Sliding Window Based Weighted Periodic Pattern Mining over Time Series Database

Redwan Ahmed Rizvee[1], Md Shahadat Hossain Shahin[1], Chowdhury Farhan Ahmed[1], and Carson K. Leung[2]

[1] Dept. of Computer Science and Engineering, University of Dhaka, Bangladesh
{rarndc@gmail.com,shshahin065@gmail.com,farhan@du.ac.bd}
[2] Dept. of Comput. Sci., Univ. of Manitoba, Winnipeg, MB, Canada
{kleung@cs.umanitoba.ca}

Abstract. Sliding window based problems have always been crucial in data mining which has led to several types of research. In time series mining, existing literature enforce on the reconstruction of the underlying structure (*Suffix Tree*) for each new window. But reconstruction performs poorly when the window size is large or when the sliding occurs frequently. Hence, in this paper, we are proposing a solution which dynamically updates the structure rather than reconstruction for each modification or sliding. Moreover, we also propose another solution to address the problem associated with mining weighted periodic patterns from time series. Existing works mostly rely on the weight of the maximum weighted item in the database to avoid testing unnecessary patterns. However, these approaches still check a lot of patterns if they can be a candidate. Our solution aims to speed up the candidate generation process by discarding many unimportant patterns beforehand. Experimental results of applying our two solutions on many real-life datasets show their effectiveness 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 a very important issue in knowledge discovery. The idea of generating patterns has evolved over time and flooded a huge set of new domains. Sequential pattern mining is one of the burning domains in the field of pattern mining and time series pattern mining is a very renowned and widely discussed topic under sequential pattern mining.

The core input of time series database is data stream or a stream of sequence of events or items found with respect to time interval. Existing literature [10] propose, the best structure to represent time series is *Suffix Tree*, upon which *frequent patterns* are mined on different thresholds and conditions. Two very important concepts in data streams were discussed in [8, 3]. Data streams are *continuous, unbounded* and *not necessarily uniformly distributed.* This creates the challenge of *dynamicity.* This *dynamicity* is also the core of *sliding window* [1] problem which has numerous real life applications, such as - weather forecasting, natural disasters prediction etc. Sliding Window problem is also very popular in time series [9, 12]. To solve this problem existing literature enforce reconstruction of the data structure to represent the modified window each time. But this solution is very expensive in case of large window size or frequent sliding of window occurring. We propose a solution DTSW (*dynamic tree based approach to handle sliding window in time series*), which focuses on updating the data structure dynamically and maintain a dynamic tree rather than reconstructing for each modified window and keep the tree suitable for any kind of pattern mining. Our approach is also suitable to handle dynamic window size for any sliding window problem.

Our second contribution centers around mining *weighted periodical patterns* from time series. Introduction of weight to patterns helps in finding out more interesting patterns than it's unweighted counterpart [1]. Weighted periodical patterns in time series mean that the weighted sequences which occur at least a certain amount of times periodically along with a weight to satisfy the user specified threshold. Weighted pattern mining can be very useful in time series to discover interesting features [4], for example - if we analyze the transactions of a sport kit shop, we will see that the sold products vary with many parameters like time, event etc. Selling rate of football jersey increases after each four years when world cup hits. Weighted time series mining can be very useful in this regard to discover these interesting features. The main challenge in any weighted pattern mining is how to avoid testing *undesired candidates* to speed up the candidate generation process because the downward closure property (*DCP*) can not be applied directly in weighted versions of pattern mining. Existing works use *Max Weight* concepts to speed up candidate generation. However, existing works still need to test a good number of unnecessary patterns for candidacy which degrades performance. Our second contribution in this paper is MPWS Pruning (*Maximum Possible Weighted Support Pruning*), an efficient pruning approach to significantly reduce the number of patterns to be tested for candidacy. In this paper we propose solutions to two problems. They are,

- DTSW, a dynamic tree based approach to handle sliding window in time series (Sec. 3.4).
- MPWS Pruning, an efficient approach to speed up the candidate generation process in weighted periodic pattern mining(Sec. 3.5).

Section 2 contains the background study and existing works related to our domain. Section 3 consists of our proposed solutions to the problems. Section 4 gives a comparative analysis between our solutions and existing solutions and conclusions are drawn in Section 5.

2 Background and Related Works

Our contributions are separated into two separate modules. One is about data structure modification and other targets efficient pruning. In [10], it has been shown that suffix tree is the most efficient data structure to represent time series and for frequent periodical pattern mining. We have also considered suffix tree as our data structure and up to now the fastest way to construct suffix tree is by using **Ukkonen's algorithm**[11]. **Ukkonen's algorithm** is a linear time algorithm. But an interesting observation is that, existing works in time series do not give any solution regarding handling the data structure in a dynamic fashion. So, to handle sliding window problem in time series existing methodologies suggest to build the structure from scratch for each new window. There is no literature on time series that provides a framework to handle both insertion and deletion of events in a single framework. Our first contribution in this paper is **DTSW** which is a framework to solve this problem. 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 proposed in **DS Tree**[8]. The main goal of this work was to keep the tree consistent for future updates and bring only the necessary modifications to reflect the current data under consideration.

Introduction of weight has been a very important concept in pattern mining because it helps to find out patterns with more important features [1, 2] and it is also very popular in time series. Existing literature regarding periodic pattern mining from time series [10, 6] use downward closure property to speed up the candidate generation process. Weighted versions of the similar works [5] use weight of the maximum weighted character in the database to speed up the candidate generation in this regard. It helps to reduce the number of unnecessary patterns tested. Our proposal MPWS Pruning is a similar tool that reduces the number of candidates to be tested using a heuristic value for the patterns.

3 Proposed Approaches

Proposed approaches will be described in multiple sections. In Sec. 3.1, we will explain how a time series database is constructed. Sec. 3.2 will contain an idea about the problems which we have approached in this paper. Sec. 3.3 will contain a discussion regarding suffix tree structure. Sec. 3.4 and Sec. 3.5 will contain detailed explanation of the techniques introduced.

3.1 Discretization

Discretization is a technique to represent a group of data with a single symbol. Time series is basically information gathered with respect to time interval. Time series can be represented as a string or sequence of characters from a given set by discretizing the values. For example "abcabababc$" is a discretized time series sequence.

3.2 Problem Definition

In this paper, we approached two specific problems. **First** problem can be stated through Fig. 1. Fig. 1 is an example of sliding window problem in time series, where window size is considered as 9. Window 1, contained sequence "abcababab". After arrival of new discretized input symbol 'c', the window slides and we get new modified window where we delete symbol 'a' from the beginning of the previous window and insert symbol 'c' to the end to construct the new window. We have already mentioned the best structure to represent time series is *suffix tree* [10]. So, the analogous problem which we approached here is,

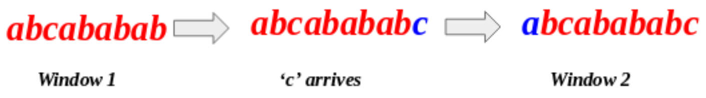


Fig. 1: Sliding Window

*If we have a sequence **S** and a suffix tree **T** for **S**, we need to update **T** efficiently in case of addition of new symbols at the end of **S** or deletion of symbols from the beginning of **S**.* Our proposed algorithm **DTSW** provides a complete framework to solve this problem.
The second problem centers around proposing an efficient pruning technique to mine weighted periodic time series patterns. Existing solutions use Maximum Weight ideas as pruning technique for candidate generation. Our proposed pruning technique **MPWS** provides a much tighter bound in this purpose.

3.3 Tree Structure

Suffix tree is used to represent time series database. The most efficient and compact way to construct suffix tree was proposed by Ukkonen [11]. We will use Ukkonen's algorithm to construct our initial suffix tree. This section will provide a brief discussion regarding the important concepts of Ukkonen's algorithm. These concepts will be helpful to understand the tree structure which will lead to better understanding of our dynamic tree solution *DTSW*.

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 **explicit** form. But if all the suffixes do not end in leaves but rather embedded in the paths then the tree is in **implicit** form. Fig. 2 is an explicit suffix tree of string "abcabababc$" and Fig. 3 is an implicit suffix tree of string "abcabababc".

An important concept in Ukkonen's algorithm is **suffix link** which helps to traverse the tree efficiently. According to Ukkonen's proposal each and every internal node of the tree will point to another internal node or root as its suffix link. Suffix link of a node **A** with path "$\alpha\beta$" from root where 'α' is exactly one symbol and 'β' can contain zero or more symbols will point to another

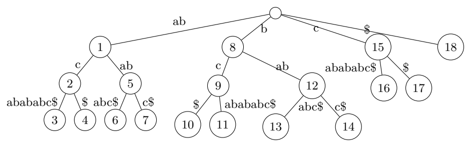


Fig. 2: Explicit Suffix Tree for string "abcababababc$"

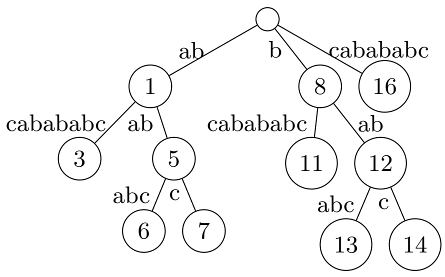


Fig. 3: Implicit Suffix Tree for string "abcabababc".

node(internal or root) **B** as its suffix link if and only if node **B** has the path 'β' from root. For example in the Fig. 2 node 1 points to node 8 as its suffix link.

Each pass of Ukkonen's algorithm to add symbol starts from the **active point** which denotes the position of largest implicit suffix in the tree at the current moment. Active point consists of **active node** which denotes the node position from which new pass will start, **active edge** provides the information about the edge from active node where suffixes overlapping is being occurred and **active length** denotes how many symbols been overlapped in the direction of active edge from active node.

Ukkonen proposed three **rule extensions** in his algorithm. Rule 1 extension says that, to add a new symbol to the end of all the existing suffixes in the tree we do not have to traverse all the leaves rather we can use a global reference. Rule 2 extension says about the reasoning behind the creation of new node from an existing tree node. Rule 3 extension ensures maximization of suffixes overlapping in the tree.

Ukkonen also used **edge label compression** in his algorithm by not saving the exact symbols for edge labels rather by storing 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.

3.4 A Dynamic Tree Based Approach to Handle Sliding Window in Time Series, ***DTSW***

This section will discuss about the solution regarding the first problem mentioned in our paper in section 3.2. The discussion will be divided into two modules. **Handling Deletion Events** module will explain how we can update suffix tree if we delete some symbols from the starting of the sequence and **Handling Insertion Events** module will explain how we can update our tree if we insert new symbols to the end of the sequence.

Handling Deletion Events: Deleting a symbol from the starting of a sequence means, deleting the largest suffix from the sequence. For example, if we have sequence “abcabababc”, then removing the first symbol ‘a’ from the sequence means deleting the largest suffix **“abcabababc”** from the sequence resulting in sequence “bcabababc”. So, the problem centers around how we can delete a suffix from the suffix tree and 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, from the explicit tree of fig. 2 if we want to remove suffix “abcabababc$”, it is enough to remove node 3 from the tree. Another important thing to notice here is, by definition deletion of suffixes from a sequence goes from larger to smaller suffixes. Now, we will discuss about the possible scenarios which can occur because of deleting nodes and the way to tackle them. We will state them as propositions.

Proposition 1. *Conversion from Internal to Leaf Node*, If after removing a node **V** from an explicit suffix tree, parent of **V**, suppose **U** looses all of its child nodes, then if **U** is not root we will convert **U** to a leaf node from an internal node and 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 which point from an internal node to another internal node and path symbols from root to any node **X** is unique because of the tree structure. So here **W’s** suffix link must be redirected to root.

Proposition 2. *Merging a Splitted path*, Suppose we remove node **V** for deletion from an explicit suffix tree. If after deletion parent of **V**, suppose **U** becomes a node having single child node **W** and if **U** is not root and has a parent node **X** then we will delete node **U**, merge the splitted path between **X** to **U** and **U** to **W** and redirect the suffix link to root if any internal node **Y** was pointing to **U** as its suffix link.
For example from the fig. 2, 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 node 2 to node 4. No node was pointing to node 2 as its suffix link, but if some node had done then we would have redirected to root. Because path symbols “abc” (from root to node 2) would not have repeated elsewhere in the tree (from root). This proposition is essential to maintain our condition 1 and insertion module.

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 approach is capable to update the suffix tree for any number of insertion or deletion events and keep the structure consistent for future updates. Before diving into discussion first we will explain the process how an implicit suffix tree is converted to an explicit suffix tree.

Conversion of Implicit to Explicit Suffix Tree: To convert an implicit suffix tree to an explicit suffix tree, an unique symbol is added to the tree. A symbol which does not exist in the sequence(upon which suffix tree is built) is considered as an unique symbol. What this addition does is it creates many nodes, splits many paths and converts every implicit suffix explicit. Fig. 2 is the explicit suffix tree of string "abcabababc$", where main string is "abcabababc" and '$' is the unique symbol. Implicit suffix tree of "abcabababc" is shown in fig. 3. Fig. 2 and fig. 3 both represents the same suffixes, but 2 gives advantage by ending all suffixes in leaves and just by ignoring the last symbol from each suffix we can extract the main suffixes to work with.

Insertion Module: Main goal of our insertion module is to convert the tree to such extent that same Ukkonen's algorithm can be used again to insert symbols to the tree. Steps are -

1. **Conversion From Explicit to Implicit:** First 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 we need to find the largest implicit suffix's position and update **active Point** for the new pass.

There exists many reasons behind step 1. **First** reason is, unique symbol is not part of the input. So, before any new addition, this symbol has to be removed from the tree and if we keep unique symbol and then add new input then extracting the main suffixes will be costly. **Second** reason is, 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 and the compact nature of the tree will be violated. Let's consider '$' as our unique symbol. Now, we will talk about the cases which can occur due to addition of '$'. We have to revert back those effects. They are -

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

Now, we will talk about the second step, how we can find active point for the new pass. The whole process and reasoning can be provided as follows.

- In step 1 we convert explicit suffixes to its implicit form. So we can actually get how many suffixes been converted. This number denotes the length of the largest implicit suffix at the current moment after conversion. Suppose the number is l. So, all the suffixes present at most l 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.
- What we will do, we will revert back the effects in reverse order by which the nodes were created or paths were splitted 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 and 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. 3 from fig. 2 we will revert back the effects of node 18, 15, 9 and 2 respectively. Removing child for '\$' from root does not help determining the largest implicit suffix because its a dummy node.

Now we will provide a simulation of our algorithm. Suppose we had a window of string "abcabababc"(the explicit tree for this window is shown in fig. 2) and then we get a new symbol 'b' and our window slides. In the following figures we have shown the steps, in fig. 4 after deleting 'a', in fig. 5 after conversion and from explicit to implicit with the largest implicit suffix "bc" and in fig. 6 after addition of 'b', the resultant explicit tree.

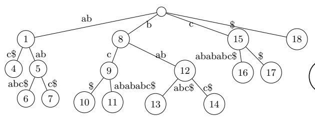


Fig. 4: deletion

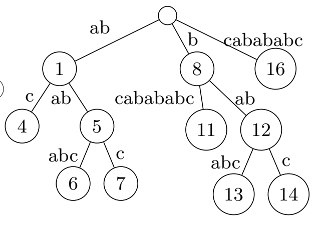


Fig. 5: conversion

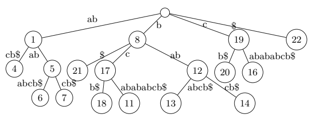


Fig. 6: insertion

3.5 Maximum Possible Weighted Support Pruning, *MPWS Pruning*

Checking every pattern if they are weighted frequent(or weighted periodic) pattern is impractical. In unweighted version of pattern mining, 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. We are proposing MPWS Pruning technique that always performs better than MaxW Pruning. Let's start with some definitions.

In the whole section, we will consider 0.8, 0.1, 0.2 and 0 as the weight of characters 'a', 'b', 'c' and '$' respectively.

Definition 1. ***sumW(N)***
sumW(N) denotes the sum of all the characters from root to node N. In Fig. 2, sumW(14) is the sum of weight of the characters 'b', 'a', 'b' and 'c' which is 1.2.

Definition 2. ***weight(X)***
weight(X) is the average weight of the characters of pattern X. If X is "abac" then weight(X) is $\frac{0.8+0.1+0.8+0.2}{4} = 0.475$

Definition 3. ***min_sup and*** σ
min_sup is real number between 0 and 100. Suppose, maximum weight of a character in the dataset is maxW.

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

Because, no pattern can have weighted support greater than $MaxW \times length\_of\_dataset$.

Definition 4. ***weightedSupport(X)***
$weightedSupport(X) = weight(X) \times support(X)$ *where support(X) denotes the actual periodicity of the pattern X.*
A pattern X is weighed periodic if, $weightedSupport(X) \geq \sigma$

Definition 5. ***cnt(A,B)***
cnt(A,B) denotes number of characters encountered on the path from node A to node B. In Fig. 2, cnt(8,13) is 6.

Definition 6. ***maxW(A,B)***
maxW(A,B) is the weight of the character having maximum weight among the characters on the path from node A to node B.
In Fig. 2, maxW(8,13) is 0.8.

Definition 7. ***sizeV(N)***
sizeV(N) is the size of the occurrence of vector of node N. It denotes the number of occurrence of the pattern that ends at node N.

Definition 8. ***subStr(A,B)***
subStr(A,B) is the substring of the time series encountered on the path from node A to node B. In Fig. 2, subStr(8,13) is "ababc$".

Definition 9. ***nodeW(N)***
Let node P be the parent of node N, E be the edge between node node N and P and R be the root.

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

$$B = \frac{sumW(P) + maxW(P,N) \times cnt(P,N)}{cnt(R,P) + cnt(P,N)} \tag{2}$$

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

Definition 10. *Let node P be the parent of node N, E be the edge between node node N and P and R be the root.*
$S_1 \leftarrow subStr(root, P)$,
$S_2 \leftarrow subStr(P,N)$, $S_3 \leftarrow$ *Any nonempty prefix of* S_2
and $S \leftarrow S_1 + S_3$

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

Proof. We know, $weightedSupport(S) = weight(S) \times support(S)$

$\max(A,B)$ is the maximum possible value of $weights(S)$ under any circumstances. Value of A and B can be calculated using Eqn. 1 and Eqn. 2.

Case 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) can not be greater than A (see Eqn. 1). Because Eqn. 1 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.

Case 2, $weight(S_1) < maxW(E)$. We need an upper bound for weight(S). So, let's 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 Eqn. 2) by assuming S_3 has maximum possible length. So, B is the maximum possible value of weight(S) in this case.

Case 3, When weight(S) and maxW(P,N) are equal, the length of S_3 doesn't matter.

So, $\max(A,B) \geq weight(S)$

Again, $sizeV(N) \geq Support(S)$.

As, $nodeW(N) = \max(A,B) \times sizeV(N)$, $nodeW(N) \geq weightedSupport(S)$

Definition 11. ***MPWS(N)***
MPWS(N) is the Maximum value among the nodeW of all the nodes in the subtree of node N. Subtree of node N includes itself.
nodeW(N) denotes the maximum possible weighted support pattern S(see Definition. 10) can achieve. As MPWS(N) is the maximum of nodeW of all the nodes in the subtree. Thus, MPWS(N) is the maximum possible weighted support any pattern can achieve that has S_1 *as prefix.*

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. But 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. But checking every pattern is time consuming. So we have to figure out a pruning condition that reduces the number of patterns checked.

The most 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 can not be weighed frequent. So, those patterns can not 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. Ignoring means we dont need to visit any node in the subtree during the candidate generation bfs.*

Proof. That's obvious because any node U in the subtree of C will not have $nodeW(U) \geq \sigma$ according to the definition of MPWS(N).

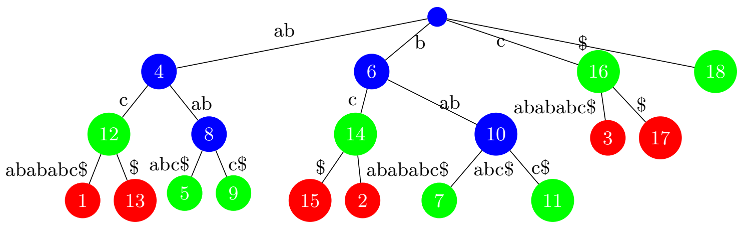


Fig. 7: Example of MPWS Pruning

All the candidate patterns are actually **weighted frequent subsequence** of the current time series. To check if they are periodic patterns too, we can test the occurrence vector of each candidate pattern with different period values using known periodicity detection algorithms [10].

An example of our pruning approach is given in Figure. 7 and Table. 1. Figure. 7 is the suffix tree of string "abcabababc$". min_sup is 10%. Detailed calculation of mpws and other associated values is shown in Table. 1.

There are 3 types of nodes in figure 7.

– Any pattern that has a blue node in it's 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.

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
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

Table 1: MPWS Pruning Necessary Values Calculated For Fig. 7

– A green node means, starting from that node, the whole subtree is unimportant and can be ignored during candidate generation bfs.
– All the nodes that has a green node as an ancestor are red nodes.

In this example, according to MPWS Pruning only 7 patterns are tested for candidacy. 5 of them eventually become candidate pattern. There are 50 patterns in total that had to be tested if we did not use any pruning. If we used MaxW Pruning, we had to test 10 patterns.

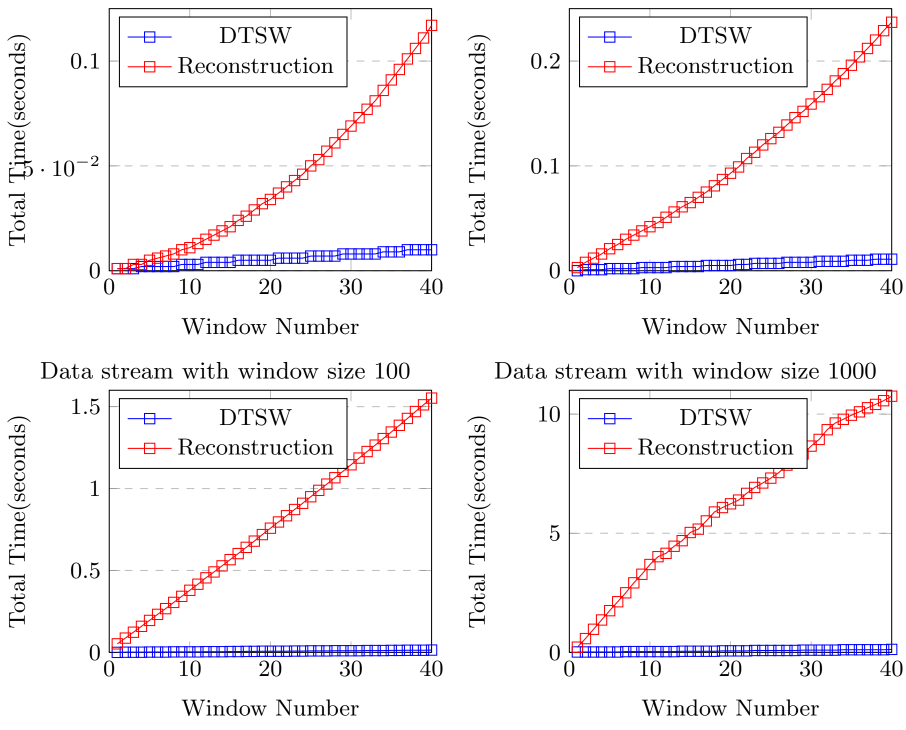
Additional Complexity for Pruning If we build a suffix tree for a string of size L, there can be at most $2 \times L$ nodes in the suffix tree. During candidate generation, we first determine the MPWS value for all the nodes which can be done by a depth first search(DFS) on the tree. We need MaxW value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

4 Experimental Results

We have used several data sets taken from UCI Machine Learning Repository [7] to compare our approach with the existing approaches. As all of them show similar results, we will be showing the results of the following 3 data sets. The datasets were discretized to turn them into string of characters.

– Individual household electric power consumption Data Set (50000 events, Discretized into 13 types)
– Appliances energy prediction Data Set (19735 events, Discretized into 12 types)
– Diabetes Data Set (2400 events, Discretized into 37 types)

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



Data stream with window size 100 Data stream with window size 1000

Data stream with window size 10000 Data stream with window size 30000

Fig. 8: Sliding Window with Window Size 100, 1000 and 30000

4.1 DTSW

Existing works on time series do not give any idea about how to handle sliding window based problems. There is no way but to build the data structure from scratch for every window. But this is an inefficient approach. We will 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 those scenarios, DTSW is very useful.

In Fig. 8, for four different window sizes, we have shown four graphs where the x-axis denotes the number of windows passed and the y-axis denotes the total time taken from the beginning. It is easily observable that with the increasing window size, the performance of reconstruction gets worse but DTSW still works very efficiently. The data for the graphs were taken from Individual Household Electric Power Consumption Data Set. For the other three datasets, the graphs show similar results.

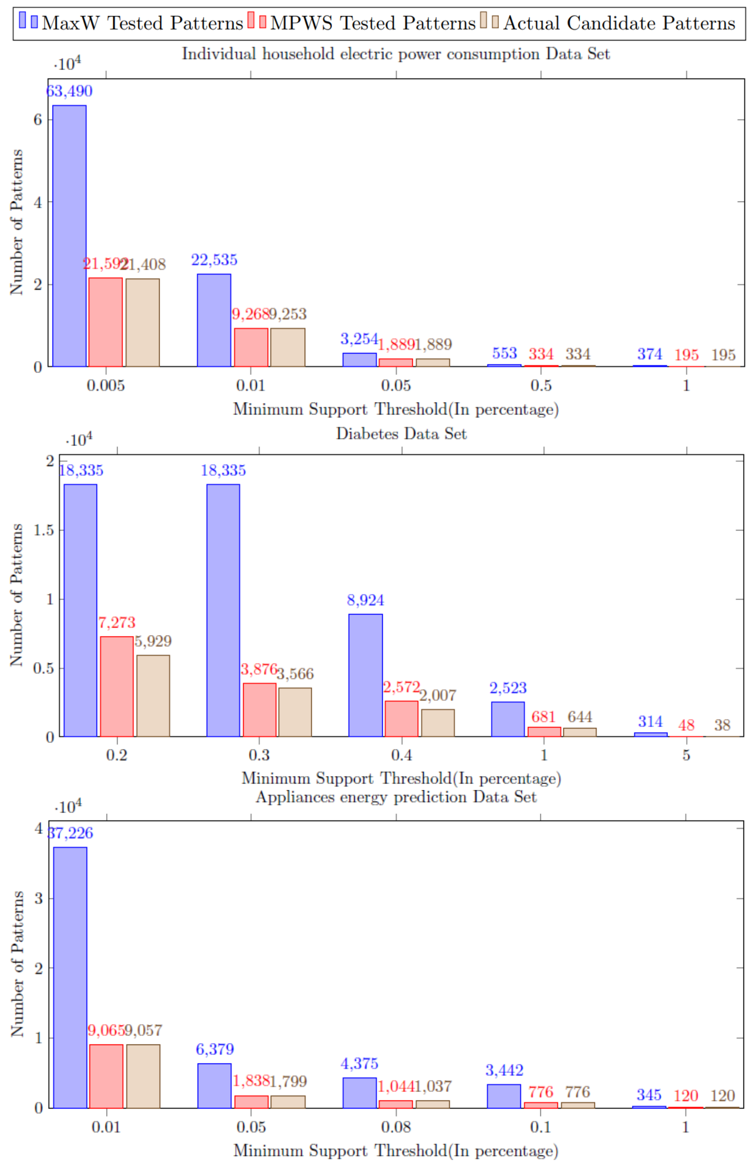


Fig. 9: MPWS Pruning vs MaxW Pruning on varying Minimum Weighted Support Threshold

4.2 MPWS Pruning

In candidate generation process without any pruning, every pattern has to be tested to check if it can be a candidate. But this is not acceptable as there can be many unnecessary patterns. Our main goal in MPWS Pruning was 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 which follows a normal distribution($\mu = 0.5$ and $\sigma = 0.2$). For different weighted support thresholds, we have compared MPWS Pruning with MaxW Pruning on all the databases. From Fig. 9, we see that MPWS Pruning tests much fewer patterns compared to MaxW Pruning.

For example, in the Individual Household Electric Power Consumption Data Set, when the minimum support threshold is 0.005%, if we try to optimize the candidate generation process using only MaxW in the database, it checks 63490 patterns whereas MPWS Pruning checks 21592 patterns only. Eventually, 21408 patterns become candidate patterns. MPWS Pruning prunes the testing of almost all the unnecessary patterns. In the given results, we see that MPWS Pruning doesn't test more patterns than MaxW Pruning. Actually, MPWS Pruning will never test more patterns than MaxW Pruning.

5 Conclusion

In this paper we solved two important problems in time series. Our first solution **DTSW** is a dynamic tree based solution to handle sliding window and our second solution **MPWS** is a pruning technique which reduces unnecessary candidate generation and both of them bring huge run time efficiency as demonstrated by our experimental analysis section. The most noticeable fact in our paper is both of these two contributions are independent of each other and can be used as two separate modules. Interesting observations regarding $DTSW$ are, it is an algorithm to dynamically update suffix tree and adaptable to run time dynamic window size and applicable for both weighted and unweighted framework. So, this solution approach can be applicable to many other problems and can create new research topic. Another important point to note that, $DTSW$ basically targets to solve the challenge of dynamic time series database. Our second contribution $MPWS$(with necessary modifications) can be used in different kinds of weighted pattern mining in place of traditional MaxW because of its unique style for approximating upper bound. Dynamic weights to time series mining has not been introduced yet. As ongoing and future work, we are planning to extend our solutions using dynamic weights in time series.

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 Transactions on Information and Systems 92, 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 TRANSACTIONS on Information and Systems 91(11), 2578–2588 (2008)

3. Almusallam, N., Tari, Z., Chan, J., AlHarthi, A.: Ufssf-an efficient unsupervised feature selection for streaming features. In: Pacific-Asia Conference on Knowledge Discovery and Data Mining. pp. 495–507. Springer (2018)
4. A'yun, K., Abadi, A., Saptaningtyas, F.: Application of weighted fuzzy time series model to forecast trans jogjas passengers. International Journal of Applied Physics and Mathematics 5, 76–85 (01 2015)
5. Chanda, A.K., Ahmed, C.F., Samiullah, M., Leung, C.K.: A new framework for mining weighted periodic patterns in time series databases. Expert Systems with Applications 79, 207–224 (2017)
6. 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. Engineering Applications of Artificial Intelligence 44, 46–63 (2015)
7. Dheeru, D., Karra Taniskidou, E.: UCI machine learning repository (2017), `http://archive.ics.uci.edu/ml`
8. Leung, C.K.S., Khan, Q.I.: Dstree: a tree structure for the mining of frequent sets from data streams. In: Data Mining, 2006. ICDM'06. Sixth International Conference on. pp. 928–932. IEEE (2006)
9. 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. Engineering Science and Technology, an International Journal 6 (12 2014)
10. Rasheed, F., Alshalalfa, M., Alhajj, R.: Efficient periodicity mining in time series databases using suffix trees. IEEE Transactions on Knowledge and Data Engineering 23(1), 79–94 (2011)
11. Ukkonen, E.: On-line construction of suffix trees. Algorithmica 14(3), 249–260 (1995)
12. Yan, X.B., Wang, Z., Yu, S.H., Li, Y.J.: Time series forecasting with rbf neural network. pp. 4680 – 4683 Vol. 8 (09 2005)