Sliding Window Based Weighted Periodic Pattern Mining over Time Series Data

Redwan Ahmed Rizvee[1], Md Shahadat Hossain Shahin[1], Chowdhury Farhan Ahmed[1], Carson K. Leung[2][0000−0002−7541−9127](✉), Deyu Deng[2], and Jiaxing Jason Mai[2]

[1] University of Dhaka, Dhaka, Bangladesh
farhan@du.ac.bd
[2] University of Manitoba, Winnipeg, MB, Canada
kleung@cs.umanitoba.ca

Abstract. Sliding windows have been crucial in mining time series. Many existing studies focus on reconstruction of the underlying structure (e.g., suffix tree) for each new window. However, when the window size is large or when the window slides frequently, reconstruction may perform poorly. In this paper, we propose a solution that dynamically updates the structure (rather than reconstruction for each modification or sliding). Moreover, many existing studies rely on the weight of maximum weighted item in the database to avoid testing unnecessary patterns when mining weighted periodic patterns from time series, but it may still require lots of weight checking to determine whether a pattern is a candidate. In this paper, we also propose an additional solution to address this problem by discarding unimportant patterns beforehand so as to speed up the candidate generation process. Evaluation results on real-life datasets show the effectiveness of our two solutions in handling sliding window and pruning redundant candidate patterns.

Keywords: Time series · Weighted periodic pattern mining · Dynamic database · Sliding window · Pruning.

1 Introduction

Discovering an efficient approach for mining frequent patterns has always been an important issue in knowledge discovery [5, 10, 12, 13, 16]. The idea of generating patterns has evolved over time and flooded a huge set of new domains. Sequential pattern mining [9, 14, 17, 19] is one of the popular areas in the field of pattern mining, and *time series pattern mining* [6, 7] is a very renowned and widely discussed topic in sequential pattern mining. The core input of time series pattern mining is data stream (e.g., a stream of sequence of events or items found with respect to time interval). A popular structure to represent time series is a *suffix tree* [18], from which *frequent patterns* can be mined on different thresholds and conditions. Data streams are *continuous, unbounded*, and *not necessarily uniformly distributed* [3, 11]. This creates the challenge of *dynamicity*, which is

also the core of *sliding window problem* [1] in numerous real-life applications (e.g., weather forecast, natural disasters prediction, etc.) [21]. To solve this sliding window problem, many existing studies rely on reconstructing the data structure to represent a modified window. However, this approach can be very expensive, especially in case of large window size or frequent sliding of windows. In this paper, we propose a *dynamic tree based solution to handle sliding window in time series* (*DTSW*), which focuses on (i) updating the data structure dynamically, (ii) maintaining (rather than reconstructing) a dynamic tree for each modified window, and (iii) keeping the tree suitable for various kinds of pattern mining.

In addition to DTSW, our second contribution centers around mining *weighted periodical patterns* [6] from time series. The introduction of weight to patterns helps in mining more interesting patterns when compared with its unweighted counterpart [1]. Weighted periodical patterns in time series are weighted sequences that periodically occur at least a certain amount of times along with a weight satisfying the user-specified threshold. Weighted pattern mining can be very useful in time series to discover interesting features. For example, if analyzing the transactions of a sports kit shop, the sold products may vary with many parameters (e.g., time, event, etc.). Selling rate of football jersey increases after every four years when the World Cup hits. A main challenge in weighted pattern mining is how to avoid testing *undesired candidates* so as to speed up the candidate generation process. Note that the *downward closure property* (*DCP*) cannot be applied directly in weighted versions of pattern mining. To speed up candidate generation, many existing works use the *Max Weight* concept, but they need to test a large number of unnecessary patterns for candidacy which in turn degrades performance. Hence, our second contribution in this paper is an efficient pruning solution—called *maximum possible weighted support* (*MPWS*) *pruning*—to significantly reduce the number of patterns to be tested for candidacy. To recap, *our key contributions* of this paper is our following two solutions:

- DTSW, a dynamic tree based solution to handle sliding window in time series (Section 3).
- MPWS pruning, an efficient solution to speed up the candidate generation process in weighted periodic pattern mining (Section 4).

The remainder of this paper is organized as follows. The next section gives background and related works. Sections 3 and 4 present our two proposed solutions. Section 5 shows evaluation results. Finally, conclusions are drawn in Section 6.

2 Background and Related Works

2.1 Sliding Window Problem

Discretization is a technique to represent a group of data with a single symbol. As time series is basically information gathered with respect to time interval, it can be represented as a string or sequence of characters from a given set by

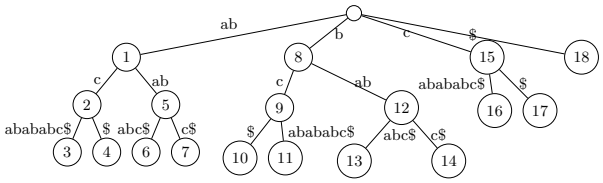


Fig. 1: Explicit suffix tree for string "abcababababc$"

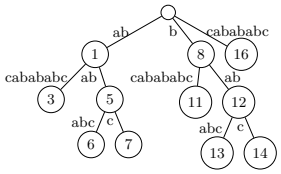


Fig. 2: Implicit suffix tree for string "abcababababc".

discretizing the values. For example "abcababababc$" is a discretized time series sequence.

As the suffix tree has been shown [18] to be an efficient data structure to represent time series and for frequent periodical pattern mining, we use it as our data structure. We use *Ukkonen's algorithm* [20], which a fast linear-time algorithm to construct suffix tree. To elaborate, a suffix tree represents all the suffixes of a string. If all the suffixes can be found by traversing from root to leaf nodes, then the suffix tree is in its *explicit* form. On the other hand, if all the suffixes do not end in leaves but rather embedded in the paths, then the tree is in its *implicit* form. Fig. 1 shows an explicit suffix tree of string "abcababababc$", and Fig. 2 shows an implicit suffix tree of string "abcababababc". An important concept in the Ukkonen's algorithm for constructing a suffix tree is *suffix link*, which helps to traverse the tree efficiently. According to Ukkonen's proposal, each and every internal node of the tree points to another internal node (or the root) as its suffix link. Suffix link of a node A with path "$\alpha\beta$" from root where (i) 'α' is exactly one symbol and (ii) 'β' can contain zero or more symbols will point to another (internal or root) node B as its suffix link if and only if node B has the path 'β' from the root. For example, in Fig. 1, Node 1 points to Node 8 as its suffix link. Each pass of Ukkonen's algorithm for adding symbol starts from an *active point*, which represent the position of largest implicit suffix in the tree at the current moment. The active point consists of (i) an *active node* representing the node position from which new pass will start), (ii) an *active edge* providing the information about the edge from active node where suffixes overlapping is being occurred, and (iii) an *active length* representing the number of symbols been overlapped in the direction of active edge from active node. Ukkonen proposed *rule extensions* in his algorithm. For instance, to add a new symbol to the end of all the existing suffixes in the tree, one does not have to traverse all the leaves. One can use a global reference. The extensions ensure maximization of suffixes overlapping in the tree. Moreover, Ukkonen also used *edge label compression* in the algorithm by not saving the exact symbols for edge labels. Instead, it stores only pointers to the starting and ending position of the input sequence. Each pass inserts a new symbol to the tree. Before each pass, every existing node must have a suffix link to some other node, and active point must be maintained accordingly.

However, many existing works in time series do not give solution for handling the data structure in a dynamic environment. To handle sliding window problem

Fig. 3: Sliding window

in time series, many existing works build the structure from scratch for each new window. Let us illustrate the sliding window problem by Fig. 3, in which the window size is considered to be 9 (symbols). Window 1 contains the sequence "abcababab". After the arrival of new discretized input symbol 'c', the window slides and we get new modified window with (i) symbol 'a' is deleted from the beginning of the old window and (ii) symbol 'c' is inserted at the end. When the time series is represented by a suffix tree, if we have a sequence S and a suffix tree T for S, we need to be able to *efficiently* update T in case of (i) insertions of new symbols at the end of S or (ii) deletion of symbols from the beginning of S.

As a preview, to alleviate this problem, we propose DTSW, which is a framework for handling *both insertion and deletion* of events in a *single* framework. The basis of our solution is to keep the tree consistent so that, any time insertion or deletion of events is possible. The idea of making a data structure consistent for batch events was inspired by Leung and Khan's work on the DSTree [11], which aims to keep the tree consistent for future updates and makes only necessary modifications to reflect the current data under consideration.

2.2 Pruning for Weighted Periodical Pattern Mining

Introduction of weight has been an important concept in pattern mining because it helps to find patterns with more important features [1, 2] and is popular in time series. Many existing studies [7, 18] mine (unweighted) periodic patterns from time series by using the downward closure property to speed up the candidate generation process. Related works [6] for mining *weighted* periodic patterns from time series use the weight of maximum weighted character/item in the time series to reduce the number of unnecessary patterns tested, and thus to speed up the candidate generation.

As a preview, we propose the MPWS pruning solution. It provides a much tighter bound so as to reduce the number of candidates to be tested by using a heuristic value for the patterns.

3 Our Dynamic Tree Based Solution to Handle Sliding Window in Time Series (DTSW)

Our dynamic tree based solution to handle sliding window in time series (DTSW) consists of two modules: (i) A module for *handling deletion events*, which updates suffix tree if we delete some symbols from the starting of the sequence; and (ii) a module for *handling insertion events*, which updates our tree if we insert new symbols to the end of the sequence.

3.1 Handling Deletion Events

Deleting a symbol from the starting of a sequence means deletion of the largest suffix from the sequence. For example, if we have sequence "abcababababc", then removing the first symbol 'a' from the sequence means deleting the largest suffix "abcababababc" from the sequence resulting in sequence "bcababababc". So, the problem centers around how we can delete a suffix from the suffix tree. Hence, we define our Condition 1.

Condition 1. *Before deleting any suffix from the suffix tree, the tree must be in its explicit form.* Main reasoning behind this is, if the tree is in explicit form, then it is always enough to remove a leaf node from the tree to delete a suffix. For example, if we want to remove suffix "abcababababc$" from the explicit tree of Fig. 1, it is sufficient to remove Node 3 from the tree. Moreover, by definition, deletion of suffixes from a sequence goes from larger to smaller suffixes. Let us discuss some possible scenarios resulted from the deletions nodes and the ways to tackle them. We will state them as propositions.

Proposition 1 (Conversion from internal to leaf node). If the parent of a node V (say, U) loses all of its child nodes after removing V from an explicit suffix tree, then if U is not root, we will (i) convert U to a leaf node from an internal node and (ii) if any node W was pointing to U as its suffix link, then the suffix link of W will be redirected to root node. Reasoning behind this redirection lies in definition of suffix links that point from an internal node to another internal node. Path symbols from the root to any node X is unique because of the tree structure. So, suffix link of W must be redirected to the root.

Proposition 2 (Merging a splitted path). Suppose we remove node V for deletion from an explicit suffix tree. After the deletion, if (i) parent of V (say, U) becomes a node having a single child node W and (ii) U is not the root and has a parent node X, then we will (i) delete node U, (ii) merge the split path between X to U and U to W, and (iii) redirect the suffix link to root if any internal node Y was pointing to U as its suffix link. For example, from Fig. 1, after removing Node 3, Node 2 will only have a single child Node 4. Then, we will remove Node 2, and merge the path between Node 1 to Node 2 and the path between Node 2 to Node 4. Here, no node was pointing to Node 2 as its suffix link. Otherwise, we would have redirected to the root because path symbols "abc" (from the root to Node 2) would not have repeated elsewhere in the tree (from the root). This proposition is essential to maintain our Condition 1 and insertion module.

3.2 Handling Insertion Events

Our proposed solution DTSW is a complete framework for maintaining a dynamic suffix tree to handle sliding window, where our algorithm considers both *insertion* and *deletion* as two independent modules. Our solution is capable to (i) update the suffix tree for multiple insertion or deletion events and (ii) keep the structure consistent for future updates.

To *convert an implicit suffix tree to an explicit suffix tree*, a unique symbol is added to the tree. A symbol that does not exist in the sequence (upon which suffix tree is built) is considered as a unique symbol. This addition creates many nodes, splits many paths, and converts every implicit suffix explicit. Fig. 1 is the explicit suffix tree of string "abcababababc$", where main string is "abcababababc" and '$' is the unique symbol; implicit suffix tree of "abcababababc" is shown in Fig. 2. Although both figures represent the same suffixes, Fig. 1 has an advantage of ending all suffixes in leaves and ignoring the last symbol from each suffix we can extract the main suffixes to work with.

The main goal of our *insertion module* is to convert the tree to such an extent that the Ukkonen's algorithm can be used to insert symbols to the tree. Key steps include:

1. Conversion from explicit to implicit: We will revert back the tree from explicit to implicit form, which means we will remove the unique symbol and erase all the effects created due to it.
2. Finding new active point: Each pass of the Ukkonen's algorithm starts from the largest implicit suffix of the tree. After Step 1, some explicit suffixes will become implicit, then we need to find the largest implicit suffix's position and update the *active point* for the new pass.

There exists many reasons behind Step 1: (i) As unique symbol is not part of the input, this symbol has to be removed from the tree before any new addition. If we keep unique symbol, then adding new input and extracting the main suffixes will be costly. (ii) Addition of unique symbol creates some extra nodes and split paths in the tree. If we do not revert back the effect before new insertion, maximization of overlapping suffixes will not be ensured. The compact nature of the tree will be violated. Consider '$' as our unique symbol. Let us discuss cases which can occur due to addition of '$' and we have to revert back those effects:

1. *Child node V created from an existing node for '$'.* In this case, we have to remove the child node V. Because of deletion, if parent of V (say, U) loses all its children and U is not the root, then we have to convert U to a leaf node following Proposition 1 and if U remains with only one single child node, then we have to delete U and merge the path following Proposition 2.
2. *Child node V created by splitting an existing path for '$'.* This case can be explained from Figs. 2 and 1. Due to addition of '$' path between Node 1 and Node 3 gets split. New node 2 is inserted in between them, and then Node 4 is created for '$'. To revert back this case, we will first delete node V and then merge the split path by following Proposition 2. Here, in our example, we will first delete Node 4, then delete Node 2 and merge the path between Nodes 1 to 2 and Nodes 2 to 3. We would also have redirected the suffix link if any suffix link was pointing to Node 2 to the root.

Step 2 is find active point for the new pass. The whole process and reasoning can be provided as follows:

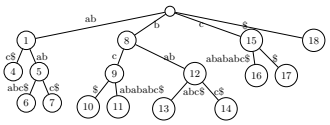


Fig. 4: Deletion

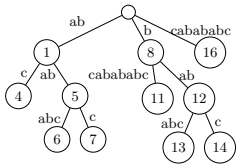


Fig. 5: Conversion

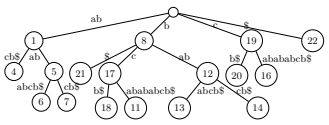


Fig. 6: Insertion

– In Step 1, we convert explicit suffixes to its implicit form. So, we can actually count the number of suffixes that have been converted. This number denotes the length of the largest implicit suffix at the moment after conversion. Suppose the number is l. So, all the suffixes present at most 1 distance from the end of the sequence will be implicit now, because a suffix becomes implicit along with all of its smaller sub suffixes. Moreover, suffix deletion is also sequential, larger suffixes will be effected first before its smaller sub-suffixes.
– Then, we will revert back the effects in reverse order by which the nodes were created or paths were split due to addition of '$'. So, while erasing the effects, if we encounter a node which has been modified, we stop reverting because all of the previous effects created due to addition of '$' have been compromised. So, we already found the largest implicit suffix of the tree. Now, by traversing the tree, we can find its position and update active point (aka *active Node*), *active Edge* and *active Length*. These information can also be saved while erasing the effects. As an example, if we want to have the tree of Fig. 2 from Fig. 1, we will revert back the effects of Nodes 18, 15, 9 and 2, respectively. Removing child for '$' from the root does not help determining the largest implicit suffix because it is a dummy node.

Let us consider a simulation of our algorithm. Suppose we had a window of string "abcababababc" (the explicit tree for this window is shown in Fig. 1) and then we get a new symbol 'b' and our window slides. Fig. 4 shows the tree after deletion of 'a', Fig. 5 shows the tree after conversion and from explicit to implicit with the largest implicit suffix "bc", and Fig. 6 shows the resultant explicit tree after addition of 'b'.

4 Our Maximum Possible Weighted Support (MPWS) Pruning

Checking every pattern if they are weighted frequent (or weighted periodic) pattern is impractical. In unweighted version of pattern mining, the downward closure property (DCP) is used. As trivial DCP does not work in weighted pattern mining, most used technique is to use the weight of the maximum weighted character (*MaxW*) of the database to reduce the number of patterns tested. Testing a pattern means evaluating if that pattern can be a candidate pattern.

In this section, we propose a maximum possible weighted support (MPWS) pruning solution, which provides a tighter bound than the use of *MaxW*. In

the remainder of this section, we will use 0.8, 0.1, 0.2 and 0 as the weights of characters 'a', 'b', 'c' and '$', respectively.

Definition 1. ***sumW*(*N*)** *is defined as the sum of all the characters from the root to node N.*

Definition 2. ***weight*(*X*)** *is defined as the average weight of the characters of pattern X.*

Definition 3. ***min_sup*** *is defined as a user-specified support threshold with a real number between 0 and 100, and* $\boldsymbol{\sigma}$ *is defined as its corresponding normalized threshold:*

$$\sigma = \frac{min\_sup \times (MaxW \times length\_of\_dataset)}{100.0} \tag{1}$$

With the maximum weight of a character in the dataset be MaxW, no pattern can have weighted support greater than $MaxW \times length\_of\_dataset$.

Definition 4. ***weightedSupport*(*X*)** *is defined as a product of the average weight of the character of pattern X and the actual periodicity* $support(X)$ *of the pattern X:*

$$weightedSupport(X) = weight(X) \times support(X) \tag{2}$$

A pattern X is defined as **weighed periodic** *if* $weightedSupport(X) \geq \sigma$.

Definition 5. ***cnt*(*A*, *B*)** *is defined as the number of characters encountered on the path from node A to node B.*

Definition 6. ***maxW*(*A*, *B*)** *is defined as the weight of the character having the maximum weight among the characters on the path from node A to node B.*

Definition 7. ***sizeV*(*N*)** *is defined as the size of the occurrence of vector of node N. In other words, it captures the number of occurrence of the pattern that ends at node N.*

Definition 8. ***subStr*(*A*, *B*)** *is defined as the substring of the time series encountered on the path from node A to node B.*

For example, in Fig. 1, $sumW(Node\ 14)$ is the sum of weight of the characters 'b', 'a', 'b' and 'c', which is $0.1 + 0.8 + 0.1 + 0 = 1.2$. If X is "abac", then $weight(X) = \frac{0.8+0.1+0.8+0.2}{4} = 0.475$. In Fig. 1, $cnt(8, 13) = 6$, meaning that 6 characters are encountered on the path from Node 8 to Node 13. $maxW(8, 13) = 0.8$ means that the maximum weight among all 6 characters on the path from Node 8 to Node 13 is 0.8. $subStr(8, 13)$ is "ababc$", which is the substring of the time series encountered on the path from Node 8 to Node 13.

Definition 9. *Let (i) node P be the parent of node N, (ii) E be the edge between nodes N and P, and (iii) R be the root. Then,* ***nodeW*(*N*)** *is defined as the*

maximum possible weighted support of a pattern ending exactly above node N (between nodes N and its parent P):

$$nodeW(N) = \max\{A, B\} \times sizeV(N) \tag{3}$$

where

$$A = \frac{sumW(P) + maxW(P, N)}{cnt(R, P) + 1} \tag{4}$$

$$B = \frac{sumW(P) + maxW(P, N) \times cnt(P, N)}{cnt(R, P) + cnt(P, N)} \tag{5}$$

Definition 10. *Let (i) node P be the parent of node N, (ii) E be the edge between nodes N and P, and (iii) R be the root. Then, $\boldsymbol{S}$ is defined as a representative of all the patterns that end between some node N and its parent P:*

$$S \leftarrow S_1 + S_2 \tag{6}$$

where

$$S_1 \leftarrow subStr(root, P) \tag{7}$$

$$S_2 \leftarrow \text{any nonempty prefix of } subStr(P, N) \tag{8}$$

Lemma 1. *$nodeW(N) \geq weightedSupport(S)$ always holds.*

Proof. By Eq. (3), $nodeW(N) = \max\{A, B\} \times sizeV(N)$ where values of A and B can be computed by Eqs. (4) and (5), respectively. And, by Eq. (2), $weightedSupport(S) = weight(S) \times support(S)$. Here, $\max(A, B)$ gives the maximum possible value of $weights(S)$ under any circumstances. Consider the following three cases:

1. $weight(S_1) > maxW(P, N)$: In this case, even if all the characters in E has the same weight as $maxW(P, N)$, $weight(S)$ cannot be greater than A. Because Eq. (4) assumes that S_3 has length 1. If we increase length of S_3, $weight(S)$ will gradually decrease. So, A is the maximum possible value of $weight(S)$ in this case.
2. $weight(S_1) < maxW(E)$: We need an upper bound for $weight(S)$. So, let us assume all the characters in E has weight equal to $maxW(P, N)$. Then, $weight(S)$ will gradually increase with the increasing length of S_3. We get the value of B (see Eq. (5)) by assuming S_3 has maximum possible length. So, B is the maximum possible value of $weight(S)$ in this case.
3. $weight(S) = maxW(P, N)$: In this case, the length of S_3 does not matter.

So, $\max\{A, B\} \geq weight(S)$, and $sizeV(N) \geq Support(S)$. Hence, $nodeW(N) = \max\{A, B\} \times sizeV(N)$, and thus $nodeW(N) \geq weightedSupport(S)$. □

Definition 11. ***MPWS(N)*** *is defined as the maximum value among nodeW() of all the nodes in the subtree of node N. Subtree of node N includes itself. Let $nodeW(N)$ be the maximum possible weighted support pattern S. Then, $MPWS(N)$ is the maximum of nodeW of all the nodes in the subtree, and it is the maximum possible weighted support any pattern can achieve that has S_1 as a prefix.*

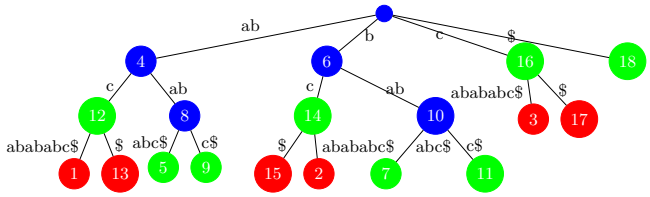


Fig. 7: Example of our MPWS pruning solution

Table 1: MPWS pruning necessary values calculated for Fig. 7

Node	sizeV	A	B	nodeW	MPWS
1	1	0.48	0.68	0.68	0.68
2	1	0.37	0.67	0.67	0.67
3	1	0.50	0.73	0.73	0.73
4	4	0.80	0.80	3.20	3.20
5	1	0.52	0.62	0.62	0.62
6	4	0.10	0.10	0.40	1.13
7	1	0.45	0.60	0.60	0.60
8	2	0.57	0.62	1.25	1.25
9	1	0.40	0.37	0.40	0.40

Node	sizeV	A	B	nodeW	MPWS
10	2	0.45	0.57	1.13	1.13
11	1	0.30	0.28	0.30	0.30
12	2	0.37	0.37	0.73	0.73
13	1	0.28	0.28	0.28	0.28
14	2	0.15	0.15	0.30	0.67
15	1	0.10	0.10	0.10	0.10
16	2	0.20	0.20	0.40	0.73
17	1	0.10	0.10	0.10	0.10
18	1	0.00	0.00	0.00	0.00

Candidate generation. Candidate patterns can be generated by a breadth first search (BFS) in the suffix tree. Following Definition 10, when we reach node N in the breadth first search, for every S, if $weight(S) \times sizeV(N) \geq \sigma$, then we will consider S as a *candidate pattern*.

Pruning condition. In the suffix tree for a string of size L, the number of nodes will be around N. However, the sum of the number of characters in the edges can be close to L^2. Thus, there can be around L^2 possible patterns in the dataset.

The candidate generation process mentioned above tests every pattern and makes that a candidate if it passes the test. However, checking every pattern is time consuming. So, we have to find a better pruning condition that reduces the number of patterns checked. The most commonly used technique is to use the weight of the maximum weighted character ($MaxW$) of the database. If $MaxW \times support(P) < \sigma$, any super pattern of P cannot be weighed frequent. So, those patterns cannot be periodic patterns either.

Lemma 2. *For any child C of node N, if $MPWS(C) < \sigma$, then we can ignore the whole subtree of C. This means that we do not need to visit any node in the subtree during the candidate generation of BFS.*

Proof. It can be easily proved because any node U in the subtree of C will not have $nodeW(U) \geq \sigma$ according to the definition of $MPWS(N)$. □

All the candidate patterns are actually *weighted frequent subsequence* of the current time series. To check if they are also periodic patterns, we can test the occurrence vector of each candidate pattern with different period values using known periodicity detection algorithms.

An example of our MPWS pruning solution is shown in Fig. 7 and Table 1. Here, the figure shows a suffix tree of string "abcababababc$" with $min\_sup = 10\%$. Detailed calculation of MPWS and other associated values is shown in Table 1. There are 3 types of nodes in Fig. 7:

1. Any pattern that has a *blue node* (e.g., Nodes 4, 6, 8 and 10) in its subtree is tested. For example, only patterns "a","ab","aba","abab","b","ba","bab" has a blue node in their subtree. So, only these patterns are tested for candidacy.
2. A *green node* N means that, starting from N, its whole subtree is unimportant and can be ignored during candidate generation BFS. Nodes 5, 7, 9, 11, 12, 14, 16 and 18 are green nodes.
3. All the nodes having a green node as an ancestor are *red nodes* (e.g., Nodes 1, 2, 3, 13, 15 and 17).

In this example, according to MPWS pruning, only seven patterns are tested for candidacy. Five of them eventually become candidate patterns. There are 50 patterns in total that had to be tested if we did not use any pruning. If we used the traditional $MaxW$ pruning, we had to test 10 patterns.

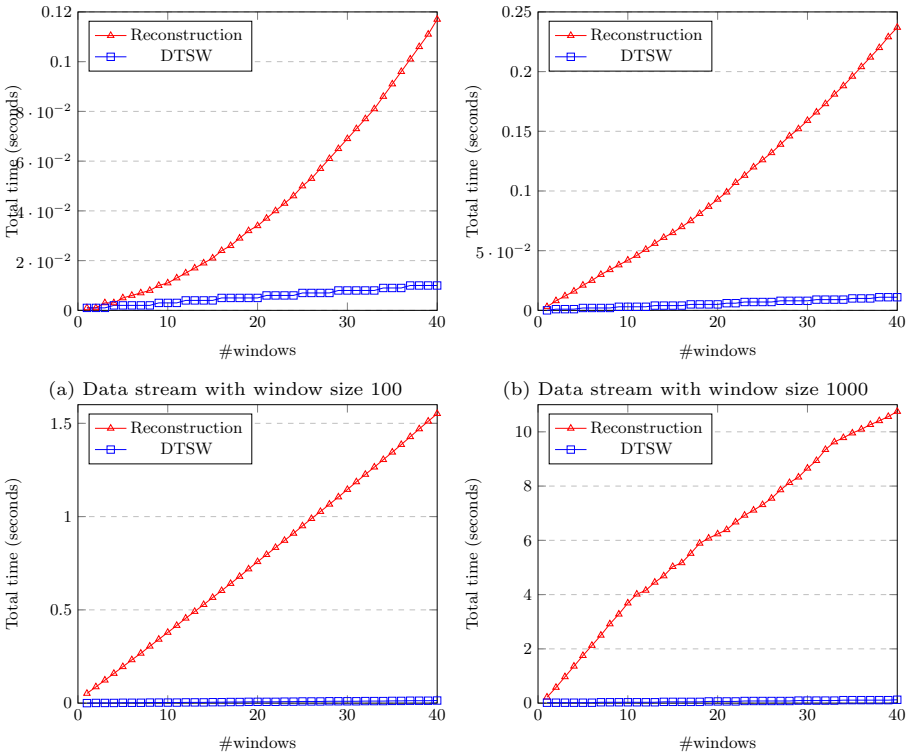
Additional complexity for pruning. If we build a suffix tree for a string of size L, there can be at most $2 \times L$ nodes in the suffix tree. During candidate generation, we first determine the MPWS value for all the nodes, which can be done by a depth first search (DFS) on the tree. We need the $MaxW$ value for each edge during that DFS. We have determined it using range minimum query (RMQ) in static data. As the query complexity for each edge is O(1), the added complexity by MPWS pruning becomes O(L).

5 Evaluation Results

We have used several data sets taken from UCI Machine Learning Repository [8] to compare our solutions with existing approaches. As all of them show consistent results, we will be showing the results of the following three datasets, which were discretized into string of characters:

1. Individual household electric power consumption dataset, which consists of 50000 events discretized into 13 types;
2. Appliances energy prediction dataset, which consists of 19735 events discretized into 12 types; and
3. Diabetes dataset, which consists of 2400 events discretized into 37 types.

All the codes were written using the C++ programming language. We used a machine having AMD Ryzen 5 machine with 1600 CPU (3.2 GHz) and 8 GB RAM for evaluation.



(a) Data stream with window size 100

(b) Data stream with window size 1000

(c) Data stream with window size 10000

(d) Data stream with window size 30000

Fig. 8: Sliding window with window sizes 100, 1000, 10000 and 30000

5.1 Evaluation on the DTSW Algorithm

Existing works on time series do not handle the sliding window based problem. Hence, they build the data structure from scratch for every window sliding, which can be inefficient. We show a comparison of the experimental result between DTSW and reconstruction of the tree for every window. Building the tree again for each new window performs poorly when the window size is large or the number of windows is large. In these scenarios, DTSW is very useful.

In Fig. 8, we show four graphs for four different window sizes from the individual household electric power consumption dataset. (As results for the other three datasets are similar, we omit them.) In the figures, the x-axis shows the number of windows passed and the y-axis shows the total time taken from the beginning. With the increasing window size, the performance of reconstruction gets worse, but DTSW runs very efficiently.

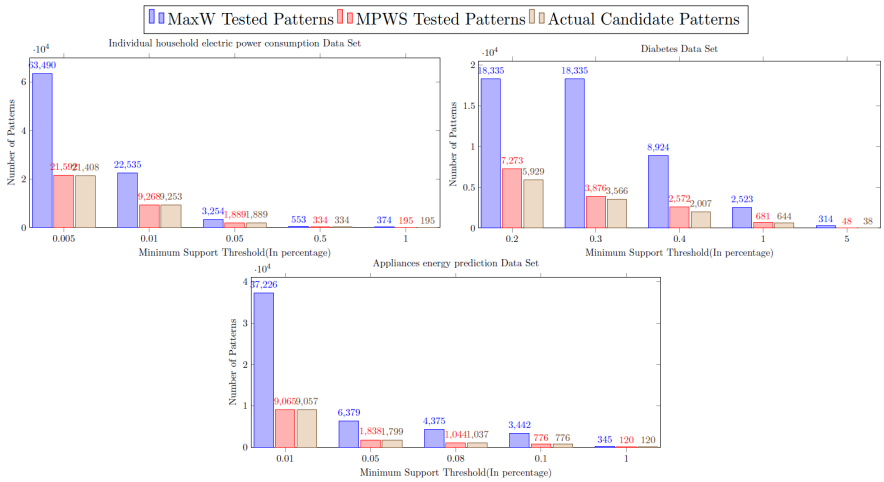


Fig. 9: MPWS pruning vs. MaxW pruning on varying minimum weighted support threshold

5.2 Evaluation on the MPWS Pruning

In the candidate generation process without any pruning, every pattern has to be tested to check if it can be a candidate. However, this is not practical as there can be many unnecessary patterns. Our main goal in the MPWS pruning is to avoid testing patterns that will not become a candidate pattern eventually.

For all the datasets, we have discretized them and assigned each unique character a weight that follows a normal distribution (μ=0.5 and σ=0.2). For different weighted support thresholds, we have compared our MPWS pruning with the traditional *MaxW* pruning on all the databases. Fig. 9 shows that our MPWS pruning tests much fewer patterns when compared with the traditional *MaxW* pruning. For example, in the individual household electric power consumption dataset, when the minimum support threshold is 0.005%, if we try to optimize the candidate generation process by using only *MaxW* in the database, it checks 63, 490 patterns. In contrast, mining with our MPWS pruning checks only 21, 592 patterns. With only 21, 408 actual candidate patterns, our MPWS pruning significantly reduces the number of tested patterns to close to the number of actual candidate patterns. Moreover, our MPWS pruning is observed not to test more patterns than the traditional *MaxW* pruning. In fact, our MPWS pruning is guaranteed to test no more patterns than the traditional *MaxW* pruning.

6 Conclusions

In this paper, we solved two important problems in time series pattern mining. Our *dynamic tree based solution to handle sliding window in time series* (*DTSW*)

is an algorithm for solving the sliding window problem. Our *maximum possible weighted support* (*MPWS*) *pruning* is a technique that reduces the number of patterns to be tested for candidacy. Both solutions are shown to be more efficient than existing approaches. Note that both of these solutions are independent of each other and can be used as two separate modules. Specifically, our first contribution—namely, *DTSW*—is an algorithm to dynamically update suffix tree. It is adaptable to run time dynamic window size, and is applicable for both weighted and unweighted framework. It solves the challenge of dynamic time series data. Our second contribution—namely, *MPWS*—can be used in different kinds of weighted pattern mining (with necessary modifications) in place of traditional *MaxW* because of its unique style for approximating an upper bound. As ongoing and future work, we are extending our solutions using dynamic weights in time series.

Acknowledgements

This project is partially supported by NSERC (Canada) and University of Manitoba.

References

1. Ahmed, C.F., Tanbeer, S.K., Jeong, B.S., Lee, Y.K.: An efficient algorithm for sliding window-based weighted frequent pattern mining over data streams. IEICE TIS E92-D(7), 1369–1381 (2009)
2. Ahmed, C.F., Tanbeer, S.K., Jeong, B.S., Lee, Y.K.: Handling dynamic weights in weighted frequent pattern mining. IEICE TIS E91-D(11), 2578–2588 (2008)
3. Almusallam, N., Tari, Z., Chan, J., AlHarthi, A.: UFSSF - an efficient unsupervised feature selection for streaming features. In: PAKDD 2018, Part II. LNCS (LNAI), vol. 10938, pp. 495–507 (2018)
4. A'yun, K., Abadi, A., Saptaningtyas, F.: Application of weighted fuzzy time series model to forecast Trans Jogja's passengers. IJAPM 5(2), 76–85 (2015)
5. Braun, P., Cuzzocrea, A., Leung, C.K., Pazdor, A.G.M., Souza, J.: Item-centric mining of frequent patterns from big uncertain data. Procedia Computer Science 126, 1875–1884 (2018)
6. Chanda, A.K., Ahmed, C.F., Samiullah, M., Leung, C.K.: A new framework for mining weighted periodic patterns in time series databases. ESWA 79, 207–224 (2017)
7. Chanda, A.K., Saha, S., Nishi, M.A., Samiullah, M., Ahmed, C.F.: An efficient approach to mine flexible periodic patterns in time series databases. EAAI 44, 46–63 (2015)
8. Dheeru, D., Karra Taniskidou, E.: UCI machine learning repository (2017), `http://archive.ics.uci.edu/ml`
9. Kane, A., Shiri, N.: Multivariate time series representation and similarity search using PCA. In: ICDM 2017. LNCS (LNAI), vol. 10357, pp. 122–136 (2017)
10. Leung, C.K., Hoi, C.S.H., Pazdor, A.G.M., Wodi, B.H., Cuzzocrea, A.: Privacy-preserving frequent pattern mining from big uncertain data. In: IEEE BigData 2018, pp. 5101–5110 (2018)

11. Leung, C.K., Khan, Q.I.: DSTree: a tree structure for the mining of frequent sets from data streams. In: IEEE ICDM 2006, pp. 928–932 (2006)
12. Leung, C.K., Zhang, H., Souza, J., Lee, W.: Scalable vertical mining for big data analytics of frequent itemsets. In: DEXA 2018, Part I. LNCS, vol. 11029, pp. 3–17 (2018)
13. Mantuan, A.B., Fernandes, L.A.F.: Spatial contextualization for closed itemset mining. In: IEEE ICDM 2018, pp. 1176–1181 (2018)
14. Morris, K.J., Egan, S.D., Linsangan, J.L., Leung, C.K., Cuzzocrea, A., Hoi, C.S.H.: Token-based adaptive time-series prediction by ensembling linear and non-linear estimators: a machine learning approach for predictive analytics on big stock data. In: IEEE ICMLA 2018, pp. 1486–1491 (2018)
15. Mozaffari, L., Mozaffari, A., L. Azad, N.: Vehicle speed prediction via a sliding-window time series analysis and an evolutionary least learning machine: a case study on San Francisco urban roads. JESTech 18(2), 150–162 (2015)
16. Phan, H., Le, B.: A novel parallel algorithm for frequent itemsets mining in large transactional databases. In: ICDM 2018. LNCS (LNAI), vol. 10933, pp. 272–287 (2018)
17. Rahman, M.M., Ahmed, C.F., Leung, C.K.: Mining weighted frequent sequences in uncertain databases. Inf. Sci. 479, 76–100 (2019)
18. Rasheed, F., Alshalalfa, M., Alhajj, R.: Efficient periodicity mining in time series databases using suffix trees. IEEE TKDE 23(1), 79–94 (2011)
19. Singh, R., Graves, J.A., Talbert, D.A., Eberle, W.: Prefix and suffix sequential pattern mining. In: ICDM 2018. LNCS (LNAI), vol. 10933, pp. 309–324 (2018)
20. Ukkonen, E.: On-line construction of suffix trees. Algorithmica 14(3), 249–260 (1995)
21. Yan, X., Wang, Z., Yu, S., Li, Y.: Time series forecasting with RBF neural network. In: ICMLC 2005, pp. 4680–4683 (2005)