An Inverted-ITL Algorithm for Mining Partial Periodic-Frequent Patterns

Ye-In Chang, Xin-Long Chen, and Sheng-Hsin Chiang

Abstract In this paper, we propose an *Inverted-ITL* algorithm for mining partial periodic-frequent patterns. Although the *GPF-growth* algorithm has been proposed to achieve the same goal, it needs to construct the prefix tree which needs too much storage space and processing time. Moreover, the *GPF-growth* algorithm will scan the database twice and sort each transactional list. To avoid those disadvantages, in this paper, we propose the *inverted-ITL* algorithm. Our algorithm is more efficient than the *GPF-growth* algorithm. We will use one data structure, the *ITL-tree*, to store the items which appear in the database. Therefore, our algorithm can need shorter processing time than the *GPF-growth* algorithm.

Keywords Frequent patterns · Itemsets mining · Periodic patterns · Periodic-frequent patterns · Transactional database

1 Introduction

In recent years, the research of data mining [1, 2] has become more popular. There are many techniques for mining interesting patterns in the database, like frequent pattern mining [3–5], frequent weighted pattern mining [6], frequent closed pattern mining [7], maximal frequent pattern mining [8], and periodic-frequent pattern mining [9–13]. Among the above techniques, current researches on periodic-frequent pattern mining have focused on discovering full periodic-frequent patterns in the database. However, partial periodic-frequent patterns are more common in the real world, i.e., finding frequent patterns which frequently occur but do not successively occur one after one in the database. The reason for this phenomenon is the imperfect nature of the real-world.

Y.-I. Chang (✉) · X.-L. Chen · S.-H. Chiang
Department of Computer Science and Engineering, National Sun Yat-Sen University, Kaohsiung, Taiwan
e-mail: changyi@mail.cse.nsysu.edu.tw

© The Author(s), under exclusive license to Springer Nature Switzerland AG 2022

C.-C. Wang, A. Nallanathan (eds.), *Proceedings of the 5th International Conference on Signal Processing and Information Communications*, Signals and Communication Technology, https://doi.org/10.1007/978-3-031-13181-3_10

The periodic-frequent pattern mining is an extension of the frequent pattern mining. In the frequent pattern mining, the count of the support is the factor which we only care about. However, in the periodic-frequent pattern mining, we do not just care about the frequency of each item but also need to make sure that the frequent pattern occurs periodically in the transactional database [9]. According to the property of the transactional database, the database consists of two fields: (1) timestamp and (2) item. The timestamp denotes the current time, when a transaction list is generated. For the topic of mining periodic-frequent patterns, the item will not appear repeatedly in a transaction list. Moreover, we do not have to care about the number of the items in a transaction list. That is, a transaction list $\{a, a, b, c\}$ will not appear. Moreover, $\{a, b, c\}$ and $\{b, c, a\}$ are the same. The timestamp is used to record the time, when a transaction list is generated and the transaction lists are sorted according to the ascending order of the timestamp. Therefore, for the entire database, there may exist time that no transaction list is generated. For instance, the list of transactions may contain [*T*1, *T*2, *T*3, *T*5, *T*6], where time *T*4 does not appear.

Take Table 1 as an example, which is denoted by transactional database TDB1. Assume that the user-specified minimum support is 2, and the user-specified maximum period is 3. At first, we scan all the transactions of transactional database TDB1 to count the total support and obtain the maximum period of the complete set of periods for each item. We can exploit these data items to discover which patterns are periodic-frequent patterns. Figure 1 shows the result after scanning database TDB1, which is denoted by database TDB2. In database TDB2, the symbols *ItemN*, *CountN*, and *MaxD* denote the item name, the count of the total support, and the maximum period of the complete set of periods for each item, respectively.

According to the data of transactional database TDB2, we can know the support and the maximum period of each item in the database. Take item *a* as an example. *CountN* is larger than or equal to 2, and *MaxD* is lower than or equal to 3. So item "a" is a periodic-frequent pattern. Let's focus on item "c," which *CountN* is not less than 2, but its *MaxD* is larger than 3, so item *c* is not a periodic-frequent pattern. Next, for item *i*, it is not a periodic-frequent pattern for the same reason as item *c*. Although its *CountN* is larger than 2, its *MaxD* is larger than 3. Then, items *j* and *g* are not periodic-frequent patterns, because they have the same reason as the items *c* and *i*. That is, the *MaxD* is larger than the threshold. Finally, item "h" also is not a periodic-frequent pattern. Because its *CountN* and *MaxD*, neither of them meet the threshold. Based on the definition of *anti-monotone*, pruning these items will not affect the result. Therefore, the result after the pruning step contains items *a*, *b*, *d*, *e*, and *f*.

Table 1 An example of the transactional database TDB1

ts	Items	ts	Items
1	*ab*	6	*degf*
2	*acdi*	7	*abi*
3	*cefij*	8	*cdej*
4	*abfgh*	9	*abef*
5	*bd*	10	*acgi*

ItemN	CountN	MaxD
√a	6	3
b	5	3
*c	4	5!
d	4	3
*i	4	4!
e	4	3
f	4	3
*j	2	5!
*g	3	4!
*h	1!!	6!

! : larger than 3
!! : lower than 2
* : not periodic-frequent pattern

Fig. 1 After scanning all transactions of database TDB1, called database TDB2 (the minimum support = 2, the maximum period = 3)

Next, we combine those items into the new candidate size 2 patterns. Then, after checking, the qualified patterns are *ab* and *ef*. In the same way, we combine the periodic-frequent size 2 patterns into the new candidate size 3 patterns to discover the periodic-frequent size 3 patterns. The similar process is repeated until no periodic-frequent patterns are generated.

Recently, Kiran et al. have proposed the *PFP-growth*++ algorithm [10], a prefix tree-based algorithm. They use the prefix tree with the unique timestamp to mine the periodic-frequent patterns. However, they do not take the real-world situation into account. Because in their algorithm, as long as the pattern does not meet the threshold once, it will not be considered as the periodic-frequent pattern. Later, Kiran et al. proposed the *GPF-growth* algorithm [12], which is also a prefix tree-based algorithm. The *GPF-growth* algorithm [12] considers fault tolerance by adding parameter periodic ratio. It can effectively solve the problems encountered by the *PFP-growth*++ algorithm [10].

The *GPF-growth* algorithm [12] can find more realistic patterns than the *PFP-growth*++ algorithm [10]. The *GPF-growth* algorithm considers about the fault tolerance of each pattern. However, we think that the construction of the prefix tree of *GPF-growth* needs too much time and memory space because the *GPF-growth* algorithm will scan the database twice and reorder each transactional list. Moreover, it constructs multiple prefix trees. Therefore, in this paper, we propose the *Inverted-ITL* algorithm to reduce the processing time. In our algorithm, we just only scan database once and store information of each pattern in two data structures. Later, we use these structures to find the periodic-frequent patterns. Therefore, our algorithm can need less processing time than the *GPF-growth* algorithm. Note that we do not need too much time to sort each transactional list and construct multiple prefix trees. From our performance study, we show that the performance of our algorithm is more efficient than that of the *GPF-growth* algorithm.

2 Related Works

The Apriori algorithm [3] has a great start and contribution to the research of data mining with frequent patterns. However, the algorithm is hard to achieve good performance. Later, Han et al. propose the *FP-growth* algorithm [5] for mining frequent patterns which scans twice in the database, and the algorithm exploits a tree structure, called FP-tree with the count of each item in the FP-tree. Next, Deng et al. propose the *PrePost* algorithm [4] and the *dFIN* algorithm [4] for mining frequent patterns with the property of scanning the database twice. Moreover, the algorithm exploits a tree structure, called PPC-tree which is an extension of the FP-tree and a vertical data structure, called N-list for each item in the PPC-tree. There are many algorithms for data mining which are developed based on this structure, such as the *PFP-growth* algorithm [14], the *NAFCP* algorithm [7], the *INLA-MFP* algorithm [8], and the *NFWI* algorithm [6]. Although many algorithms have been proposed as mentioned above, none of them have considered the period. Kiran et al. have proposed the *PFP-growth*++ algorithm [10] for mining periodic-frequent patterns, which considers both the frequency and the period for each item. Later, Kiran and Reddy use the simplified model [11] to find all frequent patterns which have exhibited complete cyclic repetitions in the database. Kiran et al. propose the *MCPF-model* [13] to discover periodic-frequent patterns involving both frequent and rare items effectively. Kiran et al. [9, 10] have discussed greedy search techniques to discover periodic-frequent patterns effectively. All of these researches have focused on finding full periodic-frequent patterns. Then, Kiran et al. propose the *GPF-growth* algorithm [12], which considers about the fault tolerance for each item. The whole algorithm is basically the same as the *PFP-growth*++ algorithm. The difference is the way to decide whether a pattern is a periodic-frequent pattern, which will be determined according to the percentage of the interest period. The components that make up the *GPF-growth* algorithm are the GPF-list and the GPF-tree. The *GPF-growth* algorithm forms the GPF-list by scanning the database once. Then, the GPF-list is sorted in the descending order of support for each item. Finally, they recursively mine the GPF-tree to discover the complete set of partial periodic-frequent patterns. Therefore, the *GPF-growth* algorithm needs to scan the database twice, sort the items once and the database once, and build a tree structure to find a complete set of the partial periodic-frequent patterns.

3 The Inverted-ITL Algorithm

In this section, we will introduce our algorithm called the Inverted-ITL (Inverted-Item-Time-List) algorithm to mine the partial periodic-frequent patterns.

3.1 Data Structure

In this subsection, we will use an example database TDB3 to explain our algorithm. Note that there is no transaction with $ts = 5$. During the process of partial periodic-frequent pattern mining, we exploit the support, the interesting period, and the periodic-ratio as main factors to determine whether the pattern is the partial periodic-frequent pattern or not. Thus, we use the *ITL-tree* and the *ITL-list* to record the information for each item. Note that we only have to scan the database once to record the information and we do not need to do any sorting operation with the database. During the mining process, we will frequently use these data structures to find the partial periodic-frequent patterns. Table 2 shows an example database for timestamp *ts*, and Table 3 shows the variables used in our algorithm. Next, we will illustrate the two data structures which we will use them in our algorithm.

The *ITL-tree*, a tree structure, is used to store all the patterns which appear in the database. It is a prefix tree, where each node may become a partial periodic-frequent pattern, but the root is an empty pattern. All k-length patterns are stored at level k of the tree, where $k \geq 1$. For instance, in Fig. 2, square boxes represent noncandidate patterns, dotted circles represent the candidate patterns, and solid circles represent the partial periodic-frequent patterns. Moreover, in the *ITL-tree*, once the pattern is confirmed as a noncandidate pattern, its super-set pattern will not appear. In other words, no pattern will generate new patterns with noncandidate patterns. Finally, the patterns in the solid circle are the partial periodic-frequent patterns that we want to find.

Table 2 An example of the transactional database TDB3

ts	Items	ts	Items
1	*ac*	8	*cdfg*
2	*abg*	9	*ab*
3	*de*	10	*cdef*
4	*abcd*	11	*abcef*
6	*abcd*	12	*abcd*
7	*abe*	13	*cef*

Table 3 Variables

Variable	Definition
ItemN	The name of the pattern
CountN	The count of the pattern
RangeC	The number of the pattern which is not larger than MaxPer during the period
FirstT	The first timestamp of the pattern
LastT	The last timestamp of the pattern
BP	Representing the timestamp of the bit pattern
Mark	A Boolean flag to check whether the last period of the pattern is not larger than the MaxPer
MinSup	The minimum support threshold
MinPR	The minimum periodic-ratio threshold
MaxPer	The maximum period threshold

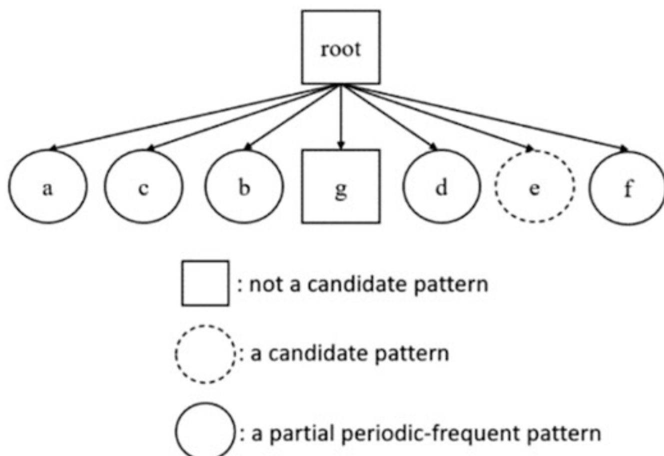


Fig. 2 The simple diagram of the *ITL-tree*

Fig. 3 The *ITL-list* and the *BP* table after scanning the second transaction (*abg*): (**a**) the *ITL-list* and (**b**) the *BP* table

ItemN	*a*	*c*	*b*	*g*
CountN	2	1	1	1
RangeC	2	1	1	1
FirstT	1	1	2	2
LastT	2	1	2	2

(a)

ItemN	T1	T2
a	T	T
c	T	
b		T
g		T

(b)

The *ITL-list* as shown in Fig. 3a (which is the result after processing the second transaction) contains *ItemN*, *CountN*, *RangeC*, *FirstT*, *LastT*, *Mark*, and *BP* for each pattern. It is used to record the useful information of each pattern. Take Table 2 as an example. Assume that we have the *MinSup* = 3, the *MinPR* = 75%, and the *MaxPer* = 2. First, we scan on the first transaction, "1: *ac*" with *ts* = 1. Since pattern *a* and pattern *c* are the first occurrence, we create their own *ITL-list* because the *ts* (timestamp) of the first occurrence is not larger than the *MaxPer* and the *RangeC* plus 1. Therefore, we modify their *CountN*, *RangeC*, *FirstT*, and *LastT* to 1, 1, 1, and 1, respectively. Since *FirstT* only records the *ts* of the first occurrence, even if it appears later, we do not need to deal with this variable. Then, we have to deal with the *BP* part as shown in Fig. 3b, where *BP* is a table composed of a series of items and Boolean values. For the above example, we know that pattern *a* and pattern *c* appear, when *ts* is 1. So in their *BP* table, we will set the Boolean value to *true* at T1.

Next, we scan the second transaction, "2: *abg*" with *ts* = 2. Since pattern *b* and pattern *g* are the first occurrence, we create their own *ITL-list*, because *ts* of the first occurrence is equal to the *MaxPer* and the *RangeC* plus 1. Therefore, we modify their *CountN*, *RangeC*, *FirstT*, and *LastT* to 1, 1, 2, and 2, respectively. Then, the

ItemN	CountN	RangeC	FirstT	LastT
a	8	9	1	12
c	8	8	1	13
b	7	8	2	12
g	2	1	2	8
d	6	6	3	12
e	5	3	3	13
f	4	4	8	13

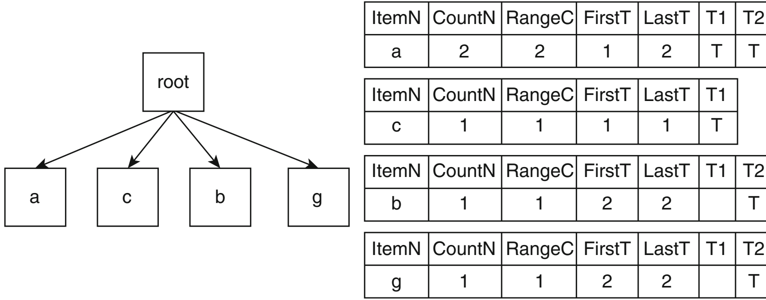
ItemN	T1	T2	T3	T4	T5	T6	T7	T8	T9	T10	T11	T12	T13	Mark
a	T	T		T		T	T		T		T	T		T
c	T			T		T		T		T	T	T	T	T
b		T		T		T	T		T		T	T		T
g		T						T						F
d			T	T		T		T		T		T		T
e			T				T			T	T		T	T
f								T		T	T		T	T

Fig. 4 The complete *ITL-list* of each pattern at level 1

CountN, *RangeC*, and *LastT* values of pattern *a* are updated to 2, 2, and 2, respectively. Because the current *ts* minus the *LastT* of pattern *a* is less than *MaxPer*, *RangeC* is increased by 1. At the same time, for pattern *a*, we set T2 in *BP* to *true*, and for patterns *b* and *g*, we do the same thing. The result is shown in Fig. 3a, b.

The same step is processed until the entire database is scanned, and the variables which have not been mentioned, Mark, will not be decided until the whole database has been scanned. For instance, through Table 2, we can know that pattern *a* last appears in ts 12. If the difference between the timestamp of the last transaction of the database and *LastT* of the pattern is less than or equal to *MaxPer*, we set Mark to True and increase *RangeC* by 1. Take pattern *a* as an example. The gap between 13 (ts of the last transaction) and 12 (*LastT* of "a") is less than *MaxPer*, so Mark of "a" is set to True and we increase its *RangeC* by 1. The complete *ITL-list* of each pattern at level 1 is shown in Fig. 4.

Let's start with a transactional database TDB3 to construct the *ITL-tree* as shown in Fig. 5 (which is the result after processing the second transaction) and the *ITL-list*. Basically, there are two cases which we must concern: (1) the item never appears before; (2) the item has appeared before. For the first case, the item never appears before; we add a new node at level 1 of the *ITL-tree* and give it a name. At the same time for this node, we create its *ITL-list*. For the second case, the item has appeared before, we need to find the node with this name from level 1 in the *ITL-tree*, and then update its *ITL-list*. When we scan the first transaction, we can know that there are



ItemN	CountN	RangeC	FirstT	LastT	T1	T2
a	2	2	1	2	T	T

ItemN	CountN	RangeC	FirstT	LastT	T1
c	1	1	1	1	T

ItemN	CountN	RangeC	FirstT	LastT	T1	T2
b	1	1	2	2		T

ItemN	CountN	RangeC	FirstT	LastT	T1	T2
g	1	1	2	2		T

Fig. 5 The *ITL-tree* and the *ITL-list* after scanning the second transaction

two patterns in the transaction, which are patterns *a* and *c*. When the algorithm reads the items in a transaction, it will change the *ITL-list* of these patterns in the *ITL-tree*. For instance, pattern *a* is the first pattern to be scanned, so we insert node *a* into level 1 of the *ITL-tree*. At the same time, we will create an *ITL-list* belonging to pattern *a* and start to modify the data inside. The *CountN* is set to 1, because pattern *a* appears for the first time. The *RangeC* is set to 1, because its period (i.e., 1–0 = 1, where 0 is the starting time) is less than *MaxPer*. The *FirstT* is set to 1, because the first occurrence is at *ts* 1. The *LastT* is set to 1, because the last occurrence is at *ts* 1. Next, we set the T1 of *BP* table to True. The remaining items in the first transaction is executed by the same way.

Next, we scan the second transaction. The second transaction has patterns *a*, *b*, and *g*. Since pattern *a* already exists at level 1 of the *ITL-tree*, we only need to update the data of its *ITL-list*. Take pattern *a* as an example. In the *ITL-list* of pattern *a*, its *CountN* is set to 2. Because it has appeared once before, we have 1 + 1 = 2. The *RangeC* is set to 2, because its period (i.e., 2–1 = 1, where the first 1 is *LastT*) is less than *MaxPer*. The *LastT* is set to 2, because the pattern *a* appears at *ts* 2 now. Then, for T2 of *BP* table, we set its *BP* table entry to True. Next, we turn to *b* and *g*. Take pattern *b* as an example. Since pattern *b* appears for the first time, we insert node *b* into level 1 of the *ITL-tree* and create its *ITL-list*. Then, we set its *CountN* to 1. The *RangeC* is set to 1, because its period (i.e., 2–0 = 2, where 0 is the starting time) is equal to *MaxPer*. The *FirstT* is set to 2, because the first occurrence is at *ts* 2. The *LastT* is set to 2, because the last occurrence is at *ts* 2. Moreover, we set the T2 of *BP* table to True. Since "b" does not appear at *ts* 1, T1 of *b* is empty. The condition of pattern *g* is the same as pattern *b*, so we do what has done for pattern *b* once for pattern *g*. As a result, the *ITL-tree* and the *ITL-list* after scanning the second transaction is shown in Fig. 5.

Finally, we perform the above process until the entire database has been scanned to complete level 1 of the *ITL-tree*, and the *ITL-list* are shown in Fig. 6. But we have not gotten the last period from the last *ts* (13). This last period will determine whether we have to increase *RangeC* by 1 and set *Mark* to *True*. At the same time, we can also know which of the patterns at level 1 are noncandidate patterns, candidate patterns, and partial periodic-frequent patterns.

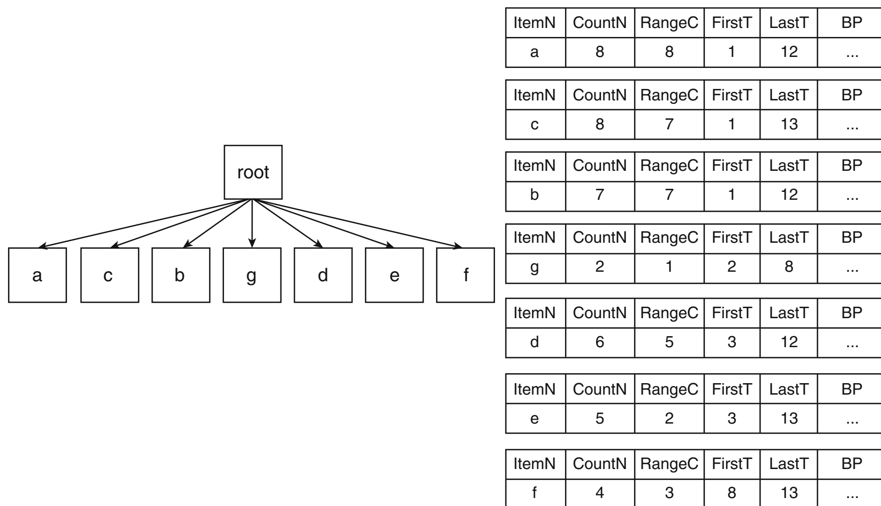
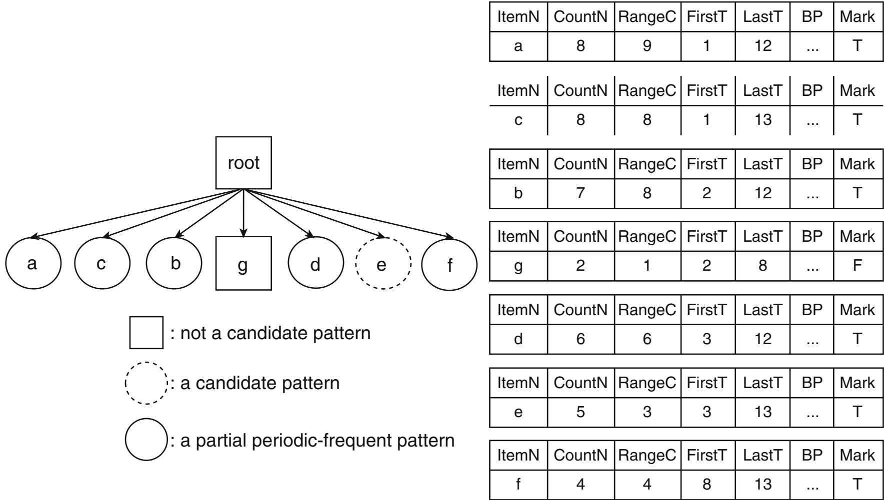


Fig. 6 The *ITL-tree* and the *ITL-list* after scanning the entire database

Therefore, the modified *ITL-tree* and the modified *ITL-list* are shown in Fig. 7. Through Fig. 7, according to the *ITL-list* of each node, we can know, whether this node can become the partial periodic-frequent pattern. As long as the number of *RangeC* in the *ITL-list* is not less than $MinPR \times (MinSup + 1)$ and *CountN* is not less than *MinSup*, we set it to the dotted circle. Furthermore, if *RangeC* is not less than $MinPR \times (CountN + 1)$, we set it to the solid circle, and this pattern is the partial periodic-frequent pattern which we need. The remaining square boxes are considered unnecessary, because its *CountN* is less than *MinSup* or its *RangeC* is less than $MinPR \times (MinSup + 1)$. So even if it is merged with other patterns, it cannot be a partial periodic-frequent pattern. Therefore, the partial periodic-frequent size 1 patterns are patterns *a*, *c*, *b*, *d*, and *f*.

Since we have found all partial periodic-frequent size 1 patterns, we can start finding partial periodic-frequent patterns of size 2. First, we will exploit these dotted or solid circle patterns in Fig. 7, where *RangeC* is larger than or equal to the $MinPR \times (MinSup + 1)$ and *CountN* is not less than *MinSup*. We merge two patterns on the same level of the *ITL-tree* with the same prefix into a (level + 1) super-set patterns. Note that the prefix can be null and we do not need to merge with the square box, because it is impossible to generate the partial periodic-frequent pattern. Since we want to merge the two patterns, we need to decide the order of the merged patterns. In the process of generating the size 2 patterns, we will merge the patterns at level 1 from left to right. Take Fig. 7 as an example. Patterns at level 1 of the *ITL-tree* have the order from left to right: [*a*, *c*, *b*, *d*, *e*, *f*]. Moreover, the pattern will generate a new pattern with each pattern after its order, and then it will be the next pattern. For instance, pattern *a* will generate the new size 2 pattern with pattern *c*, *b*, *d*, *e*, *f*, and then it will turn to pattern *c* and pattern *b*, *d*, *e*, *f* to generate the new size 2 pattern, and so on.



ItemN	CountN	RangeC	FirstT	LastT	BP	Mark
a	8	9	1	12	...	T

ItemN	CountN	RangeC	FirstT	LastT	BP	Mark
c	8	8	1	13	...	T

ItemN	CountN	RangeC	FirstT	LastT	BP	Mark
b	7	8	2	12	...	T

ItemN	CountN	RangeC	FirstT	LastT	BP	Mark
g	2	1	2	8	...	F

ItemN	CountN	RangeC	FirstT	LastT	BP	Mark
d	6	6	3	12	...	T

ItemN	CountN	RangeC	FirstT	LastT	BP	Mark
e	5	3	3	13	...	T

ItemN	CountN	RangeC	FirstT	LastT	BP	Mark
f	4	4	8	13	...	T

Fig. 7 The modified *ITL-tree* and the modified *ITL-list*

Begin (T8) End (T12)

	T1	T2	T3	T4	T5	T6	T7	T8	T9	T10	T11	T12	T13
a	T	T		T		T	T		T		T	T	
f								T		T	T		T

Fig. 8 The *BP* table for patterns *a* and *f*

Next, we illustrate how to generate size 2 patterns at level 2 of the *ITL-tree* and complete their *ITL-list*. Take pattern *a* and *f* as an example. At first, since we cannot find the pattern *af* connected to *a*, we create a node *af* and connect it to *a* with an edge. Then, we compare the *FirstT* of patterns *a* and *f* to find the larger value. This value means that when we check the *BP* table of two patterns, we only need to check from the Boolean value at this position. Because it will never appear at the same timestamp before. Similarly, we compare the *LastT* of the patterns *a* and *f* to find the smaller value. This value means that when we check the *BP* table of two patterns, we only need to check the Boolean value at this position from the front. Because after that, they will never appear at the same timestamp. Take Fig. 8 as an example. From this figure, we can know that pattern *f* does not appear before *ts* 8 and *a* does not appear after *ts* 12. Therefore, we only need to check the Boolean value between *ts* 8 and *ts* 12, which avoids unnecessary actions.

Then, we do the *AND* operation on *BPs* of patterns *a* and *f*. This *BP* table after the *AND* operation is regarded as *BP* table of pattern *af*. We will use this *BP* table to complete its *ITL-list*. At first, we need to find the timestamp with a *True* value

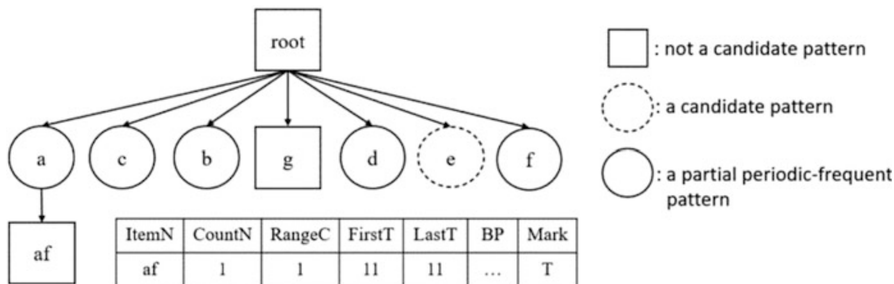


Fig. 9 The *ITL-tree* and the *ITL-list* after the pattern *af* is inserted

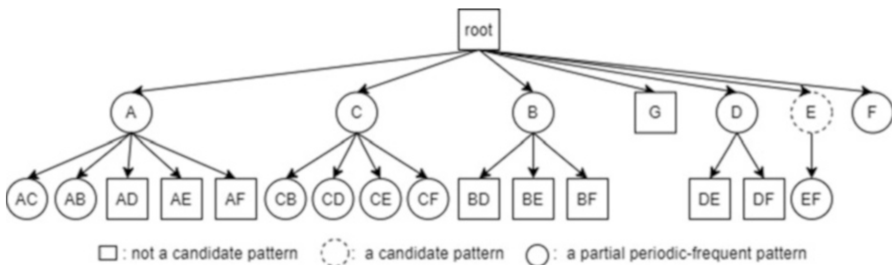


Fig. 10 The size 2 patterns of the *ITL-tree*

between *ts* 8 and *ts* 12. Then, we know that the first *ts* with *True* is at *ts* 11. Therefore, we set its *CountN*, *RangeC*, *FirstT*, and *LastT* to 1, 0, 11, and 11, respectively. Then, we find that the following timestamp has no *True*. Therefore, as long as we find the last period to modify the value of *RangeC* and *Mark*, we complete the *ITL-list* of *af*. At the same time, we can confirm whether pattern *af* is the noncandidate pattern, the candidate pattern, or the partial periodic-frequent pattern through the *ITL-list*, which is shown in Fig. 9. We apply the above steps to the remaining patterns. The result is shown in Fig. 10. The partial periodic-frequent size 2 patterns are patterns *ac*, *ab*, *cb*, *cd*, *ce*, cf, and *ef*.

After mining all the partial periodic-frequent size 2 patterns, we start to find the partial periodic-frequent size 3 patterns. Similarly, we will merge the two patterns with the same prefix at level 2 of the *ITL-tree* except for the noncandidate patterns. So based on the above sentence, we will insert the new size 3 patterns *acb*, *cbd*, *cbe*, *cbf*, *cde*, *cbf*, and *cef* into level 3 of the *ITL-tree* and follow the previous steps to complete their *ITL-list*.

Then, we confirm the type of each pattern according to their *ITL-list*. The result is shown in Fig. 11. According to this figure, we can know that patterns *acb* and *cef* are partial periodic-frequent patterns. Moreover, there is no size 4 pattern which can be generated. So all the partial periodic-frequent patterns in this database have been found.

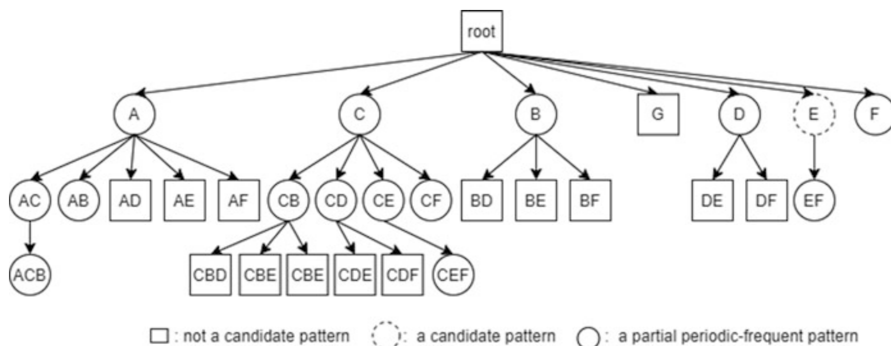


Fig. 11 The size 3 patterns of the *ITL-tree*

Finally, we compare the difference between our *Inverted-ITL* algorithm and the *GPF-growth* algorithm [12] for mining partial periodic-frequent pattern mining. We consider the original data in Table 2 as the input. When our algorithm constructs the data structures for the mining process, our algorithm only needs to scan database once and does not need the sorting step. However, the *GPF-growth* algorithm needs scanning database twice and sorts each transaction. Moreover, the *GPF-growth* algorithm needs to generate a prefix tree based on the reordered database, and it will generate many prefix trees during the mining process.

4 Performance

4.1 The Performance Model

We compare the processing time of the *Inverted-ITL* algorithm and the *GPF-growth* algorithm [12]. for mining partial periodic-frequent patterns in the real and synthetic datasets. We will consider using different values of the minimum support threshold, the maximum period threshold, and the minimum periodic-ratio threshold to execute different size datasets on the algorithm. For the real datasets, we use the Retail dataset (http://fimi.ua.ac.be/data/) for experiments. The details of the dataset Retail contains the transaction count = 88,162, the item count = 16,470, and the average item count per transaction = 10.30. For the synthetic dataset, T10.I4.D100K was generated by using the generator from the IBM Quest Dataset Generator. The parameter *T*, *I*, and *D* represents the average item count per transaction, the average maximal size of frequent itemsets, and the number of transactions in the dataset, respectively.

4.2 Experiments Results

First, let's deal with the Retail dataset. In Fig. 12, we show the comparison of the processing time for the Retail dataset and the synthetic dataset under the change of the maximum period threshold. In this experiment, we set the minimum support threshold = 0.01% and the minimum periodic-ratio threshold = 0.01%. Through Fig. 12a, we observe that both our algorithm and the *GPF-growth* algorithm maintain a fairly stable curve. The reason is that all items in the Retail dataset, except for the few specific items, the rest of the items will basically not repeat within the threshold, so this situation will happen. Moreover, according to this figure, it shows that we provide better performance than the *GPF-growth* algorithm which scans the dataset twice and sorts many times. From Fig. 12b, we observe that the performance of our algorithm is also better than that of the *GPF-growth* algorithm.

In Fig. 13, we show the comparison of the processing time for the Retail dataset and the synthetic dataset under the change of the minimum support threshold. In this experiment, we set the maximum period threshold = 10 and the minimum periodic-ratio threshold = 0.01%. Through Fig. 13a, we observe that as the value of minimum support threshold increases, the processing time of the two algorithms decreases. The reason is that only a small number of patterns can meet the threshold of support, when the minimum support threshold increases. When the number of the candidate patterns continues to decrease, the processing time also decreases. Through this figure, we observe that when the number of candidate patterns continues to decrease, the time curve of the *GPF-growth* algorithm will change greatly. However, our algorithm changes relatively small. This is because as long as the number of candidate patterns increases, the number of times that the *GPF-growth* algorithm needs to be sorted and the number of prefix trees generated will also increase. However, our algorithm only needs to process the *ITL-list* of each candidate pattern which are already in the *ITL-tree*. Therefore, in Fig. 13a, the processing time of our algorithm is faster than the *GPF-growth* algorithm. In Fig. 13b, the performance of our algorithm is also better than that of the *GPF-growth* algorithm.

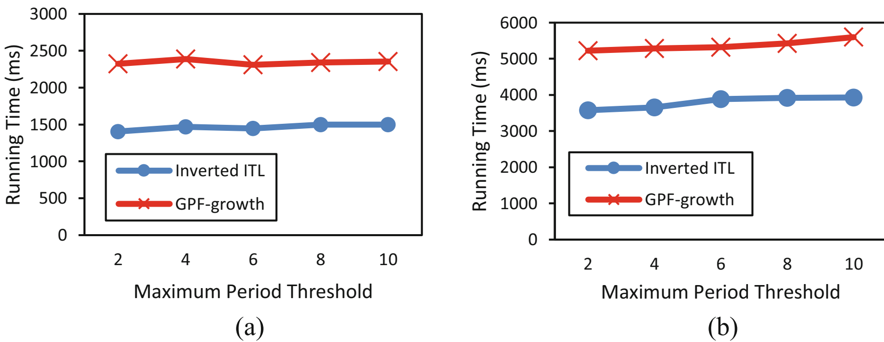


Fig. 12 A comparison of the processing time under the change of the maximum period threshold: (**a**) for the Retail dataset; (**b**) for the synthetic dataset

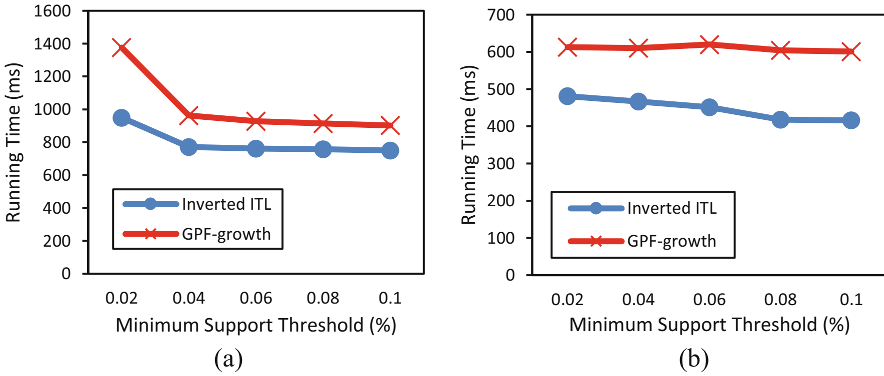


Fig. 13 A comparison of the processing time under the change of the minimum support threshold: (**a**) for the Retail dataset; (**b**) for the synthetic dataset

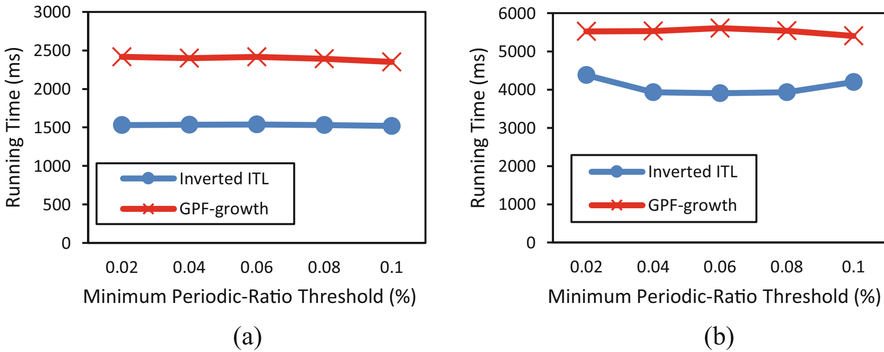


Fig. 14 A comparison of the processing time under the change of the minimum periodic-ratio: (**a**) for the Retail dataset; (**b**) for the synthetic dataset

In Fig. 14, we show the comparison of the processing time for the Retail dataset and the synthetic dataset under the change of the minimum periodic-ratio threshold. In this experiment, we set the maximum period threshold = 10 and the minimum support threshold = 0.01%. Through Fig. 14a, we can observe that the minimum periodic-ratio threshold has small effect on the performance of two algorithms. So our algorithm has better performance than the *GPF-growth* algorithm. In Fig. 14b, the performance of our algorithm is also better than that of the *GPF-growth* algorithm.

In Fig. 15, we show the comparison of the processing time for the Retail dataset and the synthetic dataset under the change of the data size. In this experiment, we set the maximum period threshold = 10, the minimum support threshold = 0.01%, and the minimum periodic-ratio threshold = 0.01%. Through Fig. 15a, we can observe

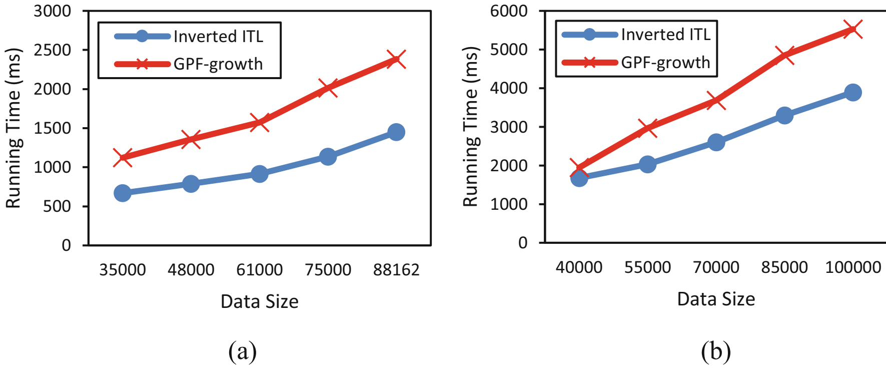


Fig. 15 A comparison of the processing time under the change of the change of the data size: (**a**) for the Retail dataset; (**b**) for the synthetic dataset

that as the size of the dataset increases, the processing time also increases. However, the growth rate of the *GPF-growth* algorithm is significantly larger than that of our algorithm. Because as the size of the dataset increases, the *GPF-growth* algorithm needs to sort the items in more transactions, but our algorithm only needs to deal with the *ITL-list* of each candidate pattern in the *ITL-tree*. Therefore, the *GPF-growth* algorithm needs a lot of time to deal with sorting and generating prefix trees, and we only need to deal with each pattern in the *ITL-tree*. Through Fig. 15b, the performance of our algorithm is also better than that of the *GPF-growth* algorithm.

5 Conclusion

In this paper, we have proposed an *Inverted-ITL* algorithm which can efficiently mine the partial periodic-frequent patterns. In the data mining, we have constructed the *ITL-tree* and the *ITL-list* which need less processing time to store information of each pattern than the *GPF-growth* algorithm for the same transactional database. From our simulation result, we have shown that our algorithm is more efficient than the *GPF-growth* algorithm.

Acknowledgments This research was supported in part by the National Science Council of Republic of China under Grant No. MOST 110-2221-E-110-054.

References

1. M.K. Gupta, P. Chandra, A comprehensive survey of data mining. Int. J. Inf. Technol. **12**, 1243–1257 (2020)
2. K. Kaithwas, P. Borkar, A review on different data mining algorithms and selection methods, in *Proc. of the 2019 Int. Conf. on Intelligent Sustainable Systems*, (2019), pp. 511–515
3. R. Agrawal, R. Agrawal, R. Srikant, Fast algorithms for mining association rules in large databases. Proc. 20th Int. Conf. Very Large Data Bases, 487–499 (1994)
4. Z.-H. Deng, DiffNodesets: An efficient structure for fast mining frequent itemsets. Appl. Soft Comput. **41**, 214–223 (2016)
5. J. Han, J. Pei, Y. Yin, Mining frequent patterns without candidate generation. Proc. 2000 ACM SIGMOD Int. Conf. Manag Data **29**(2), 1–12 (2000)
6. H. Bui, B. Vo, H. Nguyen, T.-A. Nguyen-Hoang, T.-P. Hong, A weighted N-list-based method for mining frequent weighted itemsets. Expert Syst. Appl. **96**, 388–405 (2018)
7. T. Le, B. Vo, An N-list-based algorithm for mining frequent closed patterns. Expert Syst. Appl. **42**(19), 6648–6657 (2015)
8. B. Vo, S. Pham, T. Le, Z.-H. Deng, A novel approach for mining maximal frequent patterns. Expert Syst. Appl. **73**, 178–186 (2017)
9. R.U. Kiran, M. Kitsuregawa, Novel techniques to reduce search space in periodic-frequent pattern mining. Proc. 19th Int. Conf. Database Syst. Adv. Appl., 377–391 (2014)
10. R.U. Kiran, M. Kitsuregawa, P.K. Reddy, Efficient discovery of periodic-frequent patterns in very large database. J. Syst. Software **112**, 110–121 (2016)
11. R.U. Kiran, P.K. Reddy, Towards efficient mining of periodic-frequent patterns in transactional databases. Proc. 21th Int. Conf. Database Exp Syst Appl, 194–208 (2010)
12. R.U. Kiran, J.N. Venkatesh, M. Toyoda, M. Kitsuregawa, P.K. Reddy, Discovering partial periodic-frequent patterns in a transactional database. J. Syst. Software **125**, 170–182 (2017)
13. A. Surana, R.U. Kiran, P.K. Reddy, An efficient approach to mine periodic- frequent patterns in transactional databases. Proc. 15th Pacific-Asia Conf. Knowl. Discov. Data Mining, 254–266 (2011)
14. S.-S. Chen, T.C.-K. Huang, Z.-M. Lin, New and efficient knowledge discovery of partial periodic patterns with multiple minimum supports. J. Syst. Software **84**(10), 1638–1651 (2011)