Growth
10,000	371	1.550	1.279	263	139
15,000	244	1.639	1.463	265	173
20,000	177	1.673	1.529	264	202
25,000	131	1.687	1.399	264	223
30,000	97	1.712	1.438	265	244

6 Conclusion

Traditional algorithms for mining periodic patterns have three main limitations: (1) they discard a pattern as non periodic if only one period exceed *maxPer*, (2) they do not consider by how much a period exceeds that threshold, and (3) they generally requires to set a *minSup* threshold that is hard to set. This paper addressed these limitations by proposing a novel problem of mining the top-k stable periodic-frequent patterns in a sequence of transactions (events). A new *lability* function was defined to identify patterns that have a stable periodic behavior. A pattern-growth algorithm named TSPIN was designed to efficiently find the top-k SPPs. An experimental evaluation on both synthetic and real datasets have shown that TSPIN is efficient and can find useful patterns in customer transaction data.

For future work, we plan to adapt the concept of stability to mine other types of patterns such as sequential patterns [13] and uncertain patterns that are stable. Besides, we intend to develop applications around the proposed model of stable periodic patterns to evaluate in more details their usefulness. Another interesting possibility for future work is to design alternative models for identifying stable periodic patterns that would also address the three above limitations.

Declarations

Acknowledgements. This study was partly funded by the National Natural Science Foundation of China and the Harbin Institute of Technology.

Conflicts of interest/Competing interests. The authors declare that they have no conflict of interest and competing interests.

Availability of data and material, and code availability. The code and datasets will be integrated in the SPMF data mining library (http://www.philippe-fournier-viger.com/spmf) following the article acceptance.

References

1. Afriyie, MK, Nofong, VM, Wondoh, J, Abdel-Fatao, H (2020). Mining Non-redundant Periodic Frequent Patterns. In: Proceedings of the 12th Asian Conference on Intelligent Information and Database Systems. Springer, pp 321-331
2. Agrawal R, Imielinski T, Swami AN (1993) Mining Association Rules Between Sets of Items in Large Databases. In: Proceedings of the 19th ACM SIGMOD International Conference on Management of Data. ACM, pp 207-216
3. Amphawan K, Lenca P, Surarerks A (2009) Mining top-k periodic-frequent pattern from transactional databases without support threshold, In: Proceedings of the 3rd International Conference on Advances in Information Technology. pp 18-29
4. Amphawan K, Surarerks A, Lenca P (2010) Mining periodic-frequent itemsets with approximate periodicity using interval transaction-ids list tree. In: Proceedings of the 3rd International Conference on Knowledge Discovery and Data Mining. pp 245–248.
5. Bodon F, Schmidt-Thieme L (2005) The Relation of Closed Itemset Mining, Complete Pruning Strategies and Item Ordering in Apriori-Based FIM Algorithms. In: Proceedings of the 9th European Conference on Principles and Practice of Knowledge Discovery in Databases. ACM, pp 437-444

6. Chon KW, Hwang SH, Kim MS (2018) GMiner: A fast GPU-based frequent itemset mining method for large-scale data. Information Sciences 1(439):19-38
7. Cormen TH, Leiserson CE, Rivest RL, Stein C (2009) Introduction to algorithms. MIT press
8. Dinh DT, Le B, Fournier-Viger P, Huynh VN (2018) An efficient algorithm for mining periodic high-utility sequential patterns. Applied Intelligence 48(12):4694-4714
9. Fong ACM, Zhou B, Hui SC, Hong GY, Do T (2011) Web content recommender system based on consumer behavior modeling. IEEE Transactions on Consumer Electronics 57(2):962-969
10. Fournier-Viger P, Gomariz A, Gueniche T, Soltani A, Wu C, Tseng VS (2014) SPMF: a Java Open-Source Pattern Mining Library. The Journal of Machine Learning Research 15(1):3389-3393
11. Fournier-Viger P, Lin JCW, Duong QH, Dam TL, Sevcik L, Uhrin D, Voznak M (2017) PFPM: discovering periodic frequent patterns with novel periodicity measures. In: Proceedings of the 2nd Czech-China Scientific Conference 2016. IntechOpen.
12. Fournier-Viger P, Lin JCW, Duong QH, Dam TL (2016) PHM: Mining Periodic High-Utility Itemsets. In: Proceedings of the Industrial Conference on Data Mining. pp 64-79
13. Fournier-Viger P, Lin JCW, Kiran RU, Koh YS, Thomas R (2017) A Survey of Sequential Pattern Mining. Data Science and Pattern Recognition 1(1):54-77
14. Fournier-Viger P, Lin JCW, Truong-Chi T, Nkambou R (2019) A Survey of High Utility Itemset Mining. In High-Utility Pattern Mining (pp. 1-45). Springer, Cham.
15. Fournier-Viger P, Lin JCW, Vo B, Truong TC, Zhang J, Le HB (2017) A survey of itemset mining. Wiley Interdisciplinary Reviews: Data Mining and Knowledge Discovery 7(4):e1207
16. Fournier-Viger P, Li Z, Lin, JCW, Kiran RU, Fujita H (2018) Discovering Periodic Patterns Common to Multiple Sequences. In: Proceedings of the 20th International Conference on Data Warehousing and Knowledge Discovery. Regensburg: Springer, pp 231-246
17. Fournier-Viger P, Yang P, Lin JCW, Kiran RU (2019) Discovering Stable Periodic-Frequent Patterns in Transactional Data. In: Proceedings of the 32nd International Conference on Industrial, Engineering and Other Applications of Applied Intelligent Systems. Springer, pp 230-244
18. Fournier-Viger P, Wu CW, Zida S, Tseng, VS (2014) FHM: Faster high-utility itemset mining using estimated utility co-occurrence pruning. In: Proceedings of the 21st International Symposium on Methodologies for Intelligent Systems. pp 83-92
19. Fournier-Viger P, Yang P, Lin C, Yun U (2019) HUE-Span: Fast High Utility Episode Mining. In: Proceedings of the 14th International Conference on Advanced Data Mining and Applications. pp 169-184
20. Gama, J, Zliobaite, I, Bifet, A, Pechenizkiy, M, Bouchachia, H (2014) A Survey on Concept Drift Adaptation. Journal of ACM Computing Surveys, 46(4):1–37
21. Gouda K, Zaki MJ (2001) Efficiently mining maximal frequent itemsets. In: Proceedings of the 17th IEEE International Conference on Data Mining. ACM, pp 163-170
22. Grahne G, Zhu J (2005) Fast algorithms for frequent itemset mining using fp-trees. IEEE transactions on knowledge and data engineering, 17(10):1347-1362
23. Han J, Pei J, Yin Y, Mao R. (2000) Mining frequent patterns without candidate generation: A frequent-pattern tree approach. Data mining and knowledge discovery, 8(1):53-87
24. Huang K, Chang C (2008) Efficient mining of frequent episodes from complex sequences. Information Systems, 33:96-114
25. Huang, Y, Hsu, CL, Tseng, VS (2020) PURL: Periodic user representation learning from temporal event records for personalized health management. In: Proceedings of the 7th IEEE International Conference on Big Data and Smart Computing. IEEE, pp 358-365
26. Islam, MA, Acharjee, UK (2020) Mining Periodic Patterns and Accuracy Calculation for Activity Monitoring Using RF Tag Arrays. In: Proceedings of the International Joint Conference on Computational Intelligence. Springer, pp 85-95
27. Kiran RU, Kitsuregawa M, Reddy PK (2016) Efficient discovery of periodic-frequent patterns in very large databases. Journal of Systems and Software 112:110-121
28. Kiran RU, Reddy PK (2010) Mining rare periodic-frequent patterns using multiple minimum supports. In: Proceedings of the 15th International Conference on Management of Data. pp 7-8
29. Kiran, RU, Saideep, C, Zettsu, K, Toyoda, M, Kitsuregawa, M., Reddy, PK (2019) Discovering Partial Periodic Spatial Patterns in Spatiotemporal Databases. In: Proceedings of the 2019 IEEE International Conference on Big Data. IEEE, pp 233-238
30. Kiran RU, Venkatesh JN, Fournier-Viger P, Toyoda M, Reddy PK, Kitsuregawa M (2017) Discovering Periodic Patterns in Non-uniform Temporal Databases. In: Proceedings of the 21th Pacific-Asia Conference on Knowledge Discovery and Data Mining (2). pp 604-617
31. Koh YS, Ravana SD (2016) Unsupervised rare pattern mining: a survey. ACM Transactions on Knowledge Discovery from Data 10(4):45
32. Kumar V, Valli Kumari V (2013) Incremental mining for regular frequent patterns in vertical format. Int. J. Eng. Tech. 5(2):1506–1511
33. Li H, Hai M, Zhang N, Zhu J, Wang Y, Cao H (2019) Probabilistic maximal frequent itemset mining methods over uncertain databases. Intelligent Data Analysis 23(6):1219-1241
34. Luna, JM, Fournier-Viger, P, Ventura, S (2019) Frequent Itemset Mining: a 25 Years Review. WIREs Data Mining and Knowledge Discovery. Wiley, 9(6):e1329.
35. Manku GS. (2016) Frequent itemset mining over data streams. In Data Stream Management (pp. 209-219). Springer, Berlin, Heidelberg.
36. Mannila H, Toivonen H, Verkamo AI (1995) Discovering Frequent Episodes in Sequences. In: Proceedings of the First International Conference on Knowledge Discovery and Data Mining. pp 210-215
37. Minato SI, Uno T, Arimura H (2008) LCM over ZBDDs: Fast generation of very large-scale frequent itemsets using a compact graph-based representation. In: Proceedings of the Pacific-Asia Conference on Knowledge Discovery and Data Mining. pp 234-246
38. Muthukrishnan, S, Berg, EVD, Wu, Y (2007 Sequential Change Detection on Data Streams. In: Proceedings of the 7th IEEE Intern. Conf. on Data Mining Workshops. pp. 551-550

39. Nofong VM (2015) Discovering Productive Periodic Frequent Patterns in Transactional Databases. In: Proceedings of the Second International Conference on Data Science. pp 141-150
40. Nofong VM (2018) Fast and Memory Efficient Mining of Periodic Frequent Patterns. In: Proceedings of the 10th Asian Conference onModern Approaches for Intelligent Information and Database Systems. pp 223–232
41. Pasquier N, Bastide Y, Taouil R, Lakhal L (1999) Discovering frequent closed itemsets for association rules. In: Proceedings of the 7th International Conference on Database Theory. ACM, pp 398-416
42. Rashid MM, Gondal I, Kamruzzaman J (2013) Regularly frequent patterns mining from sensor data stream. In: Proceedings of the 20th International Conference on Neural Information Processing. pp 417–424
43. Rashid MM, Karim MR, Jeong BS, Choi HJ (2012) Efficient mining regularly frequent patterns in transactional databases. In: Proceedings of the 17th International Conference on Database Systems for Advanced Applications, pp 258–271
44. Surana A, Kiran RU, Reddy PK (2012) An Efficient Approach to Mine Periodic-Frequent Patterns in Transactional Databases. In: Proceedings of the 16th Pacific-Asia Conference on Knowledge Discovery and Data Mining. pp 254-266
45. Tanbeer SK, Ahmed CF, Jeong BS, Lee YK (2009) Discovering periodic-frequent patterns in transactional databases. In: Proceedings of the 13rd Pacific-Asia Conference on Knowledge Discovery and Data Mining. pp 242-253
46. Tong YX, Chen L, She J (2015) Mining frequent itemsets in correlated uncertain databases. Journal of Computer Science and Technology 30(4):696-712
47. Truong-Chi T, Fournier-Viger P (2019) A survey of high utility sequential pattern mining. In High-Utility Pattern Mining (pp. 97-129). Springer, Cham.
48. Wong MH, Tseng VS, Tseng JC, Liu SW, Tsai CH (2017). Long-term user location prediction using deep learning and periodic pattern mining. In: Proceedings of the 12th International Conference on Advanced Data Mining and Applications. pp 582-594.
49. Yun U, Kim D, Yoon E, Fujita H (2018) Damped window based high average utility pattern mining over data streams. Knowledge-Based Systems 144:188-205
50. Yun U, Lee G, Yoon E (2017) Efficient high utility pattern mining for establishing manufacturing plans with sliding window control. IEEE Transactions on Industrial Electronics 64(9): 7239-7249
51. Zaki, MJ, Gouda, K (2003) Fast vertical mining using diffsets. In: Proceedings of the 9th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining. ACM, pp 326-335
52. Zhang R, Chen W, Hsu TC, Yang H, Chung YC (2019) ANG: a combination of Apriori and graph computing techniques for frequent itemsets mining. The Journal of Supercomputing. 6(2):646-61
53. Zhang, D, Lee, K, Lee, I (2019) Mining hierarchical semantic periodic patterns from GPS-collected spatio-temporal trajectories. Expert Systems with Applications. 122: 85-101.