Mining Weighted Patterns from Time Series Databases Based on Sliding Window

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

[1]Department of Computer Science and Engineering, University of Dhaka, Dhaka, Bangladesh
[2]Department of Computer Science, University of Manitoba, Manitoba, Canada

*Email: farhan@du.ac.bd

Received on 04 July 2023, Accepted for Publication on 25 January 2024

ABSTRACT

Data mining has traditionally relied heavily on sliding window-based challenges, which has sparked a variety of studies. For each new window in time series mining, current literature mandates the rebuilding of the underlying structure, Suffix Tree - A trie-based structure representing all the suffixes of a string. However, reconstruction struggles when the window is wide or when sliding happens frequently. As a result, we provide a new technique Dynamic Tree-Based Approach to handle Sliding Windows (DTSW) in time series in this study that dynamically changes the representative suffix tree structure rather than reconstructing it after every alteration or sliding. In addition, we also put forth a different approach to the issue of extracting weighted periodic patterns from time series. To prevent testing pointless patterns, existing studies mostly rely on the weight of the database's highest-weighted item. However, these methods continue to examine numerous patterns. These methods still examine numerous patterns to see whether they can be candidates. Our proposed measure Maximum Possible Weighted Support (MPWS) accelerates the candidate generation process by removing numerous unnecessary patterns in advance. The novelty of MPWS is it considers the maximum weighted average over the maximum weighted item extension by enforcing more constraints. The usefulness of our two techniques in handling sliding windows and trimming redundant candidate patterns is demonstrated by experimental results using a variety of real-world datasets. Our experiments state that our dynamic handling technique significantly improves runtime than the reconstruction in a dynamic sliding window-based environment with simultaneous insertion and deletion actions and MPWS reduces the number of tested patterns resulting in lesser mining time in weighted time series pattern mining.

Keywords: Time Series, Weighted Periodic Pattern Mining, Dynamic Database, Sliding Window, Pruning Strategies.

1. Introduction

Finding a practical method for mining common patterns has always been crucial to knowledge discovery [15, 16, 17, 18, 19]. Over time, the concept of creating patterns has developed and permeated a vast array of new disciplines. Time series pattern mining is a well-known and widely debated subject within sequential pattern mining, which is one of the most renowned research domains in the field of pattern mining.

The main source of data for time series databases is a stream of events or other items found in relation to time. According to existing literature, *Suffix Trees*, on which *Frequent Patterns* are mined under various thresholds and conditions, are the best structures to describe time series. In [1][2] two crucial data stream concepts were covered. Data streams are *continuous*, *unbounded*, and *not always distributed evenly*. The result is the problem of *dynamicity*." The sliding window [7] problem, which has many real-world applications, such as - weather forecasting, natural disasters prediction, etc., is likewise based on this property of dynamicity. Time series [3] is also a fairly common application of the sliding window problem. The available literature mandates reconstruction of the data structure to reflect the updated window each time as a solution to this issue. However, if the windows are huge or slide frequently, this technique based on reconstruction is highly expensive. We suggest DTSW (*Dynamic Tree-based approach to handling Sliding Windows* in time series), which focuses on dynamically updating the data structure and maintaining a dynamic tree rather than reconstructing for each changed window and keeps the tree suitable for any kind of pattern mining. Our proposed solution can handle dynamic window sizes for any problem related to sliding windows. We focus on extracting weighted periodic patterns from time series in our second contribution. Compared to its unweighted sibling [7], the addition of weight to patterns enables the discovery of more intriguing patterns. Time series with weighted periodic patterns have weights that are sufficient to reach the user-specified threshold and at least a specific number of times per period. In order to find interesting characteristics in time series, weighted pattern mining can be quite helpful.

For instance, if we examine the transactions of a sports equipment store, we will see that the sold products fluctuate with many different criteria, such as time, event, etc. Every four years, when the world cup is held, the sales rate of football jerseys increases. To find these intriguing traits, weighted time series mining might be quite helpful.

Another example can be considered by analyzing the movie series' periodical patterns. We may understand that movies are frequently released considering special occasions (e.g., Christmas, Thanksgiving, Independence Day, etc.), targeting award ceremonies (e.g., Oscar, BAFTA, etc.), seasons, etc. In these databases, the information may be kept with time stamps which make them a time series database. Also, the box office revenue calculation is also very relevant to the time. For example, during the world cup season or any worldwide sports event, new movies are not generally released. But, we all know that, during the recent epidemic

coronavirus, the OTT (over-the-top) platform business hit a spark [20]. Because most of the people had to stay at home and work from home. So, those content providers tried to give their best efforts to bring a variety of content to the audience to entertain them. So, in summary, there are some events, when movies or specific types of movies get patronized over normal times. But, also there are some moments when they are slowed down. Weighted pattern mining, keeps a great impact here, by enforcing weights to extract specific types of patterns representing special nature over generic support-based frameworks. Also, as these databases are equipped with time information, periodical statistics can be quite helpful in this regard to analyze the general behavioral nature.

Since the downward closure property (DCP) cannot be directly applied in weighted versions of pattern mining, avoiding testing undesirable candidates to speed up the candidate generation process is the key problem. Max Weight principles are used in existing works to expedite candidate creation. Existing works, however, still require testing a sizable number of pointless patterns for candidacy, which worsens performance. The MPWS Pruning method, which is our second contribution to this paper, effectively prunes patterns to minimize the number of candidates that must be examined for candidacy. In this essay, we put forward remedies to address these two issues. They are

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

This article is an extended version of our work [21]. In this extended article, we provide more in-depth motivational real-life application-based examples, a wide range of background studies, a detailed discussion with examples of our proposed methodologies along with the necessary concepts, and a set of new extensive experimental discussions to understand the solutions' merits.

Our findings state that DTSW provides a novel generic approach to capture simultaneous insertion and deletion in a dynamic sliding widow-based scenario in a linear time. DTSW focuses on adjusting the representative suffix tree structure rather than reconstructing the complete structure which overall reduces processed time. MPWS is a generic pruning measure that can be applied in weighted time series pattern mining problems. By applying MPWS, we stop the generation of a good number of undesired candidates for a weighted support threshold constraint, which overall improves the mining runtime and does not add any severe resource bottleneck.

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

Sequential Pattern mining considers the sequential relationship among the elements of the database to discover interesting patterns [22]. Time series pattern mining is a subdomain under sequential pattern mining that addresses the ordered relationship among the entities considering timestamps [23, 24].

A wide range of literature has addressed different issues related to time series pattern mining [23, 24, 25]. Some notable key issues are, periodic pattern mining [6, 24], multivariate time series [25], time series forecasting [3, 11], weighted time series mining [7, 14, 21], etc. In this study, we address the problem of efficient data representation in the context of the sliding window problem in a time series database. We also address the issue of designing a compacter pruning strategy that the existing measures in the light of weighted time series pattern mining.

We divided our contributions into two distinct modules. One focuses on changing the data structure, while the other involves efficient pruning. It has been demonstrated in [5] that the suffix tree is the most effective data structure for representing time series and for mining frequently occurring patterns. The suffix tree has also been a potential candidate for our data structure, and as of right now, Ukkonen's approach is the quickest way to build one. The Ukkonen's algorithm runs in linear time. However, it's interesting to note that current time series research offers no guidance on how to handle the data structure dynamically. Because of this, existing approaches advise creating the structure from scratch for each new window in order to address the sliding window problem in the time series. To our knowledge, no time series literature offers a single framework that can handle both the addition and deletion of occurrences simultaneously. DTSW, a framework to address this issue, is our first contribution to this study. Our approach is based on maintaining the consistency of the tree, allowing for the insertion or deletion of events at any time. In DS Tree [1], the notion of making a data structure consistent for batch events was put forth. The main objective of this work was to maintain the tree's consistency for updates in the future and to introduce only the essential changes to reflect the data currently being taken into account.

The addition of weight has been a key idea in pattern mining since it aids in the discovery of patterns with more significant properties, such as [7] and [4], and it is also widely used in time series. Current research on periodic pattern mining from time series [5][6] makes use of the downward closure property to expedite candidate creation. In order to expedite the candidate generation in this case, weighted versions of related works [8] employ the weight of the highest weighted character in the database. It aids in lowering the number of pointless patterns tested. By utilizing a heuristic value for the patterns, our proposed MPWS Pruning is a similar tool that lowers the number of candidates to be assessed.

Time series-related literature has shaded up on various real-life problems, such as [11] proposed an idea to integrate sliding window and DTW distance to measure time-series forecasting tasks, [12] suggested an approach to find anomalies in time-stamped wireless sensor networks based on sliding windows, [13] characterized non-linear time series problem via sliding window amplitude based dispersion entropy approach, [14] proposed techniques to introduce weighted dynamic transfer network and spectral entropy for weak and non-linear detection in time series, [27] added the concept of the graph with temporal information to mine patterns, mining relevant patterns from multi-source time series database, etc.

So, the time series research domain is still very active. Moreover, the addition of sliding windows has also found its applications in numerous problems. Additionally, the concept of weights has also opened new challenges are problems that should be addressed. We believe our contributions can greatly help improve the performance of various solutions related to time series and its variations.

3. Proposed Approaches

In this section, we discuss our proposed strategies. In Section 3.1, we explain how a time series database is constructed. Section 3.2 presents an idea about the problems that we have approached in this paper. Section 3.3 contains a discussion regarding suffix tree structure. Section 3.4 and Section 3.5 contain a detailed explanation of the techniques introduced.

3.1 Discretization

A method known as discretization allows a collection of data to be represented by a single symbol. Information that has been acquired over a period of time is referred to as a time series. By discretizing the data, time series can be represented as a string or series of characters from a predetermined set. For instance, a discretized time series sequence is “abcababab$”.

3.2 Problem Definition

In this essay, we focused on two distinct issues. It is possible to state the first problem using Fig. 1. A sliding window problem in a time series is illustrated in Fig. 1, where the window size is nine. The sequence “abcababab”' was found in window 1. The window slides to reveal a new, changed window after the arrival of the new discretized input symbol “c”. To create the new window, we remove the symbol “a” from the beginning of the previous window and add the symbol “c” to the end. We've already mentioned that the suffix tree [5] is the optimal structure for representing time series. Therefore, the same issue that we addressed here is,

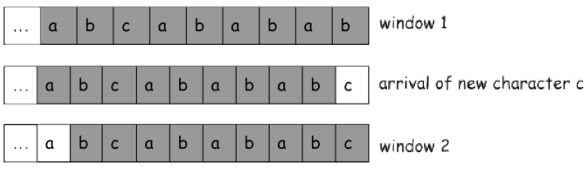

Fig. 1. Sliding Window

*“If we have a sequence called **S** and a suffix tree called **T** for **S**, we need to update **T** efficiently in the event that new symbols are added to **S**'s end or removed from its beginning”*. Our suggested algorithm, **DTSW**, offers a comprehensive framework to address this issue.

3.3 Tree Structure

The time series database is represented as a suffix tree. Ukkonen [9] suggested a method for building a suffix tree that was both the most effective and compact. We build our initial suffix tree using Ukkonen's technique. We shall have a quick overview of the key ideas in Ukkonen's algorithm in this section. These ideas will make it easier to comprehend the tree structure, which will help to understand the comprehension of our dynamic tree solution (DTSW).

All of a string's suffixes are represented by a suffix tree. The suffix tree is in the explicit form if all suffixes can be found by traversing from root to leaf nodes. However, the tree is in the implicit form if all of the suffixes do not finish in leaves but rather are embedded in the paths. An explicit suffix tree for the string “abcababab$” is shown in Fig. 2, while an implicit suffix tree for “abcababab” is shown in Fig. 3.

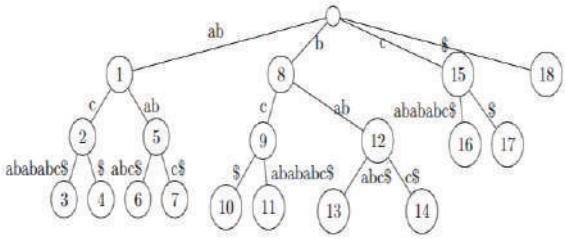

Fig. 2. Explicit Suffix Tree for string “abcababab$”

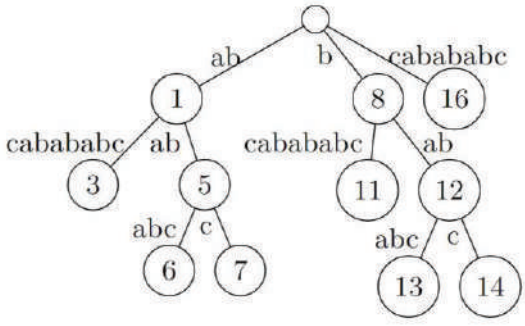

Fig. 3. Implicit Suffix Tree for string “abcababab”.

A key idea in Ukkonen's technique is the “**suffix link**” which facilitates swift tree traversal. Every internal node of the tree will, in accordance with Ukkonen's idea, point to another internal node or root as its suffix link. If and only if node **B** has the route "β " from the root, the suffix link of node **A** with the path “$\alpha\,\beta$ ” from the root, where “α ” is exactly one symbol and "β " can contain zero or more symbols, will point to node **B** as its suffix root. As an illustration, node 1 in the explicit suffix tree in Fig. 2 points to node 8 as its suffix link.

Ukkonen's approach for adding symbols begins each run at the **active point** which is the location of the largest implicit suffix in the tree at the time. The components of an active point are as follows,

1. **active node**: It indicates the position of the node from which a new pass will begin
2. **active edge**: It describes the edge of the active node where suffix overlapping is occurring,
3. **active length:** It indicates the number of symbols that have been overlapped in the direction of the active edge from the active node.

Three **rule extensions** were suggested by Ukkonen for his method. The extensions are described as follows,

- **Rule 1 extension**: This extension states that we do not need to traverse all of the leaves in order to add a new symbol to the end of all of the current suffixes in the tree; rather, we can utilize a global reference.
- **Rule 2 extension**: This extension states that during the extension for a particular symbol γ from the active node if no branch exists, we create a new branch for γ from the active node.
- **Rule 3 extension**: This extension ensures the maximization of suffixes along the suffix tree edges.

By keeping merely pointers to the beginning and end of the input sequence instead of the precise symbols for edge labels, Ukkonen also employed **edge label compression** in his technique. Every pass adds a fresh sign to the tree. Each existing node must have a suffix link to another node before each pass, and the active point must be kept correspondingly.

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

The answer to the first issue raised in our paper's section 3.2 will be covered in this section. There will be two parts to the debate. The Handling Deletion Events module will describe how to update the suffix tree if certain symbols are removed from the sequence's beginning, and the Handling Insertion Events module will describe how to update the tree if new symbols are added to the sequence's end.

3.4.1 Handling Deletion Events

Deleting the symbol from the beginning of a sequence means, deleting the largest suffix from the sequence. For instance, if the sequence is “abcababababc”, taking away the initial "a" from the sequence entails taking away the greatest suffix, “abcababababc” from the sequence, leaving us with “bcababababc”. Therefore, the issue is how to remove a suffix from the suffix tree, and this is why we define our Condition 1.

Condition 1: *Before deleting any suffix from the suffix tree, the tree must be in its explicit form*.

The main justification for this is that, if the tree is represented explicitly, it is always sufficient to remove a leaf node from the tree in order to delete a suffix. For instance, removing node 3 from the explicit tree of Fig. 2 is sufficient to remove the suffix "abcababababc" from the tree. Another crucial point to note is that, by definition, removing suffixes from a sequence result in the deletion of larger to smaller suffixes. We now talk about potential outcomes that could result from deleting nodes and how to deal with them. We'll put them forth as propositions.

Proposition 1 (Conversion from Internal to Leaf Node): If, after removing a node **V** from an explicit suffix tree, **V**'s parent, **U**, loses all of its child nodes, **U** will be converted to a leaf node from an internal node if it is not root, and if any node **W** was pointing to **U** as their suffix link, then the suffix link of **W** will be redirected to the root node.

The specification of suffix links, which point from one internal node to another internal node, and the fact that path symbols from the root to any node **X** are unique due to the tree structure explain why this redirection is necessary. Therefore, the suffix link for **W** must be directed to the suffix tree root.

Proposition 2 (Merging a Split path): Assume we remove a node **V** for deletion from an explicit suffix tree. Let the parent node of **V** is **U** and the parent node of **U** is **X**. If, after deletion, **U** becomes a single child node having **W**. Then, we remove **U** and make a single path by merging the edges **X** to **U** and **U** to **W**. If any node **Y** was pointing to **U** as its suffix link will be redirected to the root of the suffix tree.

As seen in Fig. 3, for instance, node 2 will only have one child node 4 when node 3 is removed. The path from node 1 to node 2 and node 2 to node 4 will then be combined when node 2 is removed. Node 2 wasn't pointed to by any nodes as its suffix link, but if it had, we would have been redirected to the root. because the path substring “abc” (from the root to the second node) would not have duplicated elsewhere in the tree (from the root). To keep our Condition 1 and insertion module operational, this notion is necessary.

3.4.2 Handling Insertion Events

A complete framework for maintaining a dynamic suffix tree to handle sliding windows is provided by our suggested method, DTSW, where our algorithm views Insertion and Deletion as two separate modules. Our method may update the suffix tree for any number of insertion or deletion events while maintaining a consistent structure for upcoming updates. Before detailing the process, first, we shall describe how an implicit suffix tree is transformed into an explicit suffix tree.

Converting an implicit suffix tree to an explicit suffix tree: An exclusive symbol is included in the tree to change it from an implicit suffix tree to an explicit suffix tree. A symbol is deemed unique or exclusive if it does not appear in the sequence (upon which the suffix tree is created). This addition generates a large number of nodes, divides a large number of pathways, and makes every implicit suffix apparent. The explicit suffix tree of the string "abcababababc" is shown in Fig. 2, where "abcababababc" is the main string and "$" is the special symbol. Fig. 3 displays the implicit suffix tree for the string "abcababababc". Both Fig. 2 and Fig. 3

represent identical suffixes, but Fig. 2. has the advantage of ending all the suffixes in leaves so that we may extract the primary suffixes to work with only by omitting the last symbol from each suffix.

Insertion Module: Our insertion module's primary objective is to transform the tree to the point where the same Ukkonen's technique may be applied once more to insert symbols into the tree. The steps are:

1. Conversion from explicit to implicit: First, we convert the tree from explicit to implicit form, removing the special symbol and all the impacts it had on the suffix tree.

2. Discovering a New Active Point: The Ukkonen's method begins each run at the tree's greatest implicit suffix. Following the first phase, some explicit suffixes will change to implicit ones. At this point, we must locate the largest implicit suffix and update the active point for the new pass.

There are numerous justifications for step 1. The first reason is that there is no unique symbol in the input. Therefore, this symbol must be eliminated from the tree prior to any new additions; otherwise, it will be expensive to extract the major suffixes when we add additional input and preserve the unique symbol. The inclusion of a distinctive symbol splits pathways and adds some extra nodes to the tree, which is the second reason. The maximization of overlapping suffixes will not be guaranteed, and the compact character of the tree will be broken, if we do not go back to the effect before the new insertion. Let us use "$" as our special symbol. We will now discuss the scenarios that can result from the insertion of "$". We need to undo such impacts. The scenarios are as follows,

Case 1 (Child node V formed from a "$" node that already existed): In this instance, we must get rid of the child node **V**. Let **U** be the parent node of **V** and after the deletion of **V**, **U** loses all of its children and is not the root. Then we must delete **U** and merge the path according to propositions 1 and 2, respectively. If **U** has only one child node left, then we must delete **U** and merge the path. The example has already been stated in the definition of Proposition 2.

Case 2 (Child node V produced by slicing an existing path for '$'): Both the explicit and implicit suffix trees in Fig. 1 and Fig. 2 can be used to discuss this case. The path between node 1 and node 3 splits as a result of the inclusion of "$". Then node 4 is constructed for "$" and a new node 2 is added between them. In order to reverse this situation, we will first delete node **V** and then rejoin the split path by adhering to proposition 2. In this example, we will first remove node 4, then remove node 2, and then merge the paths from node 1 to node 2 and from node 2 to node 3. If any suffix links were heading to node 2 instead of root, we would also have redirected those links.
We will now discuss the second stage, which is how to locate an active point for a fresh pass. The entire procedure and justification can be given as follows.

1. We transform explicit suffixes into their implicit form in step 1. Thus, we are able to determine how many suffixes have been transformed. The greatest implicit suffix at the time of conversion is indicated by this value. Let us say the number is **L**. Due to the fact that a suffix becomes implicit along with all of its smaller sub-suffixes, all of the suffixes that were present at least a **L** distance from the end of the sequence will now be implicit. Furthermore, suffix deletion happens in a specific order; larger suffixes are deleted first, followed by their smaller sub-suffixes.

2. What we shall do is undo the effects in the order that the nodes were added or the pathways were split apart by the inclusion of '$'. Therefore, if we come across a node that has been changed while erasing the effects, we cease reverting because all previous effects produced by the insertion of "$" have been compromised. So, having identified the tree's greatest implicit suffix, we can now traverse the tree to determine its location and update the active point, also known as the "active Node" with "active Edge", and "active Length." While eliminating the impacts, some details can still be saved. For instance, we would reverse the effects of nodes 18, 15, and 2 in order to obtain the tree of Fig. 2 from Fig. 3. Because it is a fake node, removing the child for "$" from the root does not assist in identifying the greatest implicit suffix.

We shall do a simulation of our algorithm right now. Imagine that we have a window with the string "abcababababc" (the explicit tree for this window is shown in Fig. 2) and that we subsequently received the new symbol "b" causing our window to slide. The stages are depicted in the following figures: in Fig. 4, we show the status after the deletion of the leftmost character "a"; in Fig. 5 we present the condition after the conversion from explicit to implicit suffix tree with the largest implicit suffix "bc" and in Fig. 6, we show the resultant tree after adding the character "b" at the end of the string. We also show another iteration of this simultaneous deletion of the leftmost character "b" (Fig. 7), the corresponding implicit tree after the removal of the unique character "$" (Fig. 8) and the insertion of a new character "a" along with the unique character "$" at the end (Fig. 9).

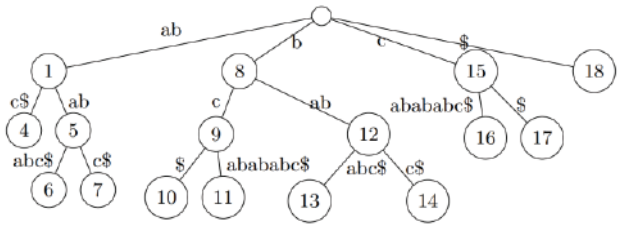

Fig. 4. After deleting the leftmost character "a" from the string.

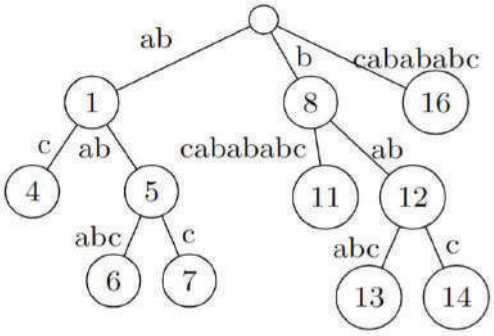

Fig. 5. Conversion from explicit to implicit after the deletion of Fig. 4.

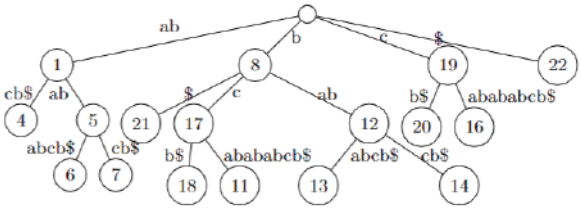

Fig. 6. After Inserting "b" at the end of the string, status after Fig. 5.

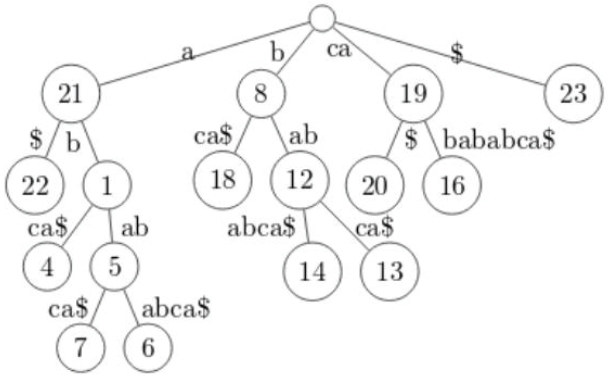

Fig. 7. After deleting the leftmost character "$" from the string.

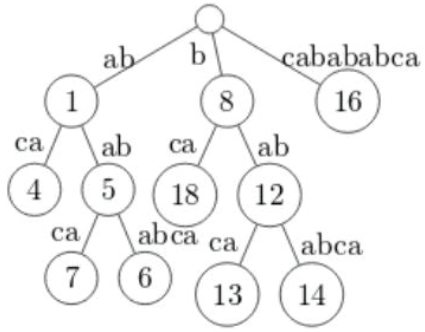

Fig 8. Conversion from explicit to implicit after the deletion of Fig. 7.

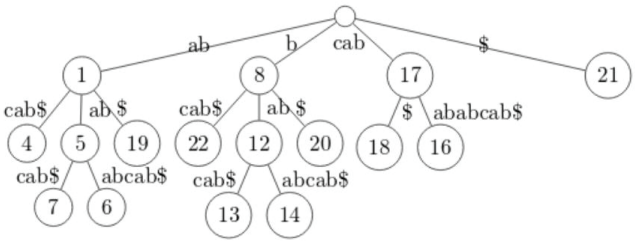

Fig. 9: After Inserting "b" at the end of the string, status after Fig. 8.

3.4.3 Pseudocodes

Now combining all the strategies, we present the pseudocode to understand the chronological steps of simultaneous insertion and deletion.

In Algorithm 1, we present the steps of our deletion module keeping the discussion aligned with the propositions and condition.

Algorithm 1 DWTS, Deletion Consistency

```
1:  procedure DeletionConsistency(Position pos)
2:      The suffix tree must be in explict form, before this operation.
3:                                  ▷ the suffix starting from position of text will be deleted
4:      Delete the suffix starting from pos by traversing from suffix tree root to V
5:      Let U denotes the parent node of V
6:      if (U has lost all of its children and is not root) then
7:          Covert U to a leaf node from an internal node
8:      else if (U has got only one child node and is not root) then
9:          Let X denotes the parent node of U and W denotes a single remaining child
    of U
10:         Merge the path (X → U) and (U → W)
11:         Any node pointing to U as its suffix link will point to the root now
12:     Any node pointing to V as its suffix link will point to root now.
13:     RETURN
```

We maintained a similar set of variables used in Proposition 2 to understand the logical fragments. This function updates the nodes in such a regard so that they can be consistent with further insertion or deletion operations simultaneously. To delete the leftmost character of the given string, this function is called with the desired indexing. Though the asymptotic complexity of the function is O(N) where N denotes the number of nodes in the tree, the actual complexity is much lesser, as we always travel the path of the largest suffix only to delete it.

Algorithm 2 DWTS, Insertion Consistency

```
1:  procedure InsertionConsistency( void )
2:      implicit ← 0
3:      if the tree is in implicit form then Return
4:      for each node V created due to '$',traverse in reverse order of their creation do
5:          if (V is not deleted ) then
6:              if (V is Case 1 type of node) then
7:                  Delete the child for '$' from V
8:                  if (V has lost all its children and is not root) then
9:                      Convert V to leaf
10:                     Redirect suffix links to root that were pointed to V
11:                     Break
12:                 Increment implicit by 1
13:             else if (V is Case 2 type of node) then
14:                 Delete the child for '$' from V
15:                 Merge the splitted path with parent of V
16:                 Increment implicit by 1
17:                 Delete V
18:                 Redirect suffix links to root that were pointed to V
19:         else if (V is deleted ) then Break
20:     remainingSuffixCount ← (implicit − 1)
21:     Remove '$' from text
22:     Remove '$' from all the existing suffixes of the tree using global reference
23:     Traverse tree to find the largest implicit suffix and update activePoint
24:     RETURN
```

Similarly, in Algorithm 2, we present the logical statements that maintain consistency in the suffix tree. For ease of discussion, we used similar variables used in the discussion of the cases in section 3.4.2. Here, we mainly simulate two Cases and finally update the active point by traversing the current largest implicit suffix. Based on this point, Ukkonen's algorithm starts trying to put a suffix in the tree by ensuring the maximization of shared suffixes. This function is called once before inserting one or more characters in the tree to convert the tree from explicit to its implicit form and to update the active point for the next phase. The asymptotic complexity of the above function is also O(N) where N denotes the number of nodes in the tree. The actual complexity is much lesser, as we only traverse the leaf nodes to remove its effect and to find the largest implicit suffix, we need to traverse only a single path guided by the value of the *implicit* variable over the given input time series string.

3.5 Maximum Possible Weighted Support Pruning, MPWS Pruning

It is not possible to check each pattern to see if it is a weighted frequent (or weighted periodic) pattern. The Downward Closure Property (DCP) is employed in the unweighted variant of pattern mining. The most popular strategy is to use the weight of the largest weighted character (MaxW) of the database to decrease the number of patterns tested because trivial DCP does not work in weighted pattern mining. When a pattern is being tested, its potential as a candidate pattern is being assessed. We recommend the MPWS Pruning approach since it consistently outperforms MaxW Pruning. We start by providing some definitions.
In this section, we 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) stands for the total number of characters from the root to node N. sumW(14) in Fig. 2 tree represents the sum of the weight of the characters "b", "a", "b", and "c", which is 1.2.

Definition 2. (weight (X)):

weight(X) denotes the average weight of all the characters of pattern X. For example If X is "abac" then weight(X) is (0.8+0.1+0.8+0.24)/4=0.475.

Definition 3. (minsup and **σ):**

minsup denotes a real number between 0 and 100. Let, the maximum weight of a character observed in the sequence is maxW. So, we can safely express that, no pattern can have a weighted support of more than equation 1. Here |S| denotes the length of the sequence.

$$\boldsymbol{\sigma} = minsup \times \frac{maxW \times sizeV(N)}{100} \quad \text{..... (1)}$$

Definition 4. (weightedSupport(X)):

A pattern X's weightedSupport is denoted by the multiplication of weight(X) and support(X). Here support(X) denotes the actual periodicity of X. A pattern X is weighted periodic if weightedSupport(X) ≥ $\boldsymbol{\sigma}$.

Definition 5. (cnt(A, B)):

The total number of characters found on the path between node A to node B is denoted by cnt(A,B). In Fig. 2, the value of cnt(8, 13) is 6.

Definition 6. (maxW(A, B)):

The weight of the maximum weighted character on the path between node A to node B is denoted by maxW(A, B). In Fig. 2, the value of maxW(8, 13) is 0.8.

Definition 7. (sizeV(N))):

The size of the occurrence vector of node N is represented by sizeV(N). This presents the number of occurrences of the patterns ending at node N.

Definition 8. (subStr(A,B)):

subStr(A, B) denotes the substring of the time series found in the path between nodes A and B. In Fig. 2, the value of subStr(8, 13) is "ababc$".

Definition 9. (nodeW(N)):

Let node P be the parent of node N, E presents the edge between nodes P and N and R denotes the root node of the suffix tree. Then, we can establish the following argument,

$$A = \frac{sumW(P) + maxW(P,N)}{cnt(R,P) + 1} \quad \text{..... (2)}$$

$$B = \frac{sumW(P) + maxW(P,N) \times cnt(P,N)}{cnt(R,P) + cnt(P,N)} \quad \text{..... (3)}$$

$$nodeW(N) = max(A,B) \times sizeV(N) \quad \text{..... (4)}$$

Now, we present a Lemma based on nodeW(N). Let us assume, s1 = subStr(R, P), s2=subStr(P, N) and s3 be any nonempty prefix of s2 and we consider a string s = s1+s3. Then, we can write,

Lemma 1. *nodeW(N) ≥ weightedSupport(S)*

Proof: As per definition 4, weightedSupport(S) = weight(S) × support(S). max(A, B) is the maximum possible value of weight S under any scenario which can be measured by equations 1 and 2 respectively. Now, there can be two possible cases,

1. Case 1. (weight(s1) > maxW(P, N)): In the first case, even if all the characters in E have the same weight as maxW(P, N), the weight of S will never be greater than A as shown in equation 2. Because, if we increase the length of s3 by a single character weight(S) will decrease. Here, A denotes the maximum value possible for weight(X).
2. Case 2. (weight(s1) < maxW(P, N)): We can measure an upper bound for weight(S). Let us consider all the characters in E has a weight equal to the maxW(P, N). Then with the gradual increase in length in s3, the weight of S starts to decay. We can calculate the value of B using equation 3 by estimating that s3 has the maximum possible length.

3. Case 3. (weight(s) = maxW(P, N)): In this particular scenario, the length of s3 does not bear any significance. So, in summary,

→ max(A, B) ≥ weight(S)
→ sizeV(N) ≥ support(S)
→ max(A, B) × sizeV(N) ≥ weight(S) × support(S)
∴ nodeW(N) ≥ weightedSupport(S)

Definition 10. (MPWS(N)):
MPWS(N) represents the maximum value of nodeW among all the nodes in the subtree of N including N.
As stated in definition 9, NodeW(N) expresses the maximum possible weighted support that can be achieved in a pattern S. MPWS holds the maximum value of nodeW found in the subtree of N, it represents the maximum possible weighted support achievable by any pattern that has prefix as s1. The definition of s1 is stated in Lemma 1's proof.
The **candidate generation** process is performed using a breadth-first search (BFS) in the suffix tree using a level-by-level pattern generation. In the breadth-first search, once we reach node N, for every pattern S if we observe that (weight(S) × sizeV(N)) ≥ **σ**, we consider S among the candidate patterns.
There will be about N nodes in the suffix tree for a string of length L. However, the total number of characters on the edges can approach L^2. As a result, the dataset may contain L^2 possible patterns.
Every pattern is tested as part of the aforementioned candidate Generation process, and if it succeeds, it becomes a candidate. However, examining each pattern takes time. Therefore, we need to come up with a pruning condition that lowers the number of patterns that are verified.
The most popular method is to use the database's maximum weighted character (MaxW) weight. Any super pattern of P cannot be a weighted frequent pattern if MaxW × support(P) < **σ**. Therefore, those patterns cannot also be periodic patterns.

Lemma2: *If MPWS(C) < **σ** for any child C of node N, we can disregard the entire subtree of C and proceed with other branches of the tree.*

The proof is very trivial because, by the definition of MPWS stated in Definition 11, any node U in the subtree of C will not satisfy nodeW(U) ≥ **σ**.

All of the candidate patterns are actually frequent weighted subsequences of the current time series. Using well-known periodicity detection algorithms, we can test the occurrence vector of each potential pattern with various period values to see if they are also periodic patterns [5].

To discuss our proposed pruning measure, we present Fig. 10 and Table 1. Fig. 10 represents the suffix tree of the string "abcababababc$". For visualization as minsup value, 10% was chosen. Table 1 holds the detailed calculation of MPWS with other associated values.

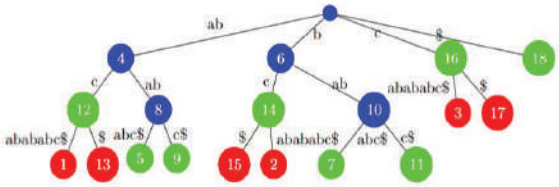

Fig. 10. An Example of MPWS Pruning.

To explain the colored nodes of Fig. 10, we note the following points,
1. Any pattern that has a blue node in its underlying subtree is tested. Here, patterns "a", "ab", "aba", "abab", "b", "ba" and "bab" are tested for candidacy.
2. Any green node denotes that, the complete subtree along with this node can be ignored during the candidate generation step.
3. Red node denotes that, these nodes are never evaluated. All the red nodes have green nodes in their ancestors. When a node is detected as its subtree should be ignored, it is colored as green and all of its underlying subtree is colored as red.

Table 1: Calculation of the Necessary Values for MPWS Pruning of Fig. 10

N	*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*	*0.5*	*0.73*	*0.73*	*0.73*
4	*4*	*0.8*	*0.8*	*3.2*	*3.2*
5	*1*	*0.52*	*0.62*	*0.62*	*0.62*
6	*4*	*0.1*	*0.1*	*0.4*	*1.13*
7	*1*	*0.45*	*0.6*	*0.6*	*0.60*
8	*2*	*0.57*	*0.62*	*1.25*	*1.25*
9	*1*	*0.4*	*0.37*	*0.4*	*0.4*
10	*2*	*0.45*	*0.57*	*1.13*	*1.13*
11	*1*	*0.3*	*0.28*	*0.3*	*0.3*
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*	*0.3*	*0.67*
15	*1*	*0.10*	*0.10*	*0.10*	*0.10*
16	*2*	*0.2*	*0.2*	*0.4*	*0.73*
17	*1*	*0.1*	*0.1*	*0.1*	*0.1*
18	*1*	*0.00*	*0.00*	*0.00*	*0.00*

In this illustration, MPWS Pruning states that only 7 patterns are evaluated for candidacy. Five of these ultimately develop into candidate patterns. If we did not utilize any pruning, we have to test 50 patterns in total. If we used only MaxW Pruning, we had to test 10 different patterns.

Additional Difficulty in Pruning: The maximum number of nodes in a suffix tree that we construct for a string of length L is 2 × L. We first calculate the MPWS value for each node during candidate generation, which can be done by a depth-first traversal on the tree. For that traversal, we need the MaxW value for each edge. We calculate it by applying a range minimum query strategy to the static data. Given that each edge's query complexity is O(1), the complexity added by MPWS pruning is O(L).

4. Experimental Results

To contrast our strategy with the existing strategies, we employed a number of data sets from the UCI Machine Learning Repository [10]. We present the outcomes of the following four data sets because they are all comparable in terms of their findings. To create a string of characters, the datasets were discretized. The complexity to discretize the dataset was O(N) where N denotes the total number of elements in the dataset.

1. Individual household electric power consumption Data Set
2. Absenteeism at Work Data Set
3. Appliances energy prediction Data Set
4. Diabetes Data Set

To generate weights of the items we drew the weight values from a normal distribution over the frequency of the items keeping the mean and standard deviation to 0 and 0.1 respectively.

4.1 Runtime Performance of DTSW

Existing time series research offers no guidance on how to approach sliding window-based issues. For each window, the data structure must be created from scratch. But this strategy is ineffective. We will present a contrast of the experimental findings between DTSW and tree reconstruction for each window. When the window size or number is enormous, building the tree from scratch for each new window performs poorly. DTSW is highly helpful in those situations.

For four distinct window widths, we have four graphs in Fig. 11, where the x-axis represents the number of windows traversed and the y-axis represents the total time taken from the start. It is clear that as window size increases, reconstruction performance degrades, but DTSW continues to function consistently with low runtime costs.

Data from the Individual Household Electric Power Consumption Data Set was used to create the graphs. Similar results are observed in the graphs for the other three datasets, so we have omitted to present them here to avoid repetition.

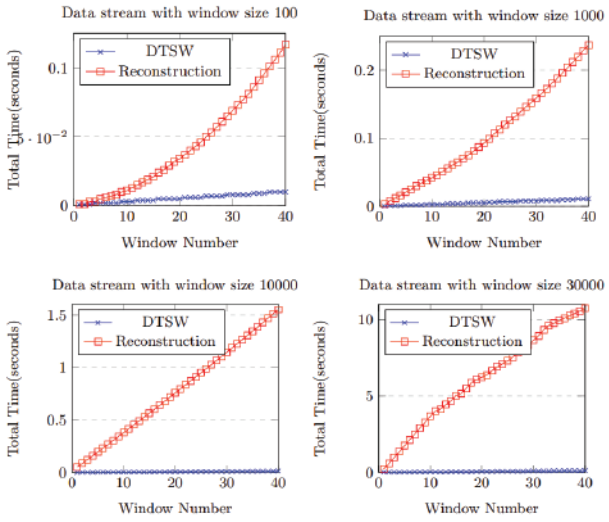

Fig. 11. Sliding Window with Window Size 100, 1000, 10000 and 30000

An important novelty of DTSW is, it is applicable to any time series pattern mining problem that uses a suffix tree as the underlying representative data structure and works in the data stream or sliding window alike environments. It is quite generic because it only focuses on how to efficiently update the data structure so that based on Ukkonen's algorithm simultaneous insertion (at the end) and deletion (from the start) can be done. As an example, we can cite [29], where the authors use our proposed strategy to update the data structure rather than reconstruction. This also supports our statement that, our proposed solution is quite flexible to be bundled within other problems.

Analytically, we can also discuss how reconstruction operation will always be costlier compared to tree modification. In the tree reconstruction, we again need to scan and generate a set of nodes to represent all the suffixes, whereas in the tree modification we need to revert a subset of such nodes. So, intuitively, though both approaches need a linear time complexity but the tree modification approach will require lesser number of operations resulting in improved runtime.

4.2 Pruning Performance of MPWS

We measure the performance of MPWS Pruning strategy over various metrics. In this section, we present our findings in brief.

4.2.1 Number of Patterns Tested with Varying minsup

Every pattern must be examined in a candidate generation process without pruning to see if it qualifies as a candidate. However, this is unacceptable because there may be numerous pointless patterns. Avoiding testing patterns that won't eventually become candidate patterns was our key objective when developing MPWS Pruning.

We discretized all of the datasets and gave each unique character a weight that fits a normal distribution ($\mu = 0.5$ and $\sigma = 0.2$). We have contrasted MPWS and MaxW pruning for various weighted support thresholds across all databases. According to Fig. 12-15, MPWS pruning tests a lot fewer patterns than MaxW pruning. We showed our results for each of the experimented datasets for different weighted support thresholds. The x-axis present the weighted support thresholds and the y-axis represents the number of patterns based on three criteria, the number of patterns tested by MaxW Pruning, the number of patterns tested by MPWS Pruning and the actual number of candidates. For instance, if we try to optimize the candidate generation process in the Individual Household Electric Power Consumption Data Set using simply MaxW in the database, it tests 63490 patterns whereas MPWS Pruning checks 21592 patterns only. 21408 patterns eventually turn into candidate patterns. The testing of practically all superfluous patterns is pruned by MPWS Pruning. We can observe from the findings that MaxW Pruning tests more patterns than MPWS Pruning. MPWS Pruning will not ever try more patterns than MaxW Pruning, in actuality.

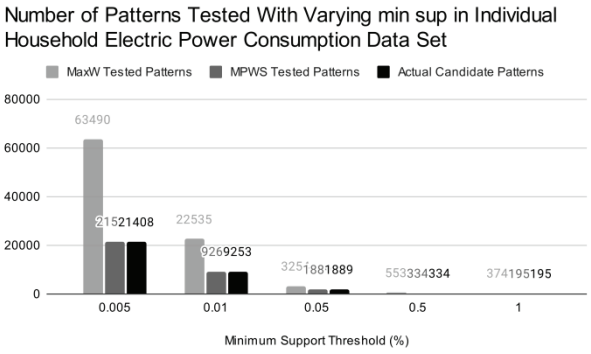

Fig. 12. Number of Patterns Tested with Varying minsup in Individual Household Electric Power Consumption Data Set.

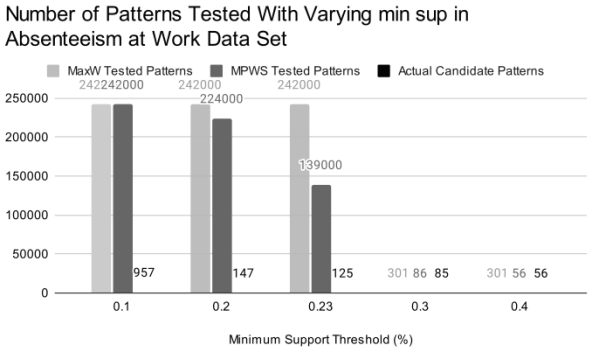

Fig. 13. Number of Patterns Tested with Varying minsup in Absenteeism at Work Data Set.

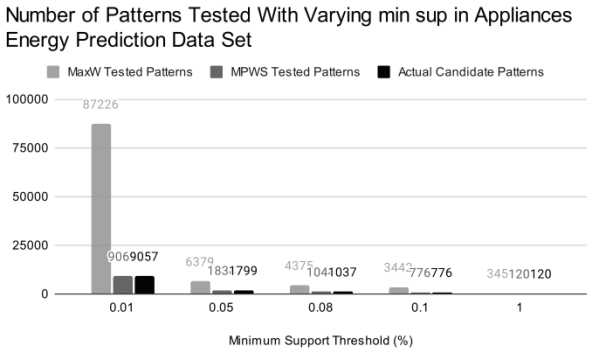

Fig. 14. Number of Patterns Tested with Varying minsup in Appliances Energy Prediction Data Set.

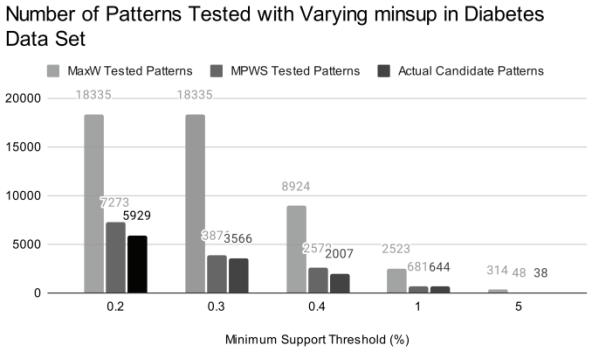

Fig. 15. Number of Patterns Tested with Varying minsup in Diabetes Data Set.

4.2.2 Runtime With Varying minsup

In lower minsup values, we need to generate more patterns. So, with the increasing minsup value, the runtime will decrease. This also becomes evident through our experimentation with various datasets.

In Fig. 16 and Fig. 17 we present the results for the individual household electric power consumption dataset and Absenteeism at Work dataset. If we analyze the charts' behavior, we can observe that, there are some in-between spikes, but overall, with the decrease in minsup value, the total runtime increases. Some anomalies in spikes can be observed due to the weight distribution and the window size.

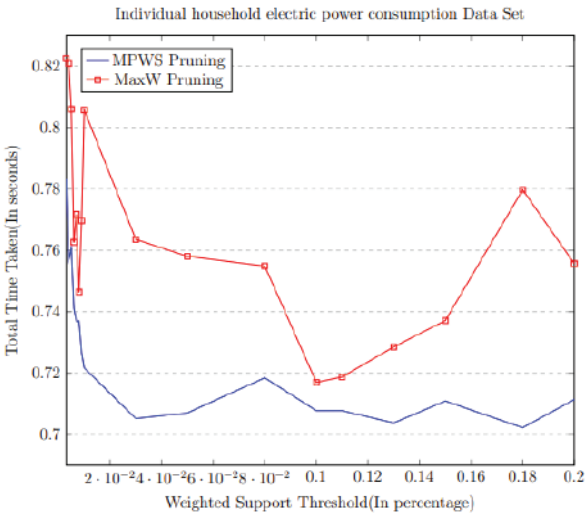

Fig. 16. Runtime with Varying minsup in Individual Household Electric Power Consumption Data Set.

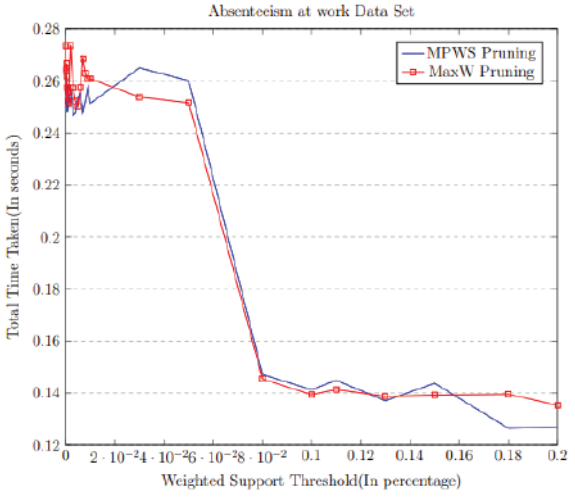

Fig. 17. Runtime with Varying minsup in Absenteeism at Work Data Set.

The underlying reasoning can also be stated numerically based on Fig. 13 over the Individual Household Electric Power Consumption Dataset. In Fig. 9-12, we have shown how the number of tested patterns fluctuates by varying minsup values. In our experiments, for this dataset, we saw that, when the minsup value is 0.005% we needed to test 21408 candidates in this dataset. But when the minsup value was increased to 0.01%, the number of tested patterns became only 9523. This variation controls the change in the total runtime. As a similar pattern was observed in other datasets, we omitted the figures here.

4.2.3 Memory Usage with varying minsup

To understand the memory usage of MPWS Pruning, we conducted experiments with MaxW Pruning. Here, for varying minsup, we record the amount of maximum memory used by the programs from start to completion.

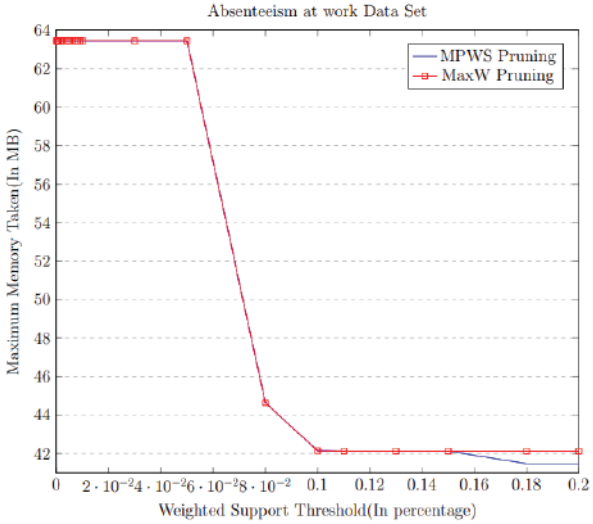

Fig. 18. Memory Usage with Varying minsup in Absenteeism at Work Data Set.

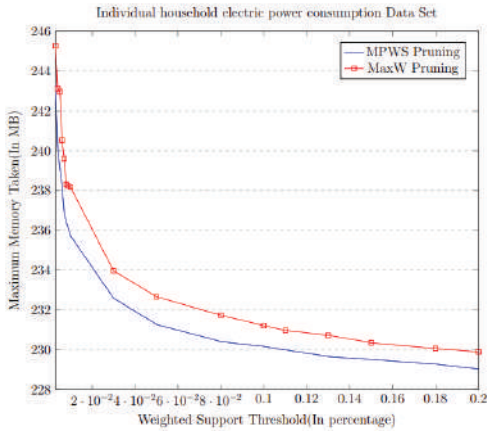

Fig. 19. Memory Usage with Varying minsup in Individual Household Electric Power Consumption Data Set.

In Fig. 18, and 19, we present the results for the Absenteeism at Work dataset and Individual Household Electric Power Consumption dataset. The other datasets exhibited similar characteristics. So, we did not show the results here. The data indicate that both methods use approximately identical amounts of memory. MPWS works slightly better than MaxW Pruning when minsup is low. The largest factor in this situation is the maximum size of the queue because we have used a breadth-first approach to construct the patterns. Increases in minsup result in fewer patterns being tried, which uses less memory. MaxW Pruning tests more patterns than MPWS Pruning. As a result, the queue requires more space, which uses more RAM.

4.2.4 Extendibility of MPWS to Other Problems

An important point to note is that MPWS is a generic pruning measure to reduce the number of candidate patterns that can be applied in any weighted time series pattern mining. Periodical time series pattern mining is a very common concept in the addressed domain. During designing MPWS,

any such constraint has not been imposed, so MPWS can easily be extended to weighted periodical time series pattern mining problems.

To have a comparison with recent one of studies, we have chosen a very relevant work HOVA-FPPM [30]. This study uses a hash-based data structure and Apriori based algorithm to mine periodical patterns from a time series database. To make a fairground comparison we first apply weights over the items of the database drawing and assigning the values from normal distribution. Then we apply HOVA-FPPM algorithm with MaxW measure and our suffix tree-based solution with MPWS measure and track the performance. HOVA-FPPM algorithm was not specifically designed to handle weights, so we brought the traditional strategies applied to MaxW measure into HOVA-FPPM to mine all periodical patterns.

In Fig. 20. We present the result found from the individual household electric power consumption dataset for various minimum support thresholds. We can easily see that, MPWS measure has considered lesser number of patterns than MaxW embedded HOVA-FPPM algorithm. As the underlying strategy we can easily state the tighter approximation property of MPWS measure.

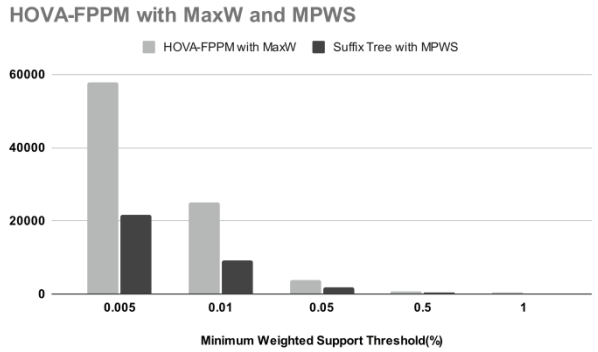

Fig. 20. Number of Patterns tested in HOVA-FPPM with MaxW and Suffix Tree with MPWS in weighted time series pattern mining.

4.3 Feasibility Analysis of the Framework

To determine whether DWTS is feasible, it is sufficient to be able to construct the suffix tree for the window currently under examination. A suffix tree uses only O(L) memory, where L is the window's length. This is obviously preferable to creating the entire tree from scratch for each window, and memory use of DWTS is also O(N), as the complexity of deleting the first P characters is O(P) and inserting P new characters at the end is also O(P). Therefore, neither runtime nor memory is growing exponentially.

With the help of a preliminary computation, MPWS prunes as many subtrees as possible. A depth-first search is necessary to determine the values required for the pruning. A query for the maximum weight on some edge is made in the depth-first search. The complexity of each response is O(1). For practical use of the entire structure, the update of the data structure for edge weight queries only requires O(log(L)) time.

4.4 Summary

The discussion of the entire section leads to the conclusion that DTWS consistently outperforms the reconstruction approach. DWTS can be helpful, particularly when mining significant patterns from time series with large window sizes and frequent alterations.

Weighted pattern mining lacks the trivial downward closure property. Candidate generation is optimized with MaxW Pruning. Our suggested MPWS Pruning technique, which is also time and memory efficient, tests fewer patterns overall than MaxW Pruning.

5. Conclusion

In this study, we address two time series pattern mining-related problems and provide our algorithms. Our first contribution, DTSW algorithm, addresses dynamic suffix tree handling issues and our second contribution, MPWS pruning strategy, provides a new measure to approximate the maximum possible weighted support for an extended pattern which can be used as a downward closure property to reduce the number of candidates during weighted time series pattern mining. Both of our contributions are independent in manner and can be used separately in different relevant problems. DTSW's dynamically updating strategies are applicable for both weighted and unweighted frameworks and can be adapted to run-time dynamic window sizes. Thus, this method can be used to address numerous problems that deal with dynamical suffix tree updates focusing on data stream behavior. MPWS measure, due to its distinct method of approximating the upper bound, can be employed in various forms of weighted pattern mining-related problems in place of conventional MaxW measure which may reduce the number of generated candidates resulting in improved mining runtime. Through analytically analyzing and experimenting with various real-life datasets, we also showed the efficiency and extendibility of our proposals with respect to the recent relevant works. As an ongoing work, we have plans to investigate our proposed strategies' performance and applicability in other time series pattern mining-related problems. We also intend to introduce the idea of dynamic weights in time series weighted pattern mining and examine the performance of our proposals in the future.

Acknowledgements

We would like to thank all the reviewers for their valuable time and suggestions to improve this article. We sincerely believe that the quality of this work has reached a new peak due to their valuable efforts.

References

1. C. K. S. Leung and Q. I. Khan, "DSTree: a tree structure for the mining of frequent sets from data streams". In Sixth International Conference on Data Mining (ICDM'06) (pp. 928-932). IEEE, 2006.
2. N. Almusallam, Z. Tari, J. Chan and A. AlHarthi, "Ufssf-an efficient unsupervised feature selection for streaming features". In *Advances in Knowledge Discovery and Data Mining: 22nd*

Pacific-Asia Conference, PAKDD 2018, Melbourne, VIC, Australia, June 3-6, 2018, Proceedings, Part II 22 (pp. 495-507). Springer International Publishing, 2018.

3. X. B. Yan, Z. Wang, S. H. Yu and Y. J. Li, "Time series forecasting with RBF neural network". In 2005 International Conference on Machine Learning and Cybernetics (Vol. 8, pp. 4680-4683). IEEE, 2005.
4. C. F. Ahmed, S. K. Tanbeer, B. S. Jeong and Y. K. Lee, "Handling dynamic weights in weighted frequent pattern mining". *IEICE TRANSACTIONS on Information and Systems*, *91*(11), 2578-2588, 2008.
5. F. Rasheed, M. Alshalalfa, and R. Alhajj, "Efficient periodicity mining in time series databases using suffix trees", *IEEE Transactions on Knowledge and Data Engineering*, *23*(1), 79-94, 2010.
6. A. K. Chanda, S. Saha, M. A. Nishi, M. Samiullah, and C. F. Ahmed, "An efficient approach to mine flexible periodic patterns in time series databases", *Engineering Applications of Artificial Intelligence*, *44*, 46-63, 2015.
7. C. F. Ahmed, S. K. Tanbeer, B. S. Jeong, and Y. K. Lee, "An efficient algorithm for sliding window-based weighted frequent pattern mining over data streams", *IEICE TRANSACTIONS on Information and Systems*, *92*(7), 1369-1381, 2009.
8. E. Ukkonen, On-line construction of suffix trees. *Algorithmica*, *14*(3), 249-260, 1995.
9. A. K. Chanda, C. F. Ahmed, M. Samiullah, and C. K. Leung, "A new framework for mining weighted periodic patterns in time series databases", *Expert Systems with Applications*, *79*, 207-224, 2017.
10. E. D. Dua, and T. Karra, "*UCI (University of California Irvine) Machine Learning Repository," Repository, 2017*
11. Z. Tao, Q. Xu, X. Liu, and J. Liu, "An integrated approach implementing sliding window and DTW distance for time series forecasting tasks", Applied Intelligence, 1-12, 2023.
12. Z. Wang, Y. Wang, C. Gao, et al., "An adaptive sliding window for anomaly detection of time series in wireless sensor networks", Wireless Network 28, 393–411, 2022. https://doi.org/10.1007/s11276-021-02852-3
13. S. Li, and P. Shang, "Characterizing Nonlinear Time Series via Sliding-Window Amplitude-Based Dispersion Entropy", Fluctuation and Noise Letters, 2350023, 2023.
14. H. Zhang, H. Wang, Y. Yan, et al., "Weighted dynamic transfer network and spectral entropy for weak nonlinear time series detection", Nonlinear Dyn 111, 9345–9359, 2023. https://doi.org/10.1007/s11071-023-08310-3
15. P. Braun, A. Cuzzocrea, C. K. Leung, A. G. Pazdor, and J. Souza, "Item-centric mining of frequent patterns from big uncertain data". Procedia Computer Science, 126, 1875-1884, 2018.
16. C. K. Leung, C. S. Hoi, A. G. Pazdor, B. H. Wodi, and A. Cuzzocrea, "Privacy-preserving frequent pattern mining from big uncertain data". In 2018 IEEE international conference on big data (big data) (pp. 5101-5110). IEEE, 2018.
17. C. K. Leung, H. Zhang, J. Souza, and W. Lee, "Scalable vertical mining for big data analytics of frequent itemsets". In Database and Expert Systems Applications: 29th International Conference, DEXA 2018, Regensburg, Germany, September 3–6, 2018, Proceedings, Part I 29 (pp. 3-17). Springer International Publishing.
18. A. Mantuan, and L. Fernandes, "Spatial contextualization for closed itemset mining". In 2018 IEEE International Conference on Data Mining (ICDM) (pp. 1176-1181). IEEE, 2018.
19. H. Phan, and B. Le, "A novel parallel algorithm for frequent itemsets mining in large transactional databases". In Advances in Data Mining. Applications and Theoretical Aspects: 18th Industrial Conference, ICDM 2018, New York, NY, USA, July 11-12, 2018, Proceedings 18 (pp. 272-287). Springer International Publishing.
20. K. Sharma, and E. E. Lulandala, "OTT platforms resilience to COVID-19–a study of business strategies and consumer media consumption in India", International Journal of Organizational Analysis, 31(1), 63-90, 2023.
21. R.A Rizvee, M.S.H. Shahin, C.F. Ahmed, C.K. Leung, D. Deng, J.J. Mai, "Sliding window based weighted periodic pattern mining over time series data". In: ICDM 2019, pp. 118-132 (2019)
22. R.A. Rizvee, M.F. Arefin, C.F. Ahmed, "Tree-Miner: Mining Sequential Patterns from SP-Tree". In: Lauw, H., Wong, RW., Ntoulas, A., Lim, EP., Ng, SK., Pan, S. (eds) Advances in Knowledge Discovery and Data Mining. PAKDD 2020. Lecture Notes in Computer Science(), vol 12085. Springer, Cham. https://doi.org/10.1007/978-3-030-47436-2_4
23. X. Wang, J. Lin, P. Senin, T. Oates, S. Gandhi, A. P. Boedihardjo, and S. Frankenstein, "RPM: Representative Pattern Mining for Efficient Time Series Classification". In EDBT (pp. 185-196), 2016.
24. M. A. Nishi, C. F. Ahmed, M. Samiullah, and B. S. Jeong, "Effective periodic pattern mining in time series databases". Expert Systems with Applications, 40(8), 3015-3027, 2013.
25. M. Karaca, M. M. Alvarado, M. R. Gahrooei, A. Bihorac, and P. M. Pardalos, "Frequent pattern mining from multivariate time series data". Expert Systems with Applications, 194, 116435, 2022.
26. Y. S. Jeong, M. K. Jeong, and O. A. Omitaomu, "Weighted dynamic time warping for time series classification", Pattern recognition, 44(9), 2231-2240, 2011.
27. K. Mishra, S. Basu, and U. Maulik, "Graft: A graph based time series data mining framework", Engineering Applications of Artificial Intelligence, 110, 104695, 2022.
28. Y. Xun, L. Wang, H. Yang, and J. Cai, "Mining relevant partial periodic pattern of multi-source time series data". Information Sciences, 615, 638-656, 2022.
29. E. W. Madill, C. K. Leung, and J. M. Gouge, "Enhanced sliding window-based periodic pattern mining from dynamic streams." International Conference on Big Data Analytics and Knowledge Discovery. Cham: Springer International Publishing, 2022.
30. J. Qu, et al, "Time series forecasting method based on frequent pattern mining." Journal of Physics: Conference Series. Vol. 1682. No. 1. IOP Publishing, 2020.