Mining Weighted Periodic Patterns by a Weighted Direction Graph Based Approach for Time-Series Databases

Ye-In Chang[1*], Cheng-An Fu [1], Jia-Zhen Que [1]

[1] Department of Computer Science and Engineering, National Sun Yat-Sen University, Kaohsiung, Taiwan Republic of China.

* Corresponding author. Tel.: 886-7-5252000 (ext. 4334); email: changyi@mail.cse.nsysu.edu.tw

Manuscript submitted May 15, 2021; accepted July 10, 2021
doi: 10.17706/jsw.16.6.267-284

Abstract: Periodic pattern mining in time series database plays an important part in data mining. However, most existing algorithms consider only the count of each item, but do not consider about the value of each item. To consider the value of each item on periodic pattern mining in time series databases, Chanda *et al*. proposed an algorithm called *WPPM*. In their algorithm, they construct the suffix trie to store the candidate pattern at first. However, the suffix trie would use too much storage space. In order to decrease the processing time for constructing the data structure, in this paper, we propose two data structures to store the candidates. The first data structure is *Weighted Paired Matrix*. After scanning the database, we will transform the database into the matrix type, and it is used for the second data structures. Therefore, our algorithm not only can decrease the usage of the memory space, but also the processing time. Because we do not need to use so much time to construct so many nodes and edges. Moreover, we also consider the case of incremental mining for the increase of the data length. From the performance study, we show that our proposed algorithm based on the *Weighted Direction Graph* is more efficient than the *WPPM* algorithm.

Key words: Flexible patterns, periodic Patterns, time series database, weighted direction graph, weighted patterns.

1. Introduction

In data mining, there are many techniques to find meaningful information in the database, such as mining association rules [1], frequent pattern mining [2], [3], utility pattern mining and periodic pattern mining [4]-[12]. Among these research topics, the periodic pattern mining is commonly used on the time-series database. Therefore, the researches for the time-series databases are mostly accompany with periodic pattern mining [4]-[7], [11], [12]. However, in the real situation, the value of each item may not be the same. Therefore, some researches consider the periodic pattern accompanied with the weighted value [4].

In the periodic pattern mining, we not only need to see the frequency of the event, but also need to make sure that the frequent pattern occurs in the similar period length [12]. For instance, we assume that the period length is 5, and the minimum threshold is 75%. There is a time-series pattern *T1* in Table 1. The event *"ab"* frequently appears 4 times over 20/5 = 4 periods, where 20 is the total length of the event sequence. However, it is not a periodic pattern. Because it does not appear averagely. On the other hand, the event *"ab"* of the time-series pattern *T2* in Table 1 is a periodic pattern, since it does not only appear frequently, but it also appears averagely in each period.

Table 1. An Example of the Event Sequences

Time-Series Database	Event Sequence
T1	*ababc daccd daccb abcab*
T2	*abcde abeec dabce abdcd*

The reason for why the periodic pattern mining is suitable for time-series database is as follows. The time event in the database has its timeliness [12]. For example, an event appears frequently in some time, but it appears frequently again after a long period of time. Such an event is not certainly valuable for the time-series database. On the other hand, an event appears regularly in the database, and that event perhaps valuable in the database.

The periodic pattern could be classified into three types in time-series database: symbol periodic patterns, partial periodic patterns and full cycle periodic patterns [11]. In the partial periodic patterns, the pattern consists of a sequence, and the sequence may appear periodically [11]. For example, a time-series database *T3* is shown in Fig. 1. The sequence pattern *"ab"* occurs periodically with the period value which is equal to 4. The length of the sequence is smaller than or equal to the period value. Moreover, the distance of two successive patterns should not be longer than the half length of the time-series sequence. For example, given a time series database *T* = *"abcd efgh abmn"*, although the count of pattern *"ab"* is 2, the distance of the patterns is 7, which is longer than the half length of the time-series sequence 12/2 = 6, where 12 is the total length of *T*. Hence, the pattern *"ab"* is not the periodic pattern.

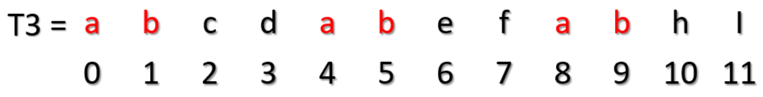


Fig. 1. A time-series database *T3.*

Moreover, in some condition, the pattern sometimes has the unwanted event, so some researches [4], [5], 11] use the *"*"* to represent it by the don't care event while finding the flexible patterns. For example, given a string *T= "abcd adcb accd"*, although the pattern *"abc"*, *"adc"*, *"acc"* would not be the periodic pattern, the pattern *"a * c"* would be the periodic pattern. Because the count of *"a * c"* is 3. Hence, it is another difficulty, when we consider about the partial periodic patterns.

Moreover, the most common situation considers the string along with the weight of each item. For instance, for a string *T* = *"bcda"*, according to the weight of each item shown in Table 2, the weight of string *T* is 0.675. Since the researches [4], [5] define the weight of a string is the average weight of each item, the weight of string *T* is (0.6+0.9+0.5+0.7)/4=0.675.

For another example, there is a string *T* = *"abdcbdcc"*, with the weight of each item shown in Table 2, and we suppose the minimum support threshold, σ=1.6. The mining result of string *T* would be *"b"* with the total weight 0.9*2 = 1.8 and *"bd"* with the total weight (0.9+0.7)/2*2 = 1.6. Although the count of pattern *"c"* is 3, the weight support is 0.5 * 3 = 1.5 < 1.6. Therefore, the pattern *"c"* would not be the answer.

This is why the weighted pattern mining is difficult, because we should consider the count along with the weight of each pattern. Furthermore, the pruning phase of the weighted pattern should be more careful. For example, we consider a pattern *T1= "a"* and a pattern *T2= "ab",* i.e, *T1* $\subset$ *T2*. If the count of pattern *T2* is larger than the minimal count *σ*, then the count of pattern *T1* is also larger than σ absolutely. However, if we consider the weight of each item, the above property would not be established. For instance, there is a string *T= "abdc bdcc abcc"*, and the weight of each item is shown in Table 2. Moreover, we suppose the minimum support threshold, *σ=1.4*. The total weight of the pattern *"ab"* is ((0.6+0.9)/2) *2 = 1.5, but the total weight of pattern *"a"* is 0.6 * 2 = 1.2. In fact, the pattern *"a"* $\subset$ *"ab"* and pattern *"ab"* is periodic pattern;

however, pattern *"a"* is not periodic pattern. Hence, this is another difficulty of weighted periodic pattern mining.

In the real life condition, the content of a database is commonly increased. Hence, the incremental mining is important. If we do not consider about the incremental mining, the processing time of the data mining would be so long. For example, there is a database with the length of 1000, and we do the data mining on this database. At this time, the database increases new data with the length of 500. If we do not consider about the incremental mining, we need to reload the whole database with the length of 1500. On the other hand, if we consider about the incremental mining, we only need to load the new data with the length of 500.

Table 2. Weight of Each Item

Item	Weight
a	0.6
b	0.9
c	0.5
d	0.7

For the concern of the related strategies, Rasheed *et al.* proposed a suffix tree based algorithm [12]. They use the suffix tree to mine the periodic patterns. However, Chanda *et al.* [5] found that the algorithm proposed by Rasheed *et al*. [12] has some limitations. For example, they cannot mine the pattern with the unimportant event *"*"*. So Nishi *et al.* proposed an algorithm [5] named *FPPM*, which is based on the suffix trie. The *FPPM* algorithm can mine the pattern with the unimportant event *"*"*. However, Chanda *et al.* [4] consider that the *FPPM* algorithm [5] still has some disadvantage, since the *FPPM* [5] algorithm cannot mine the pattern with weight of each item. It means that the *FPPM* algorithm [5] considers only the count of each item. Hence, Chanda *et al.* [4] modify the mining process part of the *FPPM* algorithm [5], they consider that the weight of each item should be put in the mining process. For example, the count of item *A* is 3, and the weight of item *A* is 0.8. Moreover, the threshold is 2. Later, they use the multiplication of the weight and the count of item *A* to judge whether item *A* is a periodic pattern or not. In this example, the multiplication of the weight and the count of item *A* is 0.8 * 3 = 2.4, which is larger than the threshold 2, so the item *A* is the periodic pattern. Hence, Chanda *et al.* proposed an algorithm named *WPPM* [4].

The *WPPM* algorithm [4] can actually handle more conditions than the *FPPM* algorithm [5] does, since the *WPPM* algorithm [4] considers about the weight of each item. Moreover, the *WPPM* algorithm [4] is more efficient than *FPPM* algorithm [5] from their experiments. However, we consider that the suffix trie part of *WPPM* algorithm [4] wastes too much memory space, which also affects the processing time. For an input string *T = "abccabdcacdcabdc"*, the suffix trie of *T* will have at least 92 nodes and 91 edges, and this will be the waste of memory use. To reduce the main memory space and the processing time, in this paper, we propose the Weighted Time Position Join algorithm based on the Weighted Direction Graph to store the candidate patterns. By using the Weighted Direction Graph, there will be only 4 nodes and 8 edges in Weighted Direction Graph for the same input string *T*, where 4 is the number of the unique symbols and 8 is the number of different pairs of continuous symbols. Later, we use Weighted Direction Graph to mine the period pattern of the input string *T*. Therefore, our Weighted Direction Graph not only can reduce the usage of the memory space, but also the processing time. Note that we do not need to use so much time to construct so many nodes and edges. Moreover, we consider about the incremental mining on the weighted periodic pattern mining based on our Weighted Direction Graph. From our experimental results, we show that our approach is more efficient than the *WPPM* algorithm [4].

The rest of the paper is organized as follows. In Section 2, we give a survey of some algorithms for mining periodic patterns in time-series databases. In Section 3, we present the proposed Weighted Time Position Join algorithm. In Section 4, we present the performance study of our algorithm and make a comparison between our algorithm and the *WPPM* algorithm [4]. Finally, Section 5 gives the conclusion.

2. A Survey of Algorithms for Mining Periodic Patterns in Time Series Databases

In this section, we will introduce how to mine periodic patterns in time series databases by the following algorithms, including the *FPPM* algorithm [5], and the *WPPM* algorithm [4].

Chanda *et al.* [5] proposed an efficient approach *FPPM* algorithm to mine flexible periodic patterns based on the suffix trie structure. This approach is capable of generating periodic patterns by skipping intermediate unimportant events and detecting all tree types of periodicity in a single run only. For the mining process, in the initial step, given a time series string, they use this string to construct a suffix trie. Later, through the pruning phase, they prune the suffix trie, and this trie will be a period value *P* depth trie. Finally, they use the pruned trie to run the *FPPM* algorithm to mine the periodicity. In the mining process of the *FPPM* algorithm. As the confidence satisfies the minimum confidence threshold, the regarded pattern is a valid pattern, and the valid patterns will be moved to the next depth to been joined in the next depth.

In the *FPPM* algorithm, it does not consider about the weight of any event, so the *WPPM* algorithm will consider the weight of each event to see the different result by the *FPPM* algorithm [4]. In the mining process, it is mostly the same as the *FPPM* algorithm, the difference is that they consider about the weight of each event. They define the weight of each event which is a real number between 0 and 1. Later, they construct the suffix trie and the pruned suffix trie by *T*. The suffix trie and the pruned suffix trie are the same as the *FPPM* algorithm. The difference is that if the WPPM algorithm wants to determine whether the pattern is valid, it need to check whether the weight support is larger or smaller than the minimum support threshold, σ.

3. The Weighted-Time Position Join Algorithm

The existing algorithms [4], [5] mostly use so much time and memory space to mine the periodic patterns, and they use the periodicity detection algorithm proposed by Rasheed *et al.* [12] to find the candidates of the periodic patterns. However, the periodicity detection algorithm uses three levels of loops to find the candidates. Moreover, the suffix trie part of the *WPPM* algorithm [4] will use so much memory space to store the nodes and edges of the trie. To reduce the time and the space of the mining process, we propose the weighted-time position join algorithm to mine the periodic patterns efficiently.

3.1. Data Structure

Here, we present the weighted time-position join algorithm, which is revised from [9]. We apply it to improve the performance of the suffix trie part of *WPPM* algorithm [4]. We consider that the suffix trie will use so much memory space, which will cause much unnecessary waste of memory space. For example, given a string *T= "abccabdcacdcabdc"*, which is the string presented in [4], with $|T|=16$, if we use string *T* to construct the suffix trie, it will have at least nighty two nodes and lots of edges. However, for the same input, by using our weighted time-position join algorithm, there will be only four nodes and eight edges. The weighted time-position algorithm contains two steps. First, we construct the weighted paired matrix, and use the weighted paired matrix to construct the weighted direction graph. Second, we traverse the weighted direction graph to find the periodic patterns. We will introduce the two steps of the weighted time-position join algorithm later, and Table 3 shows the variables which we will use in the algorithm. In the real world, each item has its own weight as the profit or price. We construct the table to store the weight of each item as shown in Table 4.

Table 3. Variables

Variable	Meaning
T	The time series string
$W(P)$	The weight value of pattern P
$Support(P)$	The support value of pattern P
θ	The maximum event skipping threshold
σ	The minimum support threshold
$MaxW$	The maximum weight of a given database

Table 4. Weighted Table

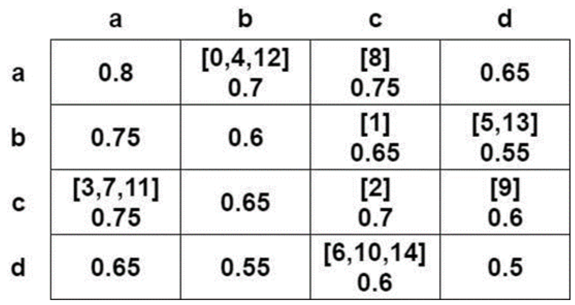
Item	Weight
a	0.8
b	0.6
c	0.7
d	0.5

The weight paired matrix *WPM* is a matrix which we use it to store the adjacent symbols. Let sequence *S=< s0, s1, s2, ..., sn>* with length *n*, and a two dimensional *m × m* weighted paired matrix *WPM*, where *m* is the count of unique symbols in *S.* For every two continues symbol *st*, *st+1* of sequence *T*, we store the occurrence position *t* of *st, st+1* into *WPM [st, st+1]*. That is, for the sequence *S*, we will get the substring *< s0, s1 >*, and then we store the occurrence position 0 of the substring into *WPM [s0, s1]*. Next, we get the substring *< s1, s2 >*, and then we store the occurrence position 1 of the substring into *WPM [s1, s2]*, and so on. On the other hand, the decimal number in *WPM [s1, s2]* is the average weight of the item *s1* and the item *s2*. Meanwhile, *WPM [si, si]*, like *WPM [s1, s1]*, is the weight of item *si*.

For example, there is a sequence *T* shown in Fig. 2, where the number under the sequence is the position of each symbol. Note that the string is proposed in [4]. Moreover, we use sequence *T* to construct the paired matrix, which is shown in Fig. 3. As we can see, the occurrence position of each continues symbol is stored in the matrix. That is, the pattern *"ab"* occurs in positions [0, 4, 12], so the positions are stored in *WPM [a, b]*. In the same way, the pattern *"ca"* occurs in positions [3, 7, 11], so the positions are stored in *WPM [c, a]*. On the other hand, the decimal number in *WPM [a, b]* is the average weight of the pattern *ab*. According to Table 4, the weight of the item *a* is 0.8, and the weight of item *b* is 0.6, so the average weight is (0.8+0.6)/2=0.75, and it is stored in *WPM [a, b].*

```
a b c c   a b d c   a c d  c    a  b  d  c
0 1 2 3   4 5 6 7   8 9 10 11   12 13 14 15
```

Fig. 2. The example string T.



	a	b	c	d
a	0.8	[0,4,12] 0.7	[8] 0.75	0.65
b	0.75	0.6	[1] 0.65	[5,13] 0.55
c	[3,7,11] 0.75	0.65	[2] 0.7	[9] 0.6
d	0.65	0.55	[6,10,14] 0.6	0.5

Fig. 3. The example of weighted paired matrix.

After all the positions are stored in the matrix *WPM*, we use matrix *WPM* to construct the weighted direction graph *WDG*. We will traverse matrix *WPM* to construct the graph. That is, if matrix *WPM [a, b]* is blank, we will ignore it. On the other hand, if node *a* or node *b* does not exist in the graph, then we will construct the node which stores the item name and the corresponding weight of node *a* or node *b*. Meanwhile, we will construct the edge which points from node *a* to node *b*, and the edge will store the occurrence position of the pattern *ab*. For example, from Fig. 3, suppose we are at matrix *WPM [a, b],* and there is nothing in the weighted direction graph. Since node *a* and node *b* are not in the graph, we construct node *a* and node *b* and store the corresponding weight first. After this, matrix *WPM [a, b]* stores [0, 4, 12], which means that pattern *ab* appears at positions 0, 4 and 12, so we construct an edge from node *a* to node *b*, and store the occurrence position in the edge. For another example, when we traverse to matrix *WPM [a, c]*, and node *a* is already in the graph, we just need to construct node *c* and store the weight of item *c* in the node. After that, we construct the edge points from node *a* to node *c* and store the occurrence position into the edge, and so on.

There will be *m* nodes in the graph, where *m* is the number of unique symbols in string *S*. Each node will store each unique symbol and its corresponding weight *W*. Moreover, there will be *n* edges, where *n* is the count of *WPM [t1, t2]* which has the occurrence positions in matrix *WPM*. Therefore, in Fig. 3, we have *n = 8*. The edge stores the occurrence position of the connected two symbols. Fig. 4 shows the example of the direct graph constructed by Fig. 3. As we can see, the edge between node *a* and node *b* stores [0, 4, 12], it means that the node *a* and the node *b* are continues symbols, and their occurrence positions are [0, 4, 12]. For another special example, there is an edge that starting from *c*, and it points to the same node *c*, it means that there is a pattern *cc* and it appears at position 2. Totally, there are four nodes and eight edges in Fig. 4. Moreover, the value besides each node means the weight of the symbol.

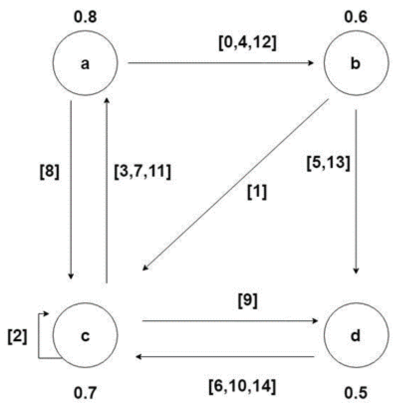
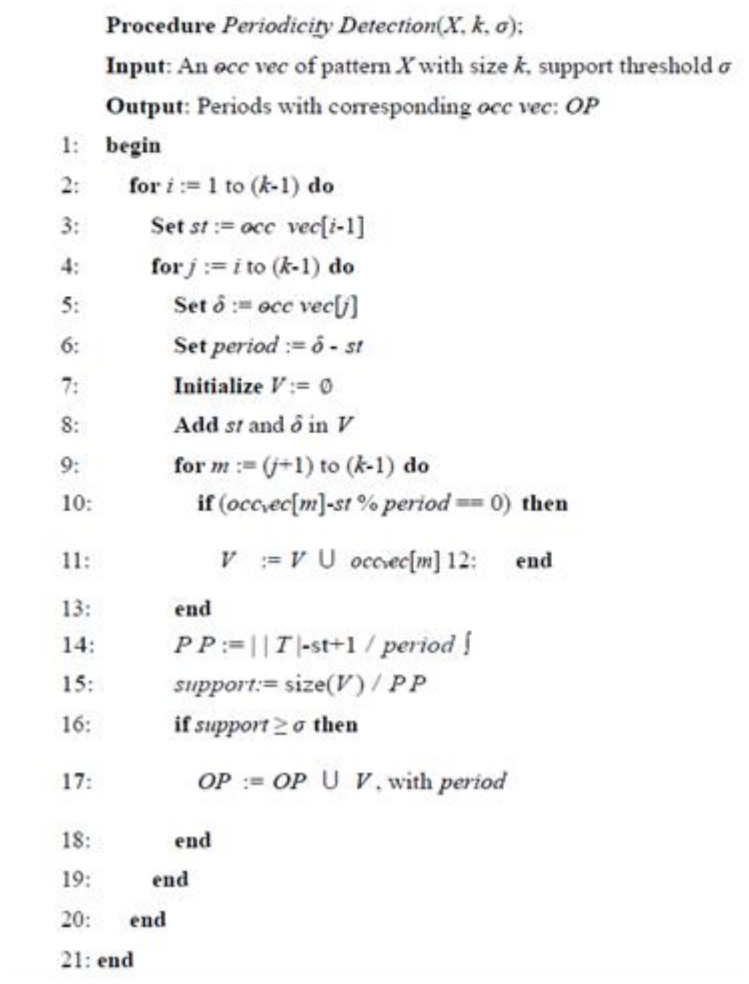


Fig. 4. The example of weighted direction graph.

3.2. The Mining Process

Here, we will introduce the mining process. In order to discover periodic patterns in the time-series database, Chanda *et al.* proposed an algorithm [5] which can find all possible valid periods with corresponding occurrence vectors.

The input of the algorithm is an occurrence vector of pattern *X* with size *k*, and support threshold *σ*. The result of the algorithm will be the periods with the corresponding occurrence vector. For example, we suppose that an occurrence vector *Occ_vec = "2, 3, 7, 9, 11, 15"* is the input, and support threshold *σ* = 1. The result of the input will be as follows. For the period value *p = 4*, one of the corresponding occurrence vectors will be *"3, 7, 11, 15"*. Moreover, for *p = 4*, another occurrence vector = *"1, 5, 13".* Note that the difference between two adjacent occurrences could be the multiple of the period value *p*. Hence, *"1, 5, 13"* is also the corresponding occurrence for p = 4. Moreover, for the period value p = 6, the corresponding occurrence vector will be *"3, 9, 15"*. The pseudo code of existing periodicity detection algorithm [4] is shown in Fig. 5 The first for loop is finding the first parameter *st* which we want to use later. The second for loop will find the second parameter δ and period = δ - *st*. Meanwhile, *st* and δ will be added into a vector *V*. The third for loop will find the third parameter, and if the *st* subtracting from the third parameter will be divisible by period value *p*, the third parameter will be added into vector *V*. After the third for loop, there will have a formula to judge whether the vector *V* is the final answer or not.



```
Procedure Periodicity Detection(X, k, σ);
Input: An occ vec of pattern X with size k, support threshold σ
Output: Periods with corresponding occ vec: OP
begin
  for i := 1 to (k-1) do
    Set st := occ vec[i-1]
    for j := i to (k-1) do
      Set δ := occ vec[j]
      Set period := δ - st
      Initialize V := ∅
      Add st and δ in V
      for m := (j+1) to (k-1) do
        if (occ vec[m]-st % period == 0) then
          V := V ∪ occvec[m] 12: end
      end
      P P := ⌊ |T|-st+1 / period ⌋
      support:= size(V) / P P
      if support ≥ σ then
        OP := OP ∪ V, with period
      end
    end
  end
end
```

Fig. 5. Existing periodicity detection algorithm [4].

After the construction of *WDG* graph, we can use it to do the mining process for the given period *p*. Given a time series string *T = "abccabdcacdcabdc"* for example, the weight of each item is shown in Table 4. Minimum threshold $\sigma = 2$, and maximum skipping threshold $\theta = 2$, and period value $p = 4$ are used in this example to explain the mining process of this algorithm.

From the string *T*, the weighted paired matrix is constructed, which is shown in Fig. 3. The weighted direction graph is constructed by the weighted paired matrix, which is shown in Fig. 4.

Basically, for each different symbol, we will perform the following mining process. Let's use symbol *a* as an example to explain the details of the mining process. The mining process of depth 1 is shown in Fig. 6. There is only one symbol *a* in depth 1, so we just need to check whether the weighted support is larger than the minimum threshold σ. Note that the occurrence vector for each different symbol is recorded in an array *OC* which will be used in determining whether such a symbol is a periodic pattern, i.e., for the following mining process of depth 1. The occurrence vector of symbol *a* is *"0, 4, 8, 12"*, as recorded in array *OC* shown in Table 5. Note that the results of procedure Periodicity Detection are stored in an array *OL*, which is shown in Table 6. The support of symbol *a* is 4, and the weight of symbol *a* is 0.8 according to Table 4. Therefore, the weighted support is 0.8*4 = 3.2, and it is larger than the minimum threshold σ = 2, so it is considered as the valid weighted periodic pattern.

Table 5. The Array *OC*

Pattern	Occurrence vector
a	[0, 4, 8, 12]
b	[1, 5, 13]
c	[2, 3, 7, 9, 11, 15]
d	[6, 10, 14]

Table 6. The Array *OL*

Pattern	(Period, Occurrence vector)
OL[*a*]	{(4, [0, 4, 8, 12])}
OL[*b*]	{(4, [1, 5, 13])}
OL[*c*]	{(4, [3, 7, 11, 15]), (6, [3, 9, 15])}
OL[*d*]	{(4, [6, 10, 14])}

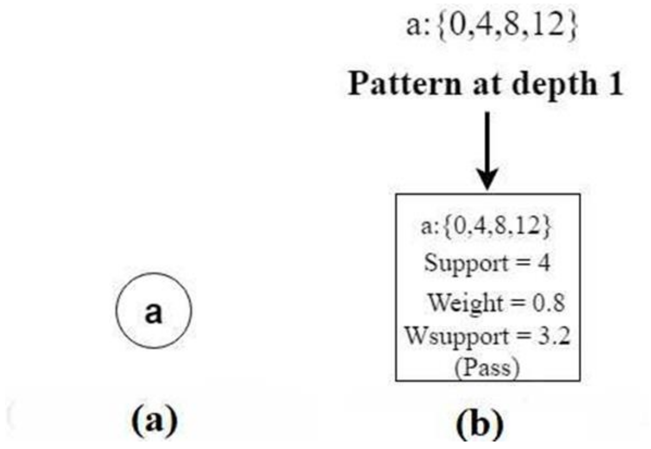
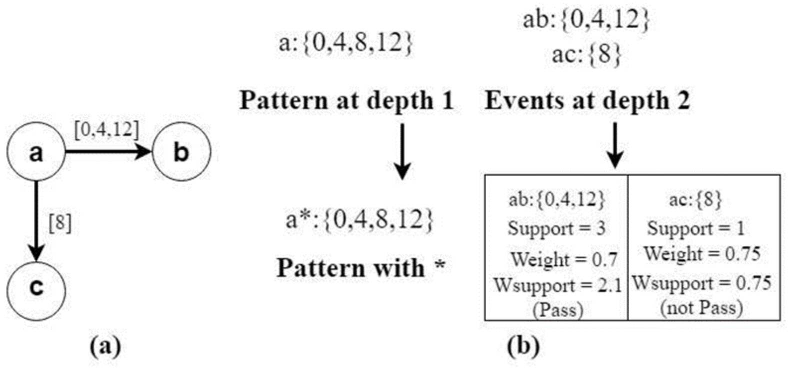


Fig. 6. The mining process of depth 1: (a) The traversed. *WDG* graph of depth 1; (b) The periodic pattern of depth 1.

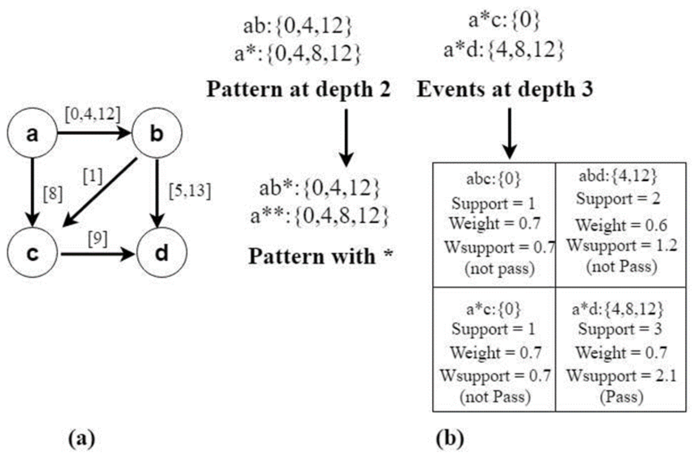
Next, the mining process moves to depth 2, and the mining process of depth 2 is shown in Fig. 7. First, we need to check where node *a* points to. In this example, the node *a* points to node *b* and node *c*. Hence, we store the occurrence vector of node *b* related to node *a* = *"0, 4, 12"* (i.e., *count* = 3) and node *c* related to node *a = "8"* (i.e., *count = 1*). Next, the support of pattern *"ab"* is 3, and the weight of pattern *"ab"* is the average of item *a* and item *b*. Hence, the average weight of pattern *ab* is (0.8+0.6)/2=0.7, and the weighted support of pattern *ab* is 3 * 0.7=2.1, which is larger than the minimum threshold σ = 2, so pattern ab is the weighted periodic pattern. However, in the same way, the weighted support of pattern *ac* is 0.75, which is lower than the minimum threshold σ = 2. Therefore, the pattern *ac* is not the weighted periodic pattern. On the other hand, there is a table storing the pattern *"a *"* (i.e., the starting symbol *a* followed with any symbol), because we want to mine the pattern which can skip the unimportant events represented by symbol *"*"*. The occurrence vector of pattern *"a *"* is the same as the occurrence vector of symbol *a*, i.e., *"0, 4, 8, 12"*. The pattern ending with "*" would not be mined in this depth, but it would help to generate the pattern in the future.



a:{0,4,8,12}
ab:{0,4,12}
ac:{8}
Pattern at depth 1
Events at depth 2
[0,4,12]
[8]
a*:{0,4,8,12}
Pattern with *
ab:{0,4,12} Support = 3 Weight = 0.7 Wsupport = 2.1 (Pass)
ac:{8} Support = 1 Weight = 0.75 Wsupport = 0.75 (not Pass)
(a)
(b)

Fig. 7. The mining process of depth 2: (a) The traversed. *WDG* graph of depth 2; (b) The periodic pattern of depth 2.

For the pattern of depth ≥ 3, we need to use the time-position join. The meaning of the time-position join is the connection between pattern *XP* (**X1, X2, ..., Xw**) and pattern *YP* (*Y1, Y2, ..., Yt*). First, we have the condition *Xw* = *Y1*. Second, the time position of symbol *Xw* is the same as the time position of symbol *Y1*. Then, the resulting pattern of the time-position join between pattern *XP* and pattern *YP* is (*X1, X2, ..., Xw, Y2, ..., Yt*). (Note that the occurrence vector of the result after the time-position join is the subset of the occurrence vector of pattern *XP*.) Later, the mining process moves to depth 3, the mining process of depth 3 is shown in Fig. 8.



ab:{0,4,12}
a*:{0,4,8,12}
a*c:{0}
a*d:{4,8,12}
Pattern at depth 2
Events at depth 3
[0,4,12]
[8]
[1]
[5,13]
[9]
ab*:{0,4,12}
a**:{0,4,8,12}
Pattern with *
abc:{0} Support = 1 Weight = 0.7 Wsupport = 0.7 (not pass)
abd:{4,12} Support = 2 Weight = 0.6 Wsupport = 1.2 (not Pass)
a*c:{0} Support = 1 Weight = 0.7 Wsupport = 0.7 (not Pass)
a*d:{4,8,12} Support = 3 Weight = 0.7 Wsupport = 2.1 (Pass)
(a)
(b)

Fig. 8. The mining process of depth 3: (a) The traversed. *WDG* graph of depth 3; (b) The periodic pattern of depth 3.

At depth 3, we first need to see the occurrence vector of the edges. For example, the occurrence vector on the edge between node *a* and node *b* is [0, 4, 12]. It means that pattern *"ab"* is at positions 0, 4 and 12. On the other hand, the occurrence vector on the edge between node *b* and node *c* is [1], it means that pattern *"bc"* is at position 1. Therefore, we can combine pattern *"ab"* at the position 0 with pattern *"bc"* at the position 1, it will become pattern *"abc"* starting at position 0. That is, we perform the time position join operation between pattern *"ab"* and pattern "*bc"*. For considering the flexible pattern *"a *c"*, we record the starting position of symbol *a*, i.e., 0 as the starting position of the flexible pattern *"a *c"*. Since the number of the occurrence vector of edge *bc* is 1, we can stop the following connecting process between edge *ab* and edge *bc*. Moreover, for position 0 in the occurrence vector of edge *ab*, we also mark it as used. In the same way, pattern *"abd"* can be constructed (i.e., by using the time position join operation) with pattern *ab* at the positions 4 and 12, and pattern *bd* at the position 5 and 13, respectively. Note that for this case of trying to connect edge *ab* (with the occurrence vector [0, 4, 12]) and edge *bc* (with the occurrence vector [5, 13]), we may have to consider all 6 possible cases of tests. However, in fact, we only have to consider the unchecked position *x* of the occurrence vector of edge *ab* with the unchecked position (*x*+1) of the occurrence vector of edge *bc*. Therefore, for the case of the connection of edge *ab* and edge *bc*, we only have to do two tests, instead of four tests of unmarked positions of edge *ab* (with two unmarked positions in its occurrence vector) and edge *bc* (with two unmarked positions in its occurrence vector). Similarly, for concerning the flexible pattern *"a * d"*, we include the concern of pattern *"abd"* (i.e., the previous pattern *"ab"* shown in level 2) and pattern *"acd"* (i.e., the previous pattern *"ac"* shown in level 2). Note that in such a concern, we care about the possible joinable pattern. (Note that pattern *ac* is an unpassed pattern in depth 2, but it still needs to be considered in depth 3 under the *"*"* condition.) In general, if the occurrence gap of the nearly two continues edges with the same direction is 1, it means that we can generate the new pattern with length 3. After we have found all the events at depth 3, we can generate the candidate patterns like these patterns at depth 2. For the concern of the candidate pattern of the following level, i.t., level 4, we record the related positions of pattern *"ab *"* and pattern *"a * *"*, which are the same as the related positions of pattern *"ab"* and pattern *"a *"*, respectively. Note that the passed pattern *"a * d"* will also be concerned in the following level 4. (Note that for the average weight of a pattern containing one or more number of *"*"* symbols, we let it be the average weight of all symbols. In our example, the average weight of all of symbols (*'a', 'b', 'c', 'd'*) is (0.8+0.6+0.7+0.5)/4 = |0.65| ≈ 0.7.) Finally, we calculate the weighted support of patterns which we generated previously and check whether the weighted support is larger than the minimum threshold σ or not.

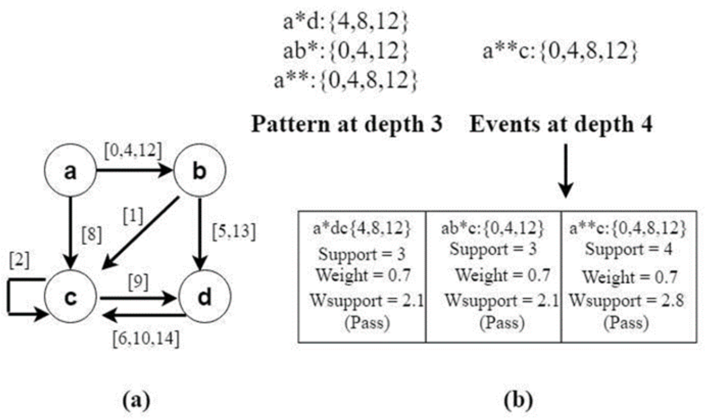

a*d:{4,8,12}
ab*:{0,4,12}
a**:{0,4,8,12}
a**c:{0,4,8,12}
Pattern at depth 3
Events at depth 4
a
b
c
d
[0,4,12]
[8]
[1]
[5,13]
[2]
[9]
[6,10,14]
a*dc{4,8,12} Support = 3 Weight = 0.7 Wsupport = 2.1 (Pass)
ab*c:{0,4,12} Support = 3 Weight = 0.7 Wsupport = 2.1 (Pass)
a**c:{0,4,8,12} Support = 4 Weight = 0.7 Wsupport = 2.8 (Pass)
(a)
(b)

Fig. 9. The mining process of depth 4: (a) The traversed. *WDG* graph of depth 4; (b) The periodic pattern of depth 4.

Finally, the mining process moves to depth 4, the mining process of depth 4 is shown in Fig. 9. The mining process at depth 4 is the same as the mining process at depth 3. For pattern *"a * d"* with positions *"4, 8, 12"*, we concern the possible joinable positions of edge *dc*. That is, whether positions 4 + 2 = 6, 8 + 2 = 10, 12 + 2 = 14 exist in edge *dc*. Those tests are all satisfied, so the support of pattern *"a * dc"* is 3. Next, for pattern *"ab *"* with positions *"0, 4, 12"*, we concern the possible joinable patterns of edges *bdc* and *bcc*. To achieve this goal, we must first check whether patterns *"bdc"* and *"bcc"* exist. Therefore, we check whether edge *bd* and edge *dc* are joinable. Moreover, we check whether edge *bc* and edge *cc* are joinable. The result of two tests are all joinable, due to 5 + 1 = 6, 13 + 1 = 14 and 1 + 1 = 2. Since the joinable positions of pattern *"ab"* are *"0, 4, 12"*, the resulting joinable positions of pattern *"ab *c"* are *"0, 4, 12"*. For pattern *"a * *"*, when we concern the following possible joinable pattern through connected directional edge, the possible pattern of length 4 is *"a * *c"* with joinable positions *"0, 4, 8, 12"*. Those possible joinable patterns include pattern *"abdc"* (with starting positions 4 and 12), pattern *"abcc"* (with starting position 0) and pattern *"acdc"* (with starting position 8). That is, the number of the joinable patterns is 4. Finally, we calculate the weighted support of the generated patterns. The result of flexible periodic patterns with the starting symbol *a* is shown in Table 7. Moreover, the result of flexible periodic patterns with the starting symbol *b* and symbol *c* are shown in Table 8. (Note that the weighted support of the strings starting with symbol *d* are smaller than the minimum threshold σ, so there is no result starting with symbol *d*.)

Table 7. The Result of Flexible Patterns with the Starting Symbol *a*

Pattern	Occurrence vector
a	[0, 4, 8, 12]
ab	[0, 4, 12]
$a * d$	[4, 8, 12]
$a * *c$	[0, 4, 8, 12]
$ab * c$	[0, 4, 12]
$a * dc$	[4, 8, 12]

Table 8. The Result of Flexible Patterns with the Starting Symbol *b* and Symbol *c*

Pattern	Occurrence vector
$b * c$	[1, 5, 13]
c	[3, 7, 11, 15]
ca	[3, 7, 11]
$ca * d$	[3, 7, 11]
$c * *d$	[3, 7, 11]

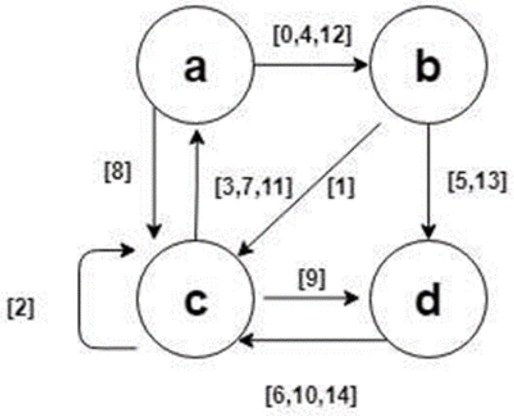


Fig. 10. The *WDG* of the input string *S.*

After the mining process, sometimes the databases will add new data into the databases. At this time, if we do not consider the incremental way to handle such condition, we need to reload all the database and reconstruct the whole data structure. Hence, we consider the incremental way of our weighted direction graph.

Suppose there is a string *S = "abccabdcacdcabdc"*, Fig. 10 shows the *WDG* graph of the string *S*. Now we add a new string *S2 = "bde"* behind the string *S*, and we do not need to reload the whole string *S*. However, we need the last symbol of string *S,* because the symbol *"c"* of string *S* and the symbol *"b"* of string *S2* should be connected. Hence, string *S3 "cbde"* will be the new input.

First of all, we use string *S3* to construct a new *WPM*. Fig. 11 shows the constructed *WPM.* Note that the reason for there are *"15, 16, 17"*, rather than *"0, 1, 2"*. Because the string *S3* is actually connected behind the string *S*. Hence, the occurrences should be added (*|S| - 1*).

	a	b	c	d	e
a					
b				[16]	
c		[15]			
d					[17]
e					

Fig. 11. The *WPM* of the input string *S*3.

Later, we use the new *WPM* to update the original *WDG* graph. Note that there are three cases, when we update the *WDG* graph. For Case 1, if the node exists and the edge also exists, then we just need to add the new occurrence on the edge. For Case 2, if the node exists but the edge does not exist, we need to construct a new edge and add the occurrence. For Case 3, if the node does not exist, we need to construct a new node and a new edge. For example, when we read *"cb"* on the *WPM* , we need to construct a new edge pointing from node *c* to node *b* and add the occurrence vector [15] on the edge. The reason is that there is no edge pointing from node *c* to node *b*, which is Case 2. Fig. 12 shows the updating process.

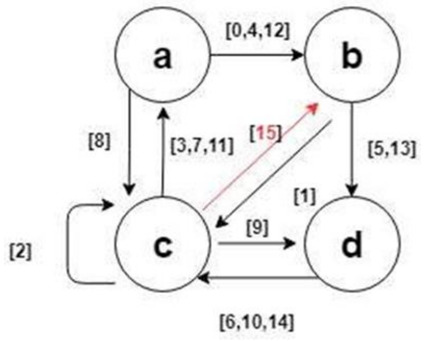


Fig. 12. The updating process of the *WDG* for case 2.

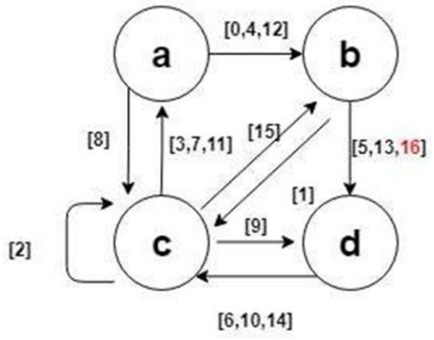


Fig. 13. The updating process of the *WDG* for case 1.

Later, when we read *"bd"* on the *WPM*, we just need to add new occurrence on the edge. The reason is that there is an edge pointing from node *b* to node *d*, which is Case 1. Fig. 13 shows the updating process.

Finally, when we read *"de"* on the *WPM*, we construct a new node *e* and an edge pointing from node *d* to node *e*. Moreover, we add the occurrence vector [16] on the edge. The reason is that there is no node *e*, which is Case 3. Fig. 14 shows the updating process. After the update of the *WDG* graph, we can use this *WDG* graph to run the mining process.

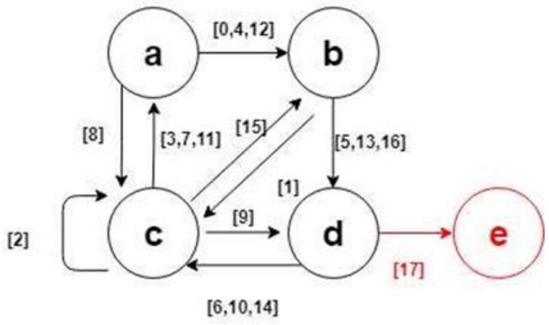


Fig. 14. The updating process of the *WDG* for case 3.

3.3. Comparison

In this section, we discuss the difference between our proposed *WDG* graph and the *WPPM* algorithm [4]. First, the number of nodes and the number of edges are different. Given a string *S*, the number of the unique symbols for string *S* is *C*, where *C ≤ |S|*. The number of nodes in our *WDG* graph is definitely *C*. The number of edges in our *WDG* graph is mostly *C*(C-1)*. Since in the worst condition, each node could be connected with other nodes, and our *WDG* graph has directionality, we do not need to divided by two. On the other hand, the number of nodes in the suffix trie is the addition of the length of all the substrings in the string *S* in the worst case. Hence, it is equal to *(|S| + (|S|-1) + ... + 1) = (|S| + 1) *|S|/2+1*, where the last plus 1 is the root node. The number of the edges in the suffix trie is *(|S| + 1) *|S|/2+1-1*, since it is a tree structure. (Note that the number of edges is the number of total nodes minus 1.)
For example, given string *S = "abccabdcacdcabdc"*, the number of unique symbol *C = 4*. The number of the nodes in our *WDG* graph is 4. The number of the edges in our *WDG* graph is 8, which is less than 4*(4-1) =12 needed in the worst case. On the other hand, the number of the nodes in the suffix trie is 92. The number of the edges in the suffix trie is 91, since it is a tree structure.

Second, our *WDG* graph considers about the incremental mining. The number of the reloaded items is different. Since we consider about the incremental mining, we only need to reload the last item in the original database to do the connecting operation. However, the *WPPM* algorithm should read the whole database with the original data and the new data.

4. Performance

In this section, we present the performance study of our weighted-time position join algorithm and the *WPPM* algorithm [4] on time-series data mining. We use the synthetic dataset as data source and the real dataset from the dataset repository https://www.investing.com/commodities/brent-oil-historical-data

4.1. The Performance Model

In the performance method, we will compare the processing time of these two algorithms. The synthetic dataset which we have used is *T3400.C5*, where the parameters *T* and *C* mean the length of the input string, and the number of the unique characters of the input string, respectively. (Note that the length of the synthetic string with five different symbols is 3400. Moreover, the weight value of each unique item will be generated randomly between 0 and 1, and the weighted table *WT* will be listed from these weight values.)

In the real datasets, we will use the Brent Oil Company Data Set [4] to do the experiment. There are 1530 tuples and the number of the unique characters is 7 in the dataset. Through these experiments, we want to see how the length of datasets affects the processing time of these two algorithms. Moreover, the number of unique characters should be the same as the number of the nodes of our algorithm.

4.2. Experiments Results

First, we compare the processing time under the change of the length of the input and the change of the period value *p* for the synthetic datasets. Second, we compare the processing time under the change of the length of the input and the change of the period value *p* for the real dataset. Third, we will compare the processing time on the incremental dataset. The correctness of our *WDG* algorithm is verified by checking whether the number of the result patterns of our *WDG* algorithm is the same as that of the *WPPM* algorithm.

In Fig. 15, we show the comparison of the processing time for the synthetic dataset T3400.C5 on the change of the input length. In this experiment, we set the period value *p = 8*, and *threshold = 2.* (Note that we create 10 different strings of length 3400 randomly. Then, for each string, we record the processing time of the concerned algorithm for the first 1500, 2000, 2500, 3000 and 3400 symbols, respectively. Finally, for the concerned algorithm, we compute the average of the processing time of the 10 cases of different strings with the same length.) From Fig. 15, we show that the processing time of our *WDG* algorithm is faster than the *WPPM* algorithm. Moreover, the increase on the processing time of the *WPPM* algorithm is much more than that of our *WDG* algorithm. The reason is that the processing time of constructing the data structure of our *WDG* algorithm is faster than that of the WPPM algorithm. For example, given a string *T = "abccabdcacdcabdc"*, with *|T| = 16*, if we use *T* to construct the suffix trie, it will have at least nighty two nodes and lots of edges. However, for the same input, by using our weighted time-position join algorithm, there will be only four nodes and eight edges.

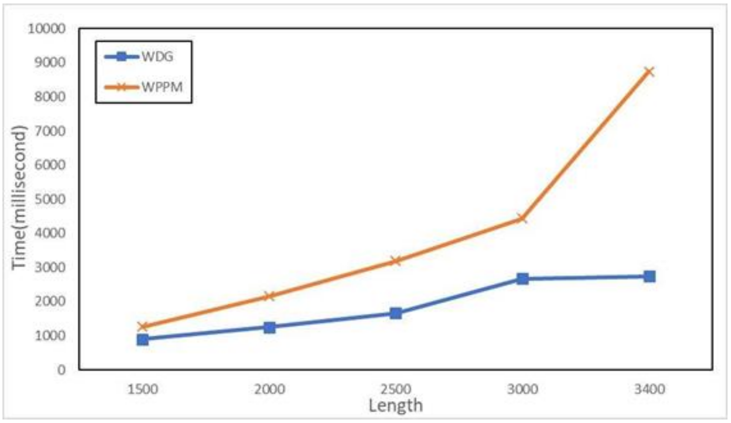


Fig. 15. A comparison of the processing time for the synthetic dataset. T3400.C5 (with period = 8) under the change of the length

In Fig. 16, we show the comparison of the processing time (i.e., the average of the processing time of testing 10 different strings with the same length) for the synthetic datasets T3400.C5 on the change of the period value *p*. In this experiment, we set the input length = 2000 (i.e., the first 2000 symbols), and *threshold* = 2. From Fig. 16. we show that the processing time of our *WDG* algorithm is faster than that of the *WPPM* algorithm. Moreover, the increase on the processing time of the *WPPM* algorithm is the same as our *WDG* algorithm. The reason is that the processing time of constructing the data structure of our *WDG* algorithm is faster than that of the *WPPM* algorithm. Although the value of p increases, the times of join operations in our *WDG* algorithm increases. However, in our *WDG* algorithm, the reducing time on the construction of the data structure is more than the increasing time on the joining property.

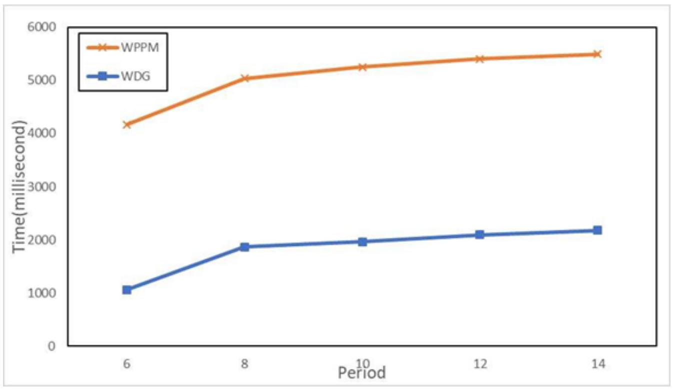


Fig. 16. A comparison of the processing time for the synthetic dataset T3400.C5 under the change of the period.

In Fig. 17, we show the comparison of the processing time for the Oil datasets on the change of the input length. In this experiment, we set the period value p = 8, and *threshold* = 2. From Fig. 17, we show that the processing time of our *WDG* algorithm is faster than that of the *WPPM* algorithm. Moreover, the increase on the processing time of the *WPPM* algorithm is much more than that of our *WDG* algorithm. The reason is the same as what we have mentioned above.

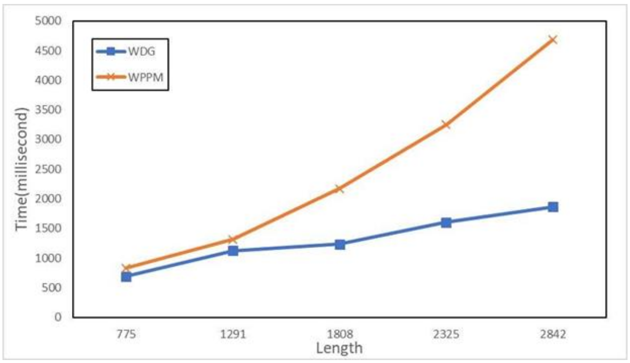


Fig. 17. A comparison of the processing time for the Oil dataset.
(with period = 8) under the change of the length

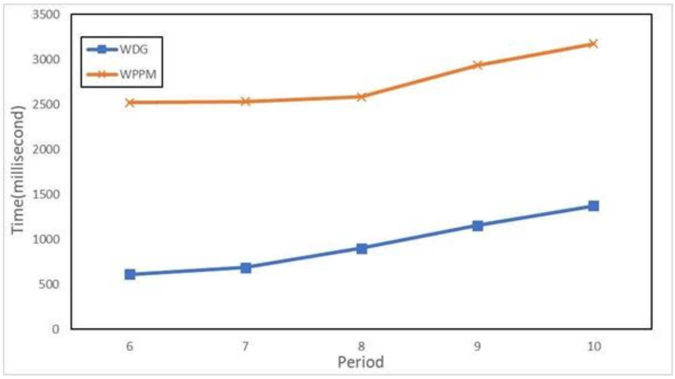


Fig. 18. A comparison of the processing time for the Oil dataset under the change of the period.

In Fig. 18, we show the comparison of the processing time for the Oil datasets on the change of the period value *p*. In this experiment, we set the input length = 2000, and *threshold* = 2. From Fig. 18. we show that the processing time of our *WDG* algorithm is faster than that of the *WPPM* algorithm. Moreover, the increase on the processing time of the *WPPM* algorithm is the same as that of our *WDG* algorithm. The reason is the same as we have mentioned above

Here, we show the processing time on incremental mining for the above two databases, including the synthetic dataset T3400.C5, and the Oil dataset. In this experiment, we set the period value *p* = 8, and *threshold* = 2. However, they [4] do not mention about the incremental way on the *WPPM* algorithm. Hence, the way of the testing on the incremental way for the *WPPM* algorithm is that we add the processing time from the previous mining and the processing time after the string size is increased. For example, after we get the processing time on the length of 1000, the database adds 300 items behind the original database, and we want to mine the database on the length of 1300. Since the *WPPM* algorithm does not have the incremental way, we need to mine the whole database on the length of 1300. Hence, the processing time on the incremental way of the *WPPM* algorithm is the addition of the processing time on the database length of 1000 and the processing time on the database length of 1300. (Note that since the WPPM algorithm should be ready for any possible value of the period *p*, the suffix trie is always needed to be constructed in the first step, Therefore, no matter the new added symbol already exists or does not exist in the current suffix trie, each path of the suffix trie will be traversed again for the new added symbol.)

In Fig. 19, Fig. 20, we show the comparison of the processing time for the incremental synthetic dataset T3400.C5, and the incremental Oil dataset respectively. From Fig. 19, Fig. 20, we show that our incremental way is much more efficient than the non-incremental way on the *WPPM* algorithm.

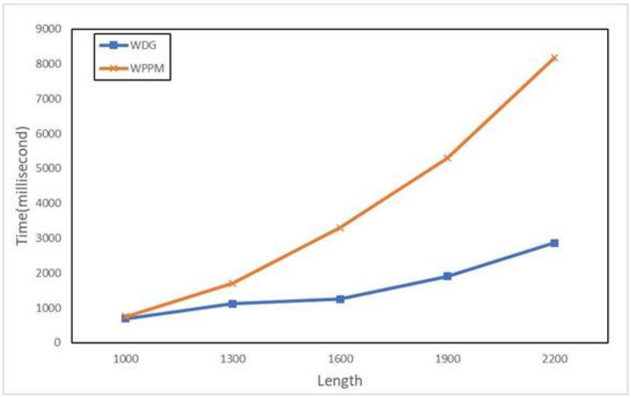


Fig. 19. A comparison of the processing time for the incremental. synthetic dataset T3400.C5 (the initial length is 1000)

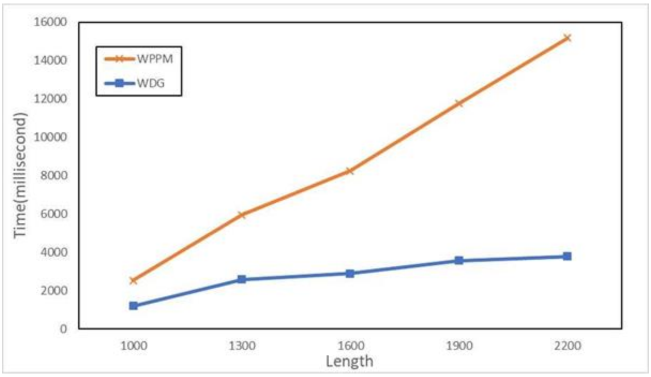


Fig. 20. A comparison of the processing time for the incremental Oil dataset (the initial length is 1000).

5. Conclusion

In this paper, we have proposed a weighted time position join algorithm that can efficiently mine the weighted periodic patterns. In the data preprocessing, we have constructed the weighted direction graph, which uses fewer number of nodes and edges than the *WPPM* algorithm for the same input. Moreover, we have considered the incremental mining for our algorithm. From the performance study, we have shown that our algorithm is more efficient than the WPPM algorithm. In the future, we plan to do the periodic pattern mining on time series databases for the case of the data decrease from the databases.

Conflict of Interest

The authors declare no conflict of interest.

Author Contributions

Chang and Fu conducted the research; Fu and Qiu analyzed the data and wrote the paper; all authors had approved the final version.

Acknowledgment

This research was supported in part by the Ministry of Science and Technology of Republic of China under Grant No. MOST 109-2221-E-110-045.

References

[1] Agrawal, R., & Srikant, R. (1994). Fast algorithms for mining association rules in large databases. *Proceedings of the 20th Int. Conf. on Very Large Data Bases* (pp. 478–499).

[2] Ahmed, C. F., Tanbeer, S. K., Jeong, B.-S., Lee, Y.-K., & Choi, H.-J. (2012). Single-pass incremental and interactive mining for weighted frequent patterns. *Expert Systems with Applications*, *39(9)*, 7976–7994, 2012.

[3] Lee, G., Yun, U., & Ryu, K. H. (2014). Sliding window based weighted maximal frequent pattern mining over datastreams. *Expert Systems with Applications*, *41*, 694-708.

[4] Chanda, A. K., Ahmed, C. F., Samiullah, M., & Leung, C. K. (2017). A new framework for mining weighted periodic patterns in time series databases. *Expert Systems with Applications*, *79*, 207–224.

[5] Chanda, A. K., Saha, S., Nishi, M. A., Samiullah, M., & Ahmed, C. F. (2015). An efficient approach to mine flexible periodic patterns in time series databases. *Engineering Application of Artificial Intelligence*, *44*, 46–63.

[6] Elfeky, M. G., Aref, W. G., & Elmagarmid, A. K. (2005). Periodicity detection in time series databases. *IEEE Trans. on Knowledge and Data Engineering*, *17(7)*, 875–887.

[7] Faraz, R., & Reda, A. (2010). STNR: A suffix tree based noise resilient algorithm for periodicity detection in time series databases. *Applied Intelligence*, *32(3)*, 267–278.

[8] Fournier-Viger, P., Li, Z., Lin, C.-W., Kiran, R. U., & Fujita, H. (2019). Efficient algorithms to identify periodic patterns in multiple sequences. *Information Sciences*, *489*, 205-226.

[9] Li, C. E., & Chang, Y. I. (2016). A time-position join method for periodicity mining in time series databases. *Proceedings of the Int. Computer Symposium*, 294–299.

[10] Ma, S., & Hellerstein, J. L. (2001). Mining partially periodic event patterns with unknown periods. *Proceedings of the 17th Int. Conf. on Data Engineering* (pp. 205–214).

[11] Nishi, M. A., Ahmed, C. F., Samiullah, M., & Jeong, B.-S. (2013). Effective periodic pattern mining in time series databases. *Expert Systems with Applications*, *40*, 3015–3027.

[12] Rasheed, F., Alshalalfa, M., & Alhajj, R. (2011). Efficient periodicity mining in time series databases

using suffix trees. *IEEE Trans. on Knowledge and Data Engineering*, *23(1)*, 79–94.

Ye-In Chang received her B.S. degree in computer science and information engineering from National Taiwan University, Taipei, Taiwan, in 1986. She received her M.S. and Ph.D. degrees in computer and information science from The Ohio State University, Columbus, Ohio, in 1987 and 1991, respectively. From August 1991 to July 1999, she joined the Faculty of Department of Applied Mathematics at National Sun Yat-Sen University, Kaohsiung, Taiwan. From August 1997, she has been a professor in Department of Applied Mathematics at National Sun Yat-Sen University, Kaohsiung, Taiwan. Since August 1999, she has been a professor in Department of Computer Science and Engineering at National Sun Yat-Sen University, Kaohsiung, Taiwan. Her research interests include database systems, distributed systems, multimedia information systems, mobile information systems and data mining.

C. A. Fu received the M.S. degrees in computer science and engineering from National Sun Yat-Sen University in 2019. His research interests include data mining and distributed computing. He is currently a system designer in Taiwan.

J. Z. Qiu received the B.S. degree from Yuan Ze University in 2019. He is currently a M.S. student in the Department of Computer Science and Engineering at National Sun Yat-Sen University in Taiwan. His research interests include data mining and distributed computing.