New Approaches to Mine Weighted Periodic Patterns From Dynamic Time Series Databases

by

Redwan Ahmed Rizvee
Exam Roll: Curzon Hall-203
Registration No: 2014-716-612
Session: 2014-15

and

Md. Shahadat Hossain Shahin
Exam Roll: Curzon Hall-213
Registration No: 2014-916-638
Session: 2014-15

A project submitted in partial fulfilment of the requirements for the degree of Bachelor of Science in Computer Science and Engineering

DEPARTMENT OF COMPUTER SCIENCE AND ENGINEERING
UNIVERSITY OF DHAKA

January 26, 2022

Declaration

We, Redwan Ahmed Rizvee and Md. Shahadat Hossain Shahin, hereby, declare that the work presented in this project is the outcome of the investigation performed by us under the supervision of Dr. Chowdhury Farhan Ahmed, Professor, Department of Computer Science and Engineering, University of Dhaka. We also declare that no part of this project has been or is being submitted elsewhere for the award of any degree or diploma.

Countersigned

.....

(Dr. Chowdhury Farhan Ahmed)

Supervisor

Signature

.....

(Redwan Ahmed Rizvee)

.....

(Md. Shahadat Hossain Shahin)

"It has been said that something as small as the flutter of a butterfly's wing can ultimately cause a typhoon halfway around the world."

Chaos Theory

Abstract

Handling a dynamic database has always been a crucial problem in data mining which has led to several types of research in the field of time series mining. Existing literature enforce on the reconstruction of the underlying structure (*Suffix Tree*) for each new window in time series data stream. But this reconstruction performs poorly when the window size is large or when the database is frequently modified. Hence, in this study, we are proposing a solution which dynamically updates the structure rather than reconstruction for each modification. Moreover, we also propose another solution to address the problem associated with mining *weighted periodic patterns* from time series data streams. Existing literature mostly rely on Maximum Weight in the database as a downward closure property for pruning infrequent candidates. However, these works 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 dynamicity and reducing the number of patterns tested.

Acknowledgements

All praise is to the Almighty, who is the most gracious and most merciful. There is no power and no strength except with him.

Our deep gratitude goes to our thesis supervisor, Dr. Chowdhury Farhan Ahmed, Professor, Department of Computer Science and Engineering, University of Dhaka, for his proper guidance. He has shared his expert knowledge and experience gathered from working in this field over an extensive period and has been an immense support and guide throughout our work by keeping regular updates, introducing us to various trending problems, providing helpful directions to think and always being the best mental support to urge us to do something good. He has always been the best admirer and scrutinizer of our work at the same time. Authors express their whole hearted thankfulness for such profound endeavour.

We want to express our thankfulness towards our families and friends for their support and suggestions. They define who we are, what we are and what we think. We specially want to thank Nahian Ashraf, Riddho Ridwanul Haque and Fariha Hossain for providing valuable suggestions and comments throughout the work. We also want to thank Ashis Kumar, Sabrina Zaman Ishita and Md. Fahim Arefin for providing us various resources essential for our work.

Lastly, we want to thank Department of Computer Science and Engineering, University of Dhaka, its faculty, staff, and all other individuals related to the department for providing us with an opportunity to be a part of this family and facilitating us with a good educational and research environment. Finally, we want to thank Batch 21 for being an inseparable part of our journey.

Redwan Ahmed Rizvee
Md. Shahadat Hossain Shahin
02 January, 2019

Contents

List of Figures

List of Tables

List of Algorithms

Chapter 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 [2], [3], [4] and flooded a huge set of new domains. Sequential pattern mining [5] is one of the trending domains in the field of pattern mining and time series pattern mining is a very renowned and widely discussed topic under sequential pattern mining.

A good amount of research works have been done regarding time series and it has always been an area of interest to find an efficient data structure to represent time series databases. Suffix tree has been the most useful among the data structures which was first proposed in [11].

Another very important field in pattern mining is dynamic databases where continuous insertion and deletion of new events take place. These modifications force updates in the existing structure to reflect the modified database. One way is to modify the data in the currently built structure, another is to build a new structure from scratch. A similar kind of work is done in [12] which depicts the idea of modifying the underlying tree for new batches or upcoming item sets. The core of time series pattern mining is data stream. The main feature of Data Streams is they are continuous and unbounded [12]. So it is a very vital issue about how to change the data structure in the eve of modification of previous data stream.

Another class of modification of data stream also includes the sliding window problem [13]. This has also created an area of discussion in time series about how to change the current structure when the window slides. [7], [28], [29] are some of the examples of works related to sliding window in various domains. But in time series mining, this precious group of work has not flourished well and is waiting for a good amount of contribution. One of our proposed approaches enlightens on this valuable section.

Introduction of weight to patterns helps to find patterns having some specific features [13], [14], [19], [20], [22]. Specially in dense databases, weighted pattern mining can be very helpful to discover interesting characteristics. In real life, all the items do not carry same kind of importance and only frequency measure is not enough to represent all types of features. Sometimes less frequent item sets can be much more interesting than frequent items, and so to discover various types of interesting features usage of weighted items can be very helpful and this idea of weight also plays a very significant role in our work domain, time series domain. Weighted periodical pattern mining in time series can be very helpful to find interesting periodical patterns from data streams [19]. But a very important concern in weighted pattern mining is the efficient usage of Downward Closure Property to speed up the candidate generation process. Inefficient generation of candidate patterns degrade performance to a good extent and so, this has always been an important concern to reduce the unnecessary tests and that's where downward closure property plays its role. But the main issue is appliance of this property in weighted framework is not as trivial as its unweighted framework and rather difficult. Main idea of downward closure property was first introduced in [2] but this concept does not fit directly in weighted framework. Because, first idea was proposed in unweighted framework which said that if a pattern is not frequent then any of its super pattern will never be frequent but this idea is directly contradicted by its weighted counterpart. So, to apply this property in weighted counterpart many ideas were introduced. Most popular and general

idea in this regard is using *Max Weight* and most of the weighted pattern mining problems are solved by using this in existing literature [14, 19]. But using this idea of max weight still generates a huge number of redundant candidates. Our second contribution focuses on this matter. In this study, we propose an efficient pruning approach to reduce the number of patterns to be tested for candidacy in weighted time series pattern mining.

1.1 Motivation

In our work, we present two specific problems. This subsection will explain the problems with some brief examples and will give an insight into our contribution.

Suppose, we have a shop. In this shop, the shopkeeper maintains a specific rule to decide policies for his business, at the end of each day he takes records of last thirty days sale and then analyses them to take future decisions. An interesting fact to observe here is that after a certain period of time some records get obsolete. Like in this example while analyzing, he only needs the latest thirty days transaction records and all the other previous transaction records become unnecessary for him. This is called a problem of *Dynamic Database* and to be more specific, this problem is known as Sliding Window Problem which covers a great amount of discussion in data mining. But, in this example, the main problem actually becomes how to handle the underlying structure on the eve of the dynamic behavior of the database. One of our contributions in this study solves this challenge of dynamicity efficiently in time series data streams where we propose an algorithm which helps to update the structure dynamically rather than reconstruction for each modification which provides efficiency in case of frequent modification of the database and large window size. There will be detailed discussion on these terminologies in Chapter 2.

Before giving an insight about the second contribution, we will give a short example. Our second contribution is directly connected to weighted pattern mining in time series. Now, suppose our shopkeeper does not want to analysis all the last

thirty days records rather only want to focus the transactions of holidays among the last thirty days. The idea of assigning weight can be very helpful in this regard because by giving extra weight to the transactions, he can perform weighted mining and discover these interesting features. So, weighted mining can be very useful to discover various types of characteristics from a database and especially from a dense database, because there exists a huge amount of possible candidates. Weighted mining can be very helpful to discard many unnecessary patterns also. Our working domain is time series and weighted mining can be very helpful to discover interesting facts from time series data streams. But a vital issue is the pruning concept in weighted mining which is known as downward closure property is not trivial and the approaches currently exist still test a huge number of patterns for candidacy. In this study, we are proposing an efficient pruning approach which targets to reduce the number of possible patterns to test for candidacy in weighted time series pattern mining.

1.2 Contributions

We have already given some examples in the previous section to show the problems on which we have worked and also presented some hints regarding our contributions to solving these problems. The main contributions of this book can be summarized as follows.

- **Handling Dynamicity in Time Series, *HDTS*:** Current literature says that the best structure to represent time series data streams is suffix tree and in case of new update events it enforces to reconstruct the tree for each modification event. In this book, we present an efficient algorithm which dynamically updates the tree rather than reconstruction for each modification event.

- **Maximum Possible Weighted Support Pruning, *MPWS Pruning*:** In this study we present an efficient pruning technique, *MPWS Pruning*

to reduce the number of patterns tested for candidacy which outperforms current maximum weight pruning(*MPWS Pruning*) techniques in a good extent.

- **Efficient Technique to Find The Maximum Weighted Symbol:** In this book we have used RMQ in static data technique to find the maximum weighted symbol in a branch of the suffix tree efficiently. This technique will have a detailed explanation in Chapter 3.

- **Experimental Analysis over Real Life Dataset:** We have shown our proposed methodologies' performance over real-life datasets and a comparative analysis between our solution and existing solution and also proved the validity of our solution.

- **Opportunity to Use Our Solution in Different Domains:** In our book we have given a core analysis of our solution and discussed the scope of using the same solution in different domains.

1.3 Organization of the Report

Our whole book is divided into five chapters. They can be summarized as follows.
Chapter 1, consists of a brief idea about our problems along with our motivation to work in these problems and a hint to our contributions.
Chapter 2, consists of our background study along with a small discussion about the related works which goes close to our work. This chapter gives a very detailed idea about the necessary terminologies to understand our solution and also focuses on the deficiencies of the existing solutions to our problems.
Chapter 3, provides a detailed discussion about our solutions and optimizations. In this chapter, we provide our solutions along with the reasoning behind them with proper examples and proofs.
Chapter 4, consists of the experimental analysis of our solutions where we run

our solutions in various real-life datasets and present a comparative analysis between our approach and existing approaches.

Chapter 5, is the last chapter of this book, where we conclude our discussion by providing the possibilities of future expansion of our work along with the possibilities of using our solution on a different problem domain.

Chapter 2

Background Study and Related Work

2.1 Introduction

This chapter will have discussion regarding our background study and other works related to our domain weighted dynamic time series. We will also discuss about the challenges of our work, problems in existing solutions and will introduce the terminologies and the concepts. All of these will give the idea about our contribution and work. This section will work as a basis for the next upcoming sections.

2.2 Frequent Pattern Mining

2.2.1 Frequent Patterns

Frequent Patterns are patterns (e.g itemsets, subsequences or substructures) that appear frequently in a data set. In **Frequent Itemset Mining** we search for those itemsets which satisfy the minimum support threshold or support count. In **Frequent Subsequence Mining** we search for the frequent sequences which satisfy minimum support threshold or support count. Same goes for frequent substructure mining. Satisfying minimum support threshold or support count

actually expresses about the fact that these elements are frequent in our database or data set or data stream. Or in another word we can say that this minimum support threshold or support count actually says these patterns must occur at least this amount of times to be considered as frequent patterns. The most basic and fundamental frequent pattern mining algorithm is **Apriori** [2]. Another two very popular and fundamental algorithms to mine frequent sequential patterns were introduced in **GSP**[8] and **SPADE** [3].

2.2.2 Association Rules

Items or sequences aka patterns that occur together can display some common characteristics among them. Association rules is one way to represent these relations. Let I = $\{I_1, I_2, ..., I_m\}$ be the set of all items. Let T be a transaction in the database D where T $\subseteq$ I. Let A be a set of items such that A$\subseteq$T. An association rule is of the form A $\Rightarrow$ B where A$\subset$I, B$\subset$I, A $\neq\emptyset$, B $\neq\emptyset$, A$\cap B$ =$\emptyset$. For example, suppose consider we are in a camera shop where camera and other tool kits regarding camera are sold. It can be found that those who buy DSLR camera generally they also buy tripod. So an association rule can be established among them , like camera $\Rightarrow$ tripod where support is 30% and confidence 50%.

2.2.3 Support

Support count of a pattern actually says that how many times a pattern has occurred. It is an estimation of how frequently this pattern is found in the dataset.

2.2.4 Confidence

Confidence is a measure of conditional probability of two patterns or items occurring together. Mathematically speaking, suppose we have two items A and B. Now a conditional probability of occurring B where A has already occurred can

be written as $P(B|A)$ and equation stands for this relation is,

$$P(B|A) = \frac{support\_count(A \cap B)}{support\_count(A)} \tag{2.1}$$

Now, this conditional probability of occurring B where A has already occurred is defined as the confidence of B with respect to A. The rules or relations which can satisfy the minimum support count or minimum support threshold are considered to be the strong rules or relations.

2.3 Sequential Pattern Mining

2.3.1 Sequence and Sequential Database

A sequence denotes a collection or a list of objects or items in a certain or defined order. A database where each and every transaction is a collection of ordered items or events and each transaction is defined with both collection of items and order is called sequential database. A ordered sequence S is written as $<e_1, e_2, e_3, e_3, ..., e_n >$where each e_i is an event and they maintain a specific order like here event e_1 occurs before e_2, e_2 occurs before e_3 and so on. For example, Suppose < a(bcd)(fg)h >is a transaction. Events occurring in first bracket are considered as a single event. So here basically occurs four events "a", "bcd", "fg" and "h" and their order is fixed. "a" must occur before "bcd", "bcd" must occur before "fg" and so on. And within the first bracket they must also maintain their order. So, "bcd" and "bdc" are not considered as same event. An example of a sequential database is given in table 2.1.

2.3.2 Sequential Pattern Mining Algorithms

"Discovering Sequential Pattern", this data mining problem was first introduced by *R. Agrawal and R. Srikant.* Input data is a set of sequences called data sequence. A sequential pattern consists of a list of set of items. The problem was

Sequence ID	Sequence
S_1	<(Camera,Lense)(Tripod) >
S_2	<(Camera)(Bag)(Tripod) >
S_3	<(Camera)(Tripod,Battery) >
.....	.....

TABLE 2.1: A Sequence Database

to find all the sequential patterns with user specified minimum support where the support of a pattern is the percentage of data sequences or transactions which contain this pattern. An example can be given like this, suppose we have a sequence S_1 = <a(ab)ac >and another sequence S_2 = <b(ab)ac >. If we define support of a pattern as the number of occurrences in transactions and if we consider minimum support count as 2, then here two length frequent sequences are "aa","ab","ac","ba" and "bc" , all of them occur two times. So they are considered to be frequent.

In the first proposal the basic idea was very simple. But the whole procedure was very costly. So various improvisations have been done till now. Two very optimized improvisations were proposed in [9] and [10].

2.4 Time Series Pattern Mining

There are numerous fields of sequential pattern mining. We have worked on **Time Series**, a very popular topic in sequential pattern mining. Our following sections will cover the basic concepts of time series pattern mining.

2.4.1 Time Series Definition

A time series is a collection of data values gathered generally at uniform interval of time to reflect certain behavior of an entity [11]. Before analysis a time series is generally first discretized [27]. Let T = $e_1, e_2, e_3, ..., e_n$ be a time series having n events where each event e_i represents the event recorded at time instance i, time

series T may be discretized by considering m distinct ranges such that all values in the same range are represented by one unique symbol from the alphabet set $\sum$. An example can be given following.

Price	Alphabet Symbol
0 - 200	a
201 - 300	b
301 - 400	c
>400	d

TABLE 2.2: Discretization Method

In the table presented above is a measure to discretize various price value ranges. So if sales amounts are between (100 - 200) we will represent that selling amount as 'a', if the range lies between (201 - 300) we will represent that as 'b' and so on. If any days total sales amount crosses 400 then we represent that event as 'd'. Assume that this will be our discretization method for the following time series database.

Day	Total Sales Amount Per Day	Range Representing Alphabet
1	120	a
2	250	b
3	150	a
4	180	a
5	220	b
6	280	c
7	500	d
8	1000	d
9	340	c
10	100	a

TABLE 2.3: Transaction Database for Time Series

So, the data sequence which will represent our above transaction will be "abaabcddca". So, by discretization method we convert our database to a sequence which helps us to mine frequent sequences or patterns in the database using efficient algorithms. Here "abaabcddca" will be our time series data stream.

2.4.2 Periodicity

Periodicity is a very important concept in time series. Periodicity actually gives testimony that a pattern is frequent or not. Because if a pattern is occurring frequently then it will follow some kind of periodicity. Some important definitions of periodicity was discussed in [11]. They are as following.

- **Perfect Periodicity**, A pattern **X** is said to satisfy perfect periodicity in some time series **T** with period **p** if starting from the first occurrence of *X(stPos)* until the end of T, every next occurrence of **X** exists p positions away from the current occurrence of **X**. If some of the expected occurrence of **X** is missing, that denotes imperfect periodicity. As it is very rare to get perfect periodicity, the idea of ***confidence*** was introduced. A pattern supporting the confidence is considered to be a member of frequent patterns. An Example can be given like this , suppose we have a time series "ababacabacaba". Now if we consider the positions of a, then we get {0,2,4,6,8,10,12} forming a perfect periodicity where period is 2. But perfect periodicity is rare, like in the string if we consider the occurrences of "ab" we can get {0,2,6,10}. Here "ab" is also occurring frequently but not with perfect periodicity.

- **Confidence**,The confidence of a periodic pattern **X** in time series **T** is the ratio between actual number of occurrences of **X** and the expected number of occurrences of **X** in **T**. Mathematically,

$$Confidence(X, stPos, p) = \frac{actual\_periodicity(X, stPos, p)}{perfect\_periodicity(X, stPos, p)} \tag{2.2}$$

  Where actual_periodicity means the occurrence of the pattern in the time series at multiple of p distance from the start position and perfect_periodicity means the expected number of occurrences to meet perfect periodicity.

$$Perfect\_periodicity = \left\lfloor \frac{|T| - stPos + 1}{p} \right\rfloor \tag{2.3}$$

An example can be given from the string "ababacabacaba". Here for pattern "a",
actual_periodicity("a", 0, 2) = 7
perfect_periodicity("a", 0, 2) = floor((13-0+1)/2) = 7
So confidence here is 1. But for pattern "ab" actual_periodicity("ab", 0, 2) = 4 and perfect_periodicity("ab", 0, 2) = 7. So confidence("ab", 0, 2) = 4/7.

- **Symbol Periodicity**, If a symbol s appears in time series T at most of the positions specified by $stPos + i * p$ then the time series is said to have symbol periodicity in T with periodicity p and having starting position $stPos$ (*must satisfy minimum support threshold or confidence to be considered as frequent*). Here, integer i takes consecutive values starting at 0. For example, in the above example , where our time series string is "ababacabacaba", here the "a" symbol is periodic with $stPos$ as 0 and occurring at positions $\{0, 2, 4, 6, 8, 10, 12\}$.

- **Sequence Periodicity**, A time series T has sequence or partial periodicity for a pattern X if starting at position $stPos$, X occurs in T at most of the positions (*must satisfy threshold*)specified by $stPos + i * p$, where p is the period and integer i takes consecutive values starting from 0. Here, $|X| > 1$.

- **Segment Periodicity**, In sequence periodicity, if period p is equal to the length of pattern X, then time series T is said to follow segment periodicity with period p, starting position $stPos$ and the pattern is X.

- **Tolerance**, Tolerance is a very important concept in periodicity finding. Generally most of the time series in real life are not noise resilient. And because of that concept of time $tolerance(tt)$ is introduced. Tolerance actually

brings some looseness over perfect periodical positions. Formal definition can be given as, a time series T which is not necessarily noise resilient, a pattern X is periodic in a subsection $[stPos, endPos]$ of T with period p and time tolerance $tt \geq 0$, if X is found at positions $stPos, stPos + p \pm tt, stPos + 2 \times p \pm tt, ...$

In the example of our string "abababacabacabaab", if we consider the occurrences of "ab" with periodicity 2 and start position $stPos$ as 0 then in the case of not considering tolerance the valid positions are $\{0, 2, 6, 10\}$ but if we consider tolerance, $tt = 1$ then we can get $\{0, 2, 6, 10, 13\}$ as valid occurrences. Sometimes if we do not consider tolerance that may create some huge issues, because the main reason of calculating periodicity is to find if the sequence is frequent or not. If a patterns position is not considered due to a little bit difference from the actual desirable position, then the result might not reflect the actual frequent patterns.

2.4.3 Pattern Mining in Time Series

A lot of efficient algorithms have been proposed to find the frequent sequences in time series. We will have detailed discussion regarding this in upcoming sections.

2.5 Consistency over Data Stream

The idea of consistency over data stream was proposed in [12]. In real life data stream is a very popular concept. In the following two sub sections we will discuss about data stream and consistency.

2.5.1 Data Stream

Data stream mining is the process of acquiring knowledge from continuous, rapid and flowing data records. Time Series is one kind of data stream where with respect to time the length of data stream increases by giving new information.

A data stream is an ordered sequence of instances that in many applications of data stream mining can be read only once or a small number of times using limited computing and storage capabilities [28]. Basically every data stream has two basic properties [11]. They are -

- Data streams are continuous and unbounded.
- Data in the streams are not necessarily uniformly distributed; their distributions are usually changing with time.

2.5.2 Consistency

The idea of consistency was proposed in [12]. Main issue of data streams are that they are not bounded to any fixed length. So with time newer information arrive and the importance of older information decrease. An example can be given like this, suppose we are dealing with the sales of a market and we have to know the total sales information of last 50 days. So if we consider the sales of the market as a data stream, here after each day newer information of that days sale is added to the stream and and as we need the latest 50 transactions, some of the information of the data stream will become unimportant to us and we can forget those information and store information of the latest fifty days only. This is known as consistency in data stream which is defined as only by remembering the latest essential information and forgetting the older unnecessary information and by keeping the data stream always updated and compacted. As time series data is one kind of data stream, this term consistency is a very important topic here. Suppose we have a time series discretized stream of 15 days as "ababacabacabaab". Now after day 15 when we get newer discretized data of day 16 as 'c', then the first symbol or the data of first symbol becomes unnecessary and should be forgotten from the stream and newer data stream becomes "babacabacabaabc". This is called the consistency of data stream which resembles new insertion in the stream and deletion from the stream but the basic structure always remains consistent.

Or formally speaking, the data structure which represents the data stream must be consistent with new insertion and new deletion which means after modification (insertion and deletion) in the stream, the whole structure need not to be built up from scratch again rather just to perform some small changing to make it suitable to represent the modified and updated stream. In the related work section we will discuss some previous literature related to consistency. This idea of having all time a fixed length string is also known as sliding window protocol. Like in the example string we always maintain a time series string of 15 length, in sliding window protocol this is called window size of 15 length. In the example after getting the information of day 16, our window slides 1 symbol from the left side to maintain a window of length 15. Idea of sliding window was given in [12], [13] and [29].

2.6 Weighted Pattern Mining

2.6.1 Introduction of weight in mining

The concept of weight is very important in mining. Idea of weighted pattern mining was given in [13]. In our real life, frequency does not indicate the importance of an entity all the time. It is possible that, an item or entity or a sequence is not so frequent but its small presence can create huge importance. For example, suppose we are in a market where both day to day goods like bread, rice, groceries are sold and also other expensive electrical goods such as phone,television are sold. Definitely the frequency of selling food products will be much higher than the selling of electrical products. But if we think about the bar of profit or expensiveness electrical goods may cross the food products even though they are not sold as frequent as them. So the idea of weight was introduced in pattern mining to find out better characteristics of database or patterns, in other words to find specific weighted important entities [13]. There is another very important usage of applying weight and that is to discover special itemsets or patterns. In real life

there exists many scenarios where we express interests towards very special items rather not focusing on each and every other existing items in search domain. So, to discover the characteristics of our demanded product we apply comparatively higher weight to them than other products such that during mining our product of interest is only captured.

2.6.2 Downward Closure Property

A very important concept in pattern mining is downward closure property, a vast important topic in pruning unnecessary candidates generation and to make the desirable candidate generation faster and efficient. Downward Closure property has been a heart of many pattern generation algorithms. This property may not be directly focused everywhere but analogically used. This property actually says that, if a set of items is not frequent then any of its super patterns can not be frequent. And this concept helps to prune a huge amount of candidate generation. But when the weight was introduced this property could not play its part trivially [13]. Basically after weight addition its equally possible that an weighted set of items which was not considered as a proper candidate, its super patterns might be needed to be considered as a candidate. Suppose, we have a string "ababacabacabaab". And and we define weight of symbol 'a' is 0.1 and symbol 'b' is 0.9 and symbol 'c' is 0.5. Now the positions where 'a" occurred are $\{0, 2, 4, 6, 8, 10, 12, 13\}$ and positions where "ab" occurred $\{0, 2, 6, 10, 13\}$. Suppose we define the weight of a pattern as the average of weighted sum of each symbol of the pattern. So by this definition weight of pattern "a" is 0.1 and weight of pattern "ab" is $(0.1 + 0.9)/2 = 0.5$. And the importance of a pattern we define as *weight of a pattern* $\times$ *number of positions where it occurred* and minimum support threshold is 2.0. Here the importance of pattern "a" is $0.1 \times 8 = 0.8$ and importance of pattern "ab" is $0.5*5 = 2.5$. So though pattern "a" is not considered as an important frequent pattern but its super pattern "ab" will be considered as a frequent pattern. And this was an example where exact downwards closure

property could not be applied while mining weighted frequent patterns. Our future sections will describe much more about this concept where we will discuss about how this property been modified for using in weighted versions.

2.7 Related Works

2.7.1 Efficient Periodicity Mining in Time Series Databases Using Suffix Trees [11]

Problem Statement: This paper proposes a technique of mining several types of periodic patterns from time series databases.

Terminologies: The terms Perfect Periodicity, Confidence, Symbol Periodicity, Sequence Periodicity, Segment Periodicity are prerequisites for the following sections. All of the terms are described in 2.4.2.

Algorithm Description: The time series can be represented as a string. In the first step of the algorithm, a suffix tree is built containing the string. For every node of the suffix, an occurrence vector is created. Lets suppose we get string S, traversing from the root of the suffix tree to some node. Occurrence vector of that contains the starting positions of all the suffixes that have string S as a prefix. By using occurrence vector of some node, all types of periodicity containing string S as a pattern can be determined efficiently. For example if occurrence vector for some node is $\{3, 7, 11, 19\}$ we see that the string S is periodic with $p = 4$ and $stPos = 3$. As position 15 is not in occurrence vector, its not perfect periodicity. By running a 2 dimensional loop on the elements of occurrence vector, we can determine some candidate p and check the occurrence vector using another loop for all possible starting positions to determine periodicity. According to downward closure property, for some node N, if there are no frequent patterns ending at the

node of a suffix tree, there will not be any frequent pattern ending at the nodes which are in the subtree of node N.

Analysis and Challenges:

- Introduction of suffix tree and occurrence vector has added a new dimension to time series mining.
- The algorithm can mine a time series for several kinds of periodicities considering different starting positions.
- Periodicity detection allows some noise.
- Measures are taken to avoid redundant patterns.
- Requires a single scan of the database to construct the suffix tree and the rest of the mining can be done from the suffix tree.

2.7.2 An Efficient Approach to mine Flexible Periodic Patterns in Time Series Databases [18]

Problem Statement: This paper also proposes a technique of mining several types of periodic patterns from time series databases. The advancement is introduction to flexible pattern mining in time series database by escaping characters.

Terminologies: Suppose you have a time series T = "abcc abdc acdc abdc". You want to generate patterns like T = "a*c" or "a**c" or "a***c". Where '*' represents that the event of that position can be anything. A threshold denotes the maximum number of intermediate do not care events. The pattern "a**c" has frequency 4 in the provided time series.

Algorithm Description: Tree construction process and technique is same as [11]. The pattern mining process is different. New candidate patterns are generated through a technique that is similar to breadth first search. If the current

candidate pattern for example is “aba”, we can generate new candidates from it in the next step by appending any event ‘x’ (“abax”) or by adding do not care event (“aba*”). Every candidate will have an occurrence vector generated with it. If the size of the occurrence vector does not satisfy minimum support threshold the algorithm stops to generate candidate patterns from it. Like, here, If candidate pattern “aba” is not frequent, we can stop generating candidate patterns from it.

Analysis and Challenges:

- Time series for several kinds of periodicities considering different starting positions can be mined.
- Flexibility is a useful advancement.
- Only a single scan of the database is needed.

2.7.3 A New Framework for mining Weighted Periodic Patterns in Time Series Databases [19]

Problem Statement: This paper also proposes a technique of mining flexible periodic patterns from time series databases. The advancement is introduction of giving weight to the events.

Terminologies: What does assigning weights to event mean is described in 2.6.1.

Support/Frequency of Pattern, The support of a pattern is the number of times the pattern occurs after the considered starting position. We will define support of pattern P as $Support(P)$.

Weight of a Pattern, The weight of an item is a non-negative real number. The weight of a pattern is the average weight of the items/events in the pattern. We will define weight of pattern P as $Weight(P)$.

Weighted Support of a Pattern, $Wsupport(P) = Weight(P) * Support(P)$ A pattern P is weighted frequent if $Wsupport(P)$ is greater than or equal to minimum support threshold(σ).

Algorithm Description: Adding weight values to events does not change the tree construction process. In the mining process of [18] we pruned following trivial downward closure property. In the weighted pattern mining process, it is possible to get a sequential pattern that is weighted infrequent, but super patterns of the sequential pattern may be weighted frequent pattern.

Lemma, If P is not a weighted periodic pattern, its super patterns can be mined as a weighted periodic pattern, when $Support(P) \times MaxW \geq \sigma$ So, to prune the process of generating redundant candidates, maximum weight(**MAXW**) is used. If $Support(P) \times MaxW < \sigma$ any other super pattern of P can not be weighted frequent. But to generate final patterns, we need to check that $Support(P) \times Weight(P) \geq \sigma$.

Analysis and Challenges:

- Addition of weight helps to generate more interesting patterns.
- The algorithm cant prune all the unnecessary candidate patterns but a good number of unnecessary candidates.

2.7.4 An Efficient Algorithm for Sliding Window-Based Weighted Frequent Pattern Mining over Data Streams [13]

Problem Statement: This paper proposes an algorithm for sliding window-based weighted frequent pattern mining over data streams. It discovers useful recent knowledge from data stream by using only a single scan.

Terminologies: Some important definition provided in this literature are,

Batch, A batch consists of several transactions.

Window, A window consists of several batches.

Algorithm Description Every transaction is considered as a string. The items in the transaction are sorted by the increasing order of their weight. Then a prefix tree is built containing the transactions. Every node in the prefix tree contains a list of size n. Here n denotes the number of batches in a window. For example, if $n = 3$ and the list is [2,3,1] for some node then it means that the character of that node occurs twice in first batch of the window, three times in the second batch and once in the last batch. When the window slides, the list is left shifted to delete the information of the unnecessary batches. New entry is added to the right of the list to keep information of the added new batches. If at some point a node contains [0,0,0] as its list, it means that the node in not needed for the current window and the whole subtree of that node can be deleted from the prefix tree. Thus the tree is kept consistent and only the necessary information about the current batch is stored. The existing algorithms show that the main challenge in weighted frequent pattern mining is that the the weighted frequency of an item does not have downward closure property. In the mining process, global maximum weight(GMAXW) is used to utilize downward closure property. Its the maximum weight among all the items in the current window. Local maximum weight(LMAXW) is needed in mining a particular item. As the tree is sorted in weight ascending order, the advantage of bottom up mining order can be used. Suppose the bottom most item of the sorted item list is “a”. At first, the prefix tree of “a” is created by taking all the branches that ends in “a”. In this tree, the frequency of an item is the sum of its list items. Maximum possible weighted frequency of the items for item “a” will be the items frequency multiplied by the weight of “a” which actually denotes “a”. If the maximum weighted item for the tree of item **LMAXW** is “a”, then a new tree is built for pattern “ab”. Here, for every item of the tree, their weighted frequency will be determined by multiplying

their frequency by weight of "b" as this is the current **LMAXW**. Thus, patterns are generated recursively from the prefix tree.

Analysis and Challenges:

- The ordering of the elements is unaffected by the incremental update.
- The algorithm keeps the tree updated and keeps only the necessary information of the current window by a single scan of the database only.

2.8 Summary

This chapter contained a brief discussion regarding the important concepts and terminologies of our working domain. This chapter is actually a summary of our whole background study. The concepts and terms discussed here will work as a building block to understand the future upcoming chapters. We have also presented a summary of some related works to understand the problems which we have identified and tried to solve. This chapter will be very helpful to understand our algorithms and contributions.

Chapter 3

The Proposed Approach

3.1 Introduction

In chapter 2 we discussed the important concepts and terminologies related to our work. We also provided a short discussion of the previous literature to get a better insight into them and to have an idea about our contributions.
In this chapter, we will go through our proposed techniques in detail. In short, our following sections will discuss the topics mentioned below.

- Tree construction technique and necessary terminologies.
- Algorithm to handle dynamic events (*insertion and deletion*) in time series and a complete framework to handle both of them independently.
- A detailed example to understand our dynamicity handling algorithm.
- An efficient technique to reduce the number of patterns tested for candidacy in weighted time series pattern mining.
- An explanatory example to understand our pruning technique along.
- A discussion about the optimization tricks to implement the algorithms.

3.2 Tree Construction

We worked on time series, a field of sequential pattern mining. The coal of time series pattern mining is data streams. Many works have been done on time series. They have already shown that the best representation technique to represent data streams over time series is Suffix Tree Based Representations [11], [18], [19]. So, in the beginning of this section first we will introduce necessary terminologies related to represent our suffix tree and then the algorithm for building the tree will be discussed.

The most efficient(both memory and time) algorithm to create suffix tree is **Ukkonens algorithm** [15] . Ukkonens algorithm represents the suffix tree in a very compact and efficient mode. This is the most efficient technique to represent and construct suffix tree upto now. We will discuss our total contribution on the basis of this compact and efficient suffix tree representation.

3.2.1 Compact and Efficient Suffix Tree

At the beginning of this section, first we will talk about the terminologies and concepts which are important to understand our suffix tree and our approach. Then we will dive into the algorithm afterwards. This section will work as a building block for our algorithm.

- **Suffix Tree:** A special kind of tree through which we can get all the suffixes of a string by traversing the paths. An example representation of ukkonen's suffix tree structure is given in figure 3.1 for discretized string "abcababababc$".

- **Phase:** Phase actually means the iteration. Each iteration for a character or a symbol in data stream will be called a phase. So, for data stream "abcababababc" [we will start from leftmost symbol] , in 0th phase we will insert 'a', in 1st phase we will insert 'b', in 2nd phase we will insert 'c', in

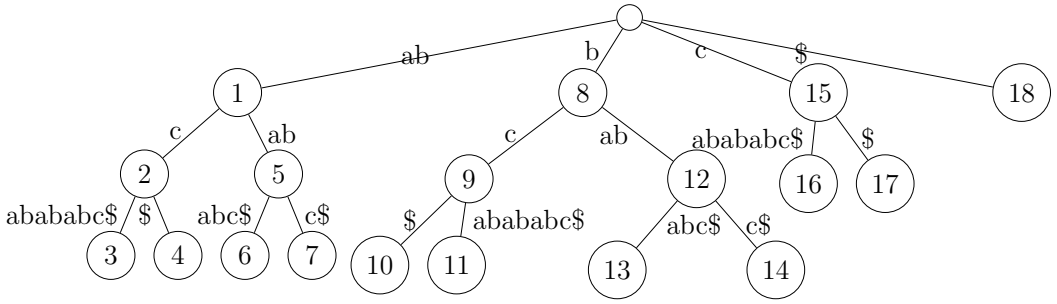

FIGURE 3.1: Explicit Suffix Tree for abcababababc$.

3rd phase we will insert 'a' and so on. So ith symbol in the data stream will correspond to ith phase of the algorithm.

- **Extension:** Branches of the suffix tree represents different suffixes of a string and suffixes may overlap. So while adding a symbol in the tree, the symbol can be added in multiple branches of the suffix tree. This is known as an extension of a phase.

- **Implicit Suffix Tree:** Implicit suffix tree basically means that at the current moment all the suffixes of the string, for which the tree is built, is not found explicitly but rather implicitly. Because of overlapping path characteristics many suffixes of the string could be present in the tree but might not end in leaf. To convert an implicit suffix tree to an explicit suffix tree [a suffix tree where all the suffixes of the string end in leaves] we have to add a unique symbol [a symbol which is not found upto now in the string] to the end of the string and complete the suffix tree. Now due to the addition of the unique symbol in the string, our next iteration of the algorithm will convert the implicit suffix tree to the explicit suffix tree for the string.

  For example, in the figure 3.2, we showed the implicit suffix tree for the string "abcababababc", where suffixes "abc", "bc" and "c" are not explicitly ending in leaves. But after addition of '$' symbol we get the explicit version which is shown in figure 3.1 where all the suffixes end in leaves. The interesting fact to observe is that, while converting an implicit suffix tree to its explicit form

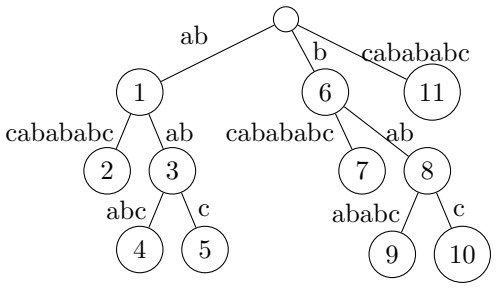

FIGURE 3.2: Implicit Suffix Tree for abcababab.

we introduce a new symbol to the string and because of that all the main suffixes add an additional symbol with them. This is not a costly operation because if we consider the figure 3.1 and figure 3.2, we can see that though 3.1 is the explicit suffix tree for string "abcababab$", it basically represents all the suffixes of string "abcababab" by ending in leaves with additional '$' symbol with each of the suffixes, if we just neglect the last symbol we can get all the real suffixes to work with.

- **Suffix Link:** Suffix link is a very important concept in suffix tree. This helps to traverse tree efficiently to add symbol in the tree. Definition of the suffix link can be given like this, *For every internal node **U** of the suffix tree with path $\alpha\beta$[where α is exactly one symbol and β can be a string of 0 or more length] there exists another node V with path β in the tree.* Basically suffix link of a node **A** tells us the position of another node **B** which is just one symbol different[from left side] from the string (from *root*) represented by **A** and according to Ukkonen's proposal each and every internal node of the tree will have a suffix link. From the figure 3.1 we can show the suffix links as follows,

Table 3.1: Suffix Links

Node	Suffix Link Pointing Node
1	8
2	9
5	12
8	root
9	15
12	1
15	root

- **Active Point:** Active Point basically consists of three important sub parts. They are described following.
  - *Active Node:* Active node means the node of the suffix tree from which the algorithm will work.
  - *Active Edge:* Active edge means the edge from active node upon which the algorithm proceeds in case of suffix overlapping.
  - *Active Length:* Active Length means from active node, in the direction of active edge how many characters been overlapped.

  These concepts of *Active Node*, *Active Edge* and *Active Length* is very helpful to construct suffix tree. The core importance of *Active point* is that it represents the current largest implicit suffix in the tree.

- **Remaining Suffix Count:** It means the length of the largest suffix which is implicit at the current moment in the tree.

- **Edge Label Compression:** The most important part of our compact suffix tree is we always try to maximize overlapping of suffixes in the tree paths. This brings both memory and implementation efficiency. Each path from root to a node denotes a suffix [for leaf nodes] or a substring of a suffix[for internal nodes]. Each edge contains a string which represents addition of characters from previous node and edge label compression actually denotes that we do not store the total string **S** to the edge rather we denote one *start* and *end* pointer from the main string to represent the same edge label **S**.

- **Rule 1 Extension:** This extension simply tells that for a new upcoming symbol in any phase, this symbol will be added into each and every leaf of the current tree.

- **Rule 2 Extension:** This extension simply denotes that during any phase and during any extension of the same phase, if for a symbol no branch is found from the current *activeNode* then create a branch for that symbol from the current *activeNode*.

- **Rule 3 Extension:** This simply says about the compact structure of the tree by maximizing suffixes overlapping in the branches.

3.2.2 Pseudo code for Tree Construction

In the previous section, we discussed about the important concepts and terminologies related to Ukkonen's algorithm to construct suffix tree. These concepts also work as a basis for our dynamicity handling algorithm. In this section we will provide the pseudo code for tree construction and the following section will provide an example to understand the procedure.

Algorithm 1 Tree Construction Algorithm

1: **Input:** A data stream *text* in time series.
2: **Output:** An explicit suffix tree for the provided data stream *text*.
3: **Method:**
4: **procedure** INITIALIZE
5: **SET** SuffixTreeNode address ref.*root*, *lastNewNode*, *activeNode* set NULL
6: **SET** *activeEdge*, *leafEnd* ← -1
7: **SET** *activeLength*, *remainingSuffixCount* ← 0
8: **SET** *rootEnd*, *splitEND* ← NULL
9: **procedure** NEWNODE(**Integer** *start*,**Integer Reference** *end*)
10: **DECLARE** SuffixTreeNode ref.*node*
11: **SET** *node.child* ← *NULL*, *node.suffixLink* ← *root*
12: **SET** *node.start* ← *start*, *node.end* ← *end*
13: **return** *node*
14: **procedure** EDGELENGTH(**SuffixTreeNode Reference** *node*)
15: **return** (*node.end ref. value* − *node.start* + 1)
16: **procedure** WALKDOWN(**SuffixTreeNode Reference** *node*)
17: **if** (*activeLength* is greater or equal **edgeLength** (*node*)) **then**
18: **UPDATE** *activeNode, activeLength, activeEdge* by traversing tree
19: **return***True*
20: **procedure** EXTENDSUFFIXTREE(**Integer** *pos*)
21: **SET** *leafEnd* ← *pos*, *lastNewNode* ← NULL
22: **INCREMENT** *remainingSuffixCount* by 1
23: **while** *remainingSuffixCount* is greater than zero **do**
24: **if** (*activeLength* equals zero) **then**
25: **SET** *activeEdge* ← *pos*
26: **if** (*activeNode* doesn't have branch for the symbol in *activeEdge*) **then**
27: **SET** *newChild* ← **newNode**(*pos*, *leafEnd*)
28: **ADD** *newChild* in *activeNode.child* list
29: **if** (*lastNewNode* is not NULL) **then**
30: **SET** *lastNewNode.suffixLink* ← *newChild*
31: **SET** *lastNewNode* ← *newChild*

```
32:         else if (activeNode has branch for the symbol in activeEdge ) then
33:             SET next ← activeNode.child for the symbol in activeEdge
34:             if (walkdown(next) returns True ) then
35:                 goto while with updated activePoint
36:             if (symbol in pos matches in direction of activePoint) then
37:                 INCREMENT activeLength
38:                 if lastNewNode not NULL and activeNode not root then
39:                     SET lastNewNode.SuffixLink ← activeNode
40:                     SET lastNewNode ← NULL
41:                 break while
42:                 SPLIT path
43:                 CREATE new reference for splitNode
44:                 UPDATE the path link through splitNode as parent child
45:                 if lastNewNode is not NULL then
46:                     lastNewNode.suffixLink ← splitNode
47:                 lastNewNode ← splitNode
48:         DECREMENT remainingSuffixCount
49:         if activeNode is root and activeLength greater than zero then
50:             DECREMENTactiveLength
51:             SET activeEdge ← (pos − remainingSuffixCount + 1)
52:         else if activeNode not equals root then
53:             SET activeNode ← activeNode.suffixLink
54:         goto while
55: procedure MAIN_PROCEDURE
56:     for Each position i in text do
57:         call ExtendSuffixTree( i )
```

3.2.3 Simulation of Tree Construction Algorithm

In this section, we will show a simulation of tree construction algorithm. For example, here we are considering string "abcababababc$". For each *phase* of the algorithm(for each symbol from left to right), we will show the generated tree along with resultant *activeNode, activeEdge, activeLength* and *remainingSuffixCount* respectively.

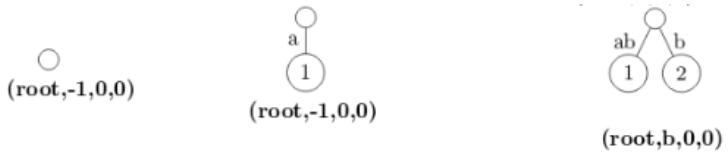

FIGURE 3.3: Step 0 - 2, added "ab"

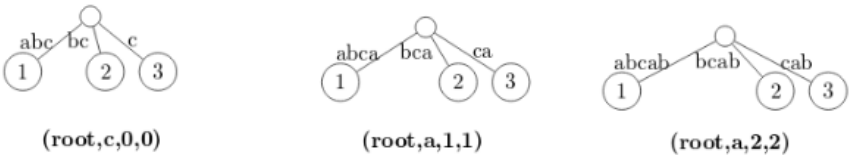

FIGURE 3.4: Step 3 - 5, added "cab"

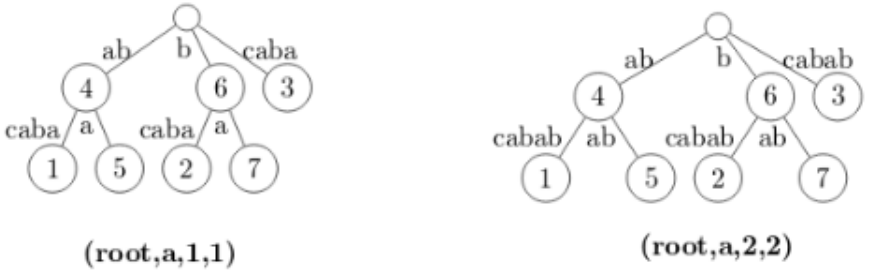

FIGURE 3.5: Step 6 - 7, added "ab"

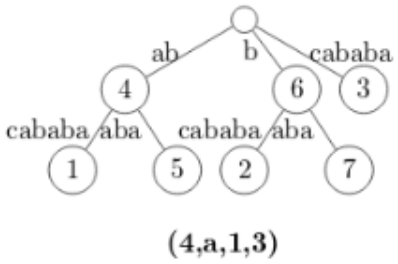

FIGURE 3.6: Step 8, added "a"

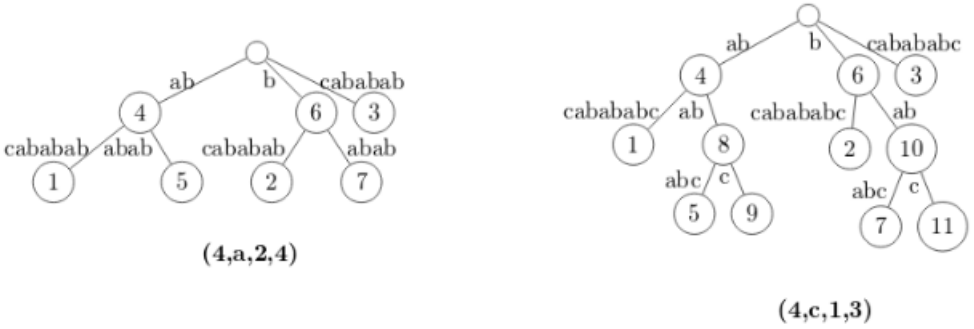

FIGURE 3.7: Step 9 - 10 added "bc"

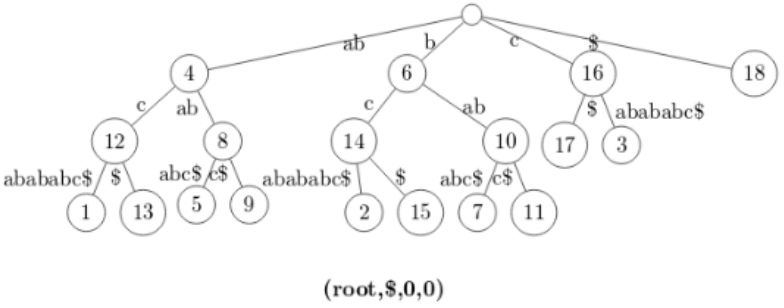

FIGURE 3.8: Step 11, added "$", Tree Construction Algorithm Simulation

3.3 HDTS, Handling Dynamicity in Time Series

In this study we provide two contributions. This section will have a detailed description about our first contribution. Our First contribution centers around the problem of how to handle the dynamic database or dynamic data stream in time series efficiently. In this section, first we will introduce the necessary terminologies to understand our algorithm, then we will dive into our approach and in the end of this section we will provide a pseudo code and a simulation to understand our algorithm.

3.3.1 Definitions and Terminologies

In this section we will provide all the necessary definitions and terminologies to understand our Dynamicity Handling Algorithm.

- **Modification Events:** By modification events we mean those events which change the current input string. There can be two type of modifications events, *insertion events* and *deletion events*, they are defined as follows,
  - **Deletion Events:** By deletion in data stream, we define the deletion of the largest suffix present at current moment in the string resulting in a shorter string.
    For example, if we run a deletion event on the string of figure 3.1, then we delete the largest suffix "abcababababc$" resulting in a shorter main string from "abcababababc$" to "bcababababc$".
  - **Insertion Events:** By definition of insertion in data stream we mean to add new symbol at the end of current string/data stream resulting in increasing the length of existing suffixes and adding new suffixes.
    For example, if we consider the string of figure 3.1, we get "abcababababc$", where main string is "abcababababc" and '$' is added to it to convert the suffix tree from implicit to explicit. Now, if we run an insertion event for symbol α, then the resulting string or data stream upon which we will

work will be "abcababababcα\$", where main string is "abcababababcα" and '\$' works as before. This shifting of '\$' and bringing necessary change in tree plays a very vital role in our algorithm.

- **Consistency:** By consistency we define that, on the eve of modification events (**insertion** or **deletion**) on the database, the whole underlying structure will never have to be built from scratch again rather necessary modifications will be brought about in the structure to reflect the current database.

3.3.2 Problem in Existing Solution

Existing literature proposes to reconstruct the whole underlying structure for each modification event. But this approach is very costly in case of huge data stream or frequent modification events where the total cost actually is $\mathcal{O}(k \times length(datastream))$, where k = number of times modification events occurred.

Our solution focuses on dynamically updating the tree in lieu of reconstruction for modification events and our algorithm's asymptotic complexity is $\mathcal{O}(k \times length(\text{number of modified symbols }))$. In general scenario, number of modified symbols are much less compared to the length of the data stream and there can be many modification events occurring. So, it can be easily observed that, our algorithm will perform much much better.

3.3.3 Handling Deletion Events

In this section we will discuss how to handle deletion events. We have already gave definition of deletion events in section 3.3.1 and here we will explain our algorithm to handle deletion events.

According to the definition of deletion in data stream we mean deleting the largest suffix present at the current moment from data stream resulting in a shorter

string or data stream. But before deleting the suffix from the tree, the tree must be in its explicit form and in our algorithm we define this condition as ***Deletion Condition***.

Condition 1. ***Deletion Condition*****,** Before each deletion event in the tree the whole tree must be explicit by ending all the suffixes in the leaves.

Reason: The logic behind this is, if all the suffixes ending in leaves then the whole process becomes much simpler by just removing one leaf node from the tree to delete a suffix. Like for the tree in figure 1, if we just remove node 3, then the largest suffix ***"abcababababc$"*** will be deleted from the tree.

Now we will discuss about the facts which can occur because of deletion events and how to handle those facts so that the tree is always explicit for the remaining string. We will also give the rationale behind the steps.

Proposition 1. ***Conversion from internal to leaf*** , If after one or more deletion events there remains any internal node (which is not **root**) which has lost all of its child nodes then this internal node has to be converted to a leaf node and if any other internal node was pointing to this node as its **suffix link**, then this link has to be removed and will be redirected to **root**.

Proof. Idea of this proof is very trivial. Suppose, we have an explicit suffix tree of figure 3.9, where 0 is the root node, internal nodes are $1, 4$ and leaf nodes are $2, 3, 5$ and 6. 1's suffix link is pointed to root(not showed in figure) and 4's suffix link is pointed to 1. In the figure edge labels are A, B, C, D, E and F, all of them are strings having length of at least one and as node 4's suffix link is pointing to node 1, we can say that from string D, if we leave the first symbol then we get the string A.

Now, suppose we have to delete the suffixes "AB" and "AC". For this, we remove node 2 and node 3 and get the next tree as result. Now, when node

1, looses all of it's children, then definitely it can not be an internal node from Ukkonen's proposal. So, we need to covert 1 to a leaf node from an internal node. Now, according to ukkonen's proposal, each and every internal node of the tree will point to another internal node of the tree or the root of the tree. As, now 1 has become a leaf node, so now 4 can not point to 1 as its suffix link. So, this link will be removed from 1 now and will be redirected to root. Because leaving the first symbol from string D, there can be no other paths rather than A. from *root*.

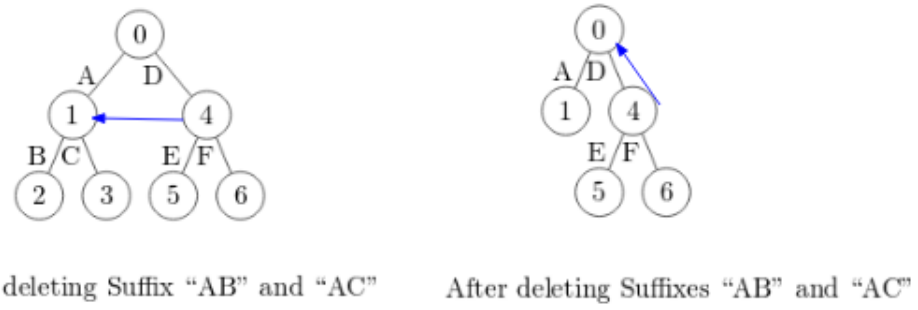

FIGURE 3.9: Conversion From Internal to Leaf

□

Proposition 2. ***Merging Path*****,** If after one or more deletions, any internal node becomes a node having only one child then the child of this node will be connected directly with the parent of this node(if exists because **root** doesn't have parent) and this node will be deleted and the node which was pointing it as its suffix link will now point to **root** as suffix link.

Proof. This proof is also trivial. Suppose, we have the tree of figure 3.10, where we have shown two states of the tree, *before merging* and *after merging*. In the first tree, there exists a single childed internal node 1. From our proposition we will delete this node and merge its only child node 2 with its parent *root* with merged edge label AB.

The reasoning behind this is, according to Ukkonen's proposal the tree will always be in compact form by maximizing overlapping of suffixes and minimizing the number of internal nodes. An internal node with just single child node violates this factor. To keep the tree always compact and consistent according to Ukkonen's proposal we provide this proposition. After merging, another issue arises about adjusting the suffix links. Like in figure 3.10, node 5 points to node 1 as its suffix link. But after merging we delete node 1. So, now suffix link of node 5 will point to root. The rational behind adjusting the suffix link to root has already been discussed in the proof of previous proposition. □

FIGURE 3.10: Merging Path

So, in this section we discussed about the factors or cases which can occur due to deletion events and how to handle them so that the tree is always consistent with the updated data stream along with its compact and explicit form. Our main goal has always been to dynamically update the tree for the modified data stream rather than reconstructing every time.

3.3.4 Handling Insertion Events

This section will discuss about our approach to handle the insertion events. We have already provided the definition of insertion events in section 3.3.1 and also gave an example to understand. This section will first explain the core ideologies

behind our algorithm, the facts which we actually had to consider to design our algorithm. Then we will dive into the algorithm and will provide a simulation to understand the technique.

From the discussion of section 3.3.1, we can give a brief reminder about the definition of insertion event. By definition of insertion in data stream we mean to add new symbol at the end of current data stream resulting in increasing the length of existing suffixes and adding new suffixes. In the provided example, we showed that before insertion event we had string or data stream "abcababababc$" where main string was "abcababababc" and '$' was added to convert the tree from implicit to explicit. After addition of new insertion event α, we get the resultant data stream as "abcababababc$\alpha$$", where as before the main string part is "abcababababcα" and '$' is a unique symbol. This shifting of unique symbol is a very important concept of our algorithm and will be discussed shortly.

Before discussing our approach first we will talk about our goal to handle insertion.

Goal, We use Ukkonen's algorithm to construct our suffix tree for initial configuration without thinking of any kind of dynamicity. But then the main problem is how to modify the tree in such a way so that the same algorithm can be used again to add symbols. Our algorithm converts the tree in such a measure along with setting the variables in such a way that the same algorithm of Ukkonen's can be used again without any alteration.

Now we will explain the challenges faced to convert the tree to the desired state along with the rationals behind them.

Problem 1. ***Removal of unique symbol***, Before any insertion event in the tree, if the tree contains unique character which was added to convert the tree from implicit to explicit, it has to be removed.

There are a good amount of reasons behind this problem. They are described as follows.

- **Importance of unique symbol:** A strong rationale behind the problem is this unique symbol. First we need to understand why we introduce this symbol. The sole purpose to add this symbol in data stream is if the tree is in implicit form, then by adding this symbol in tree, it will be converted to explicit form. It can be the case that, even if we do not add the symbol, the tree is already in explicit form. Then by adding this symbol we will get a new node from *root* for this symbol along with an extra symbol added in the last of all the existing strings. But, if the tree is in implicit form, then this addition will complete all the implicit suffixes by making the ending in leaves. We already discussed beforehand that this extra addition of symbol is not very costly operation because for each suffix if we ignore the last symbol then we get the main suffix and as all the suffixes are ending in leaves, it becomes very easy to access and use them. So, for case independence, we always consider adding a unique symbol in the end.

- **String Conversion Issue:** Suppose we have a suffix tree(implicit or explicit) for string α, where the $length(\alpha) > 0$, now we add '$' to the tree, resulting in an explicit suffix tree for string $\alpha\$$. Now, if we add new string β (where the $length(\beta) > 0$) to the existing string, what should be the real string to consider. Unarguably, $\alpha\beta$ will be the main string to consider(unique symbol is not actually the input symbol) and the tree must represent this. Then we can add '$' to the tree for case independent implicit to explicit conversion issue. So, before inserting β to the tree, we must erase the effect of '$' from the tree. Now, what can be the problem, if made tree for string "$\alpha\$\beta$" and then introduce another new unique symbol, suppose 'C' resulting in explicit suffix tree for string "$\alpha\$\beta C$". We know, while working with the suffixes, we ignore the unique symbols. It is very easy to remove unique

symbol from the last than to middle for any string. So, we can easily remove 'C' from here but removing '$' will not be easy.

- **Rule 3 Extension Violation:** We have already given a very brief discussion regarding rule 3 extension in Section 3.2.1, which maximizes the overlapping of suffixes. Now, we know what addition of a unique symbol does. It can create leaves, split existing paths etc. Now, from the above discussions, we already gave rationals behind erasing the effects of unique symbol before new insertion events. Another very important reason can be stated as compactness. If we do not remove unique symbols, then while adding symbols (for new insertion events) to the tree we will not be able to maximize overlapping of suffixes. Because by erasing the effects we can actually convert the tree back to implicit form and implicit form guarantees maximization of overlapping of suffixes.

Problem 2. ***Finding Active Point for New Pass***, In problem 1, we discussed why we need to remove the unique symbol for the new insertion events. So, when we will remove the unique symbol and its effects then the tree will be converted to another compact state. Now, to insert new upcoming symbols to the tree, we will again run the same Ukkonen's tree construction algorithm (according to our **Goal**). But then a very vital issue arrives. Each new pass of Ukkonen's algorithm starts with the current largest suffix which is implicit in the tree. This is found through *activeNode*, *activeLength*, *activeEdge* and *remainingSuffixCount*. These variables need to be set before any new pass of the Ukkonen's algorithm.

Now we will dive into the solutions. At first we will discuss our approach to remove the effect of unique symbol from the tree and then will describe the algorithm to find new *active point* for the new pass.

Solution 1. ***Removing Unique Symbol and Its Effects,*** This is the solution of problem 1. This solution focuses on erasing the effects of unique symbol

from the tree. Due to addition of unique symbol in the tree, we face two types of cases, which we need to solve to revert back the effects.

- **Case 1:** *A node for symbol '$' is created from an existing node.*
  In this case, we need to delete the created node for '$' symbol. It is possible that, after deleting the node, its parent looses all of its child nodes, then we need to convert its parent(if it's not *root*) to a leaf node from an internal node according to the proposition stated in Section 3.3.3. This case is shown as an example in figure 3.11, where the first tree represents the state before addition of '$' in the tree and the second tree represents the state after addition, when the tree becomes explicit. Now according to this case, when we will start new phase, we will revert back to the original tree before adding '$'.

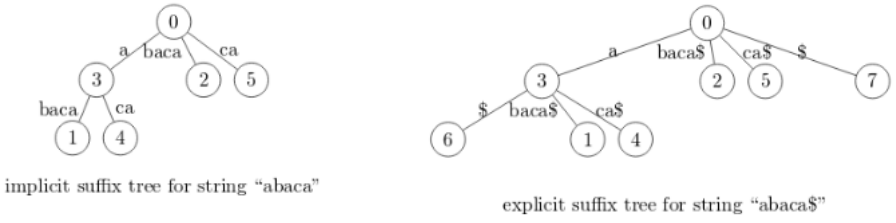

FIGURE 3.11: Case 1 Example

- **Case 2:** *A Path is Splitted.*
  The second case which can occur due to addition of unique symbol in the tree is splitting of an existing path. This case can be explained using figure 3.12. In the figure, we can see, after adding '$', to the tree, we get some extra nodes where two existing paths get splitted, path from *root* to node 1 and path from *root* to node 2, creating new splitted nodes 3 and 5 along with child nodes for unique symbol (node 4 and 6). The solution of this case is to revert back the effect to the tree of (a), where we delete nodes created for '$' symbol and merge the paths(*root* to node 1 and *root* to node 2) to get the previous state according to the proposition 3.3.3 (**Merging Paths**).

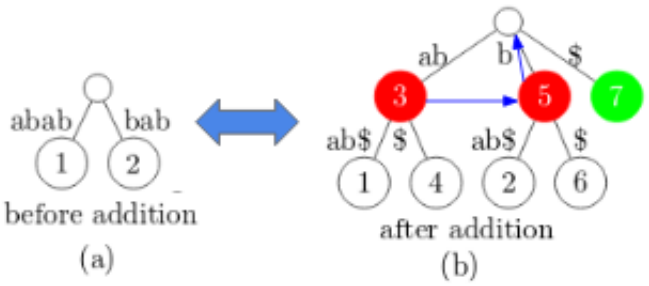

FIGURE 3.12: Case 2 Example

So, what is the main intuition behind the solution 1. Main ideology is, when new input stream appears what it does is, it expands the already existing stream. We modify the tree to such extent that our tree becomes ready to expand the old existing suffixes to their new state. Thus why our dynamically update algorithm works.

Solution 2. Finding Active Node for New Phase, Each deletion of '$' related node either converts an explicit suffix to implicit or creates a leaf node. So, it can be calculated that how many suffixes became implicit and *this number will denote the length of the current largest implicit suffix in the tree.* The rationale behind this is that in Ukkonen's compact structure a suffix becomes implicit along with all of its sub suffixes and these strings belong to the ending side of the string. So if a string "abc" becomes implicit then definitely "bc" , "c" both will become implicit also and then by traversing in the tree to find "abc" we can find the new *activePoint*, *activeEdge* and *activeLength* for the new pass and the number of implicit suffixes in the tree will be length of "abc", so 3. These information can also be saved while deleting the nodes.

Idea of the solution, can be explained with figure 3.13. In the figure, we have an explicit suffix tree, where blue arrows denote suffix links direction and '$' is the unique symbol. Now, from the figure it can be understood that while adding the '$' in the tree, first we create node V1 (with edge label $\alpha\beta\gamma$), then node V2 (with edge label $\beta\gamma$), then node V3 (with edge label γ) and at last node

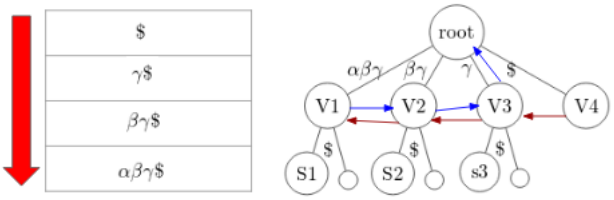

FIGURE 3.13: Finding Active Point
.0.0

V4 (with edge label $). So, the interesting thing to observe here is the relation between the length of the largest implicit suffix and the number of moves through suffix links are equal. Due to any deletion event, larger explicit suffix is affected first before any of its smaller sub suffixes. Like from figure 3.13, "$\alpha\beta\gamma$\$" will be deleted before "$\beta\gamma$\$", "γ\$" and "\$" gets deleted.

Now, if we want to erase the effect of '$' from the tree, what we can do is start from the smallest explicit suffix created due to addition of '$' and then move forward to the larger explicit suffixes (created due to '$') and remove the effect. Now, it can be understood that, we will end up in the largest explicit suffix which still exists (deletion events effect suffixes). But the relation is sequential (because deletion event will effect the largest one first, then second largest and so on). So, while traversing from smaller to larger if we encounter some explicit suffix (created due to '$') is now not valid, we do not need to go forward any more, because all the larger suffixes before it has already been compromised. In simple words, as in deletion events we delete the nodes and modify the paths, we can save the modified nodes (due to addition of '$') sequentially (the order by which they were created) in a list. Now, when we will start to erase the effect we will traverse the modified nodes in reverse order from the list. If we get any node i which is not valid now, we can stop traversing, because we have already got the largest implicit suffix possible at the current tree in previous step. The idea of implementation can vary (using list or online while erasing the effects). But *remainingSuffixCount* will be how many suffixes we have converted from explicit

to implicit which is equal to the length of the largest implicit suffix, because it becomes implicit along with all of its smaller sub suffixes. Red links in the figure show the direction how we will traverse the nodes to erase effects.

3.3.5 Pseudocode For Handling Dynamicity in Time Series

In this section we will provide a pseudo code to understand our dynamicity handling algorithm. Our contribution is, in this study we provide a complete framework to handle both insertion and deletion events as independent module.

Algorithm 2 HDTS

```
1: Input: An input data stream text and an explicit suffix tree, tree, made upon
   this data stream along with different type of modification events ( insertion
   events, deletion events ).
2: Output: A modified explicit suffix tree.
3: Method:
4: procedure DECREMENT_TREE(position)
5: /* the suffix starting from position of text will be deleted*/
6:     Delete the suffix starting from position by traversing from root to V
7:     if ( parent of V has lost all of its children and is not root) then
8:         Covert parent of V to a leaf node from an internal node
9:     else if ( parent of V has got only one child node and is not root) then
10:        Merge  the path from V's parent's parent to V's only single child
11:    RETURN
12: procedure INCREMENT_TREE( void )
13:    implicit ← 0
14:    if ( the tree is in implicit form ) then
15:        RETURN
16:    for each node V created due to '$',traverse in reverse do
17:        if  (V is not deleted ) then
18:            if  (V is case 1 type of node)  then
19:                Delete the child for '$' from V
20:                if  ( V has lost all its children and is not root) then
21:                    Convert V to leaf
22:                    Break
23:                Increment implicit by 1
```

```
24:             else if ( V is case 2 type of node) then
25:                 Delete the child for '$' from V
26:                 Merge the splitted path with parent of V
27:                 Increment implicit by 1
28:                 Delete V
29:         else if (V is deleted ) then
30:             Break
31:     remainingSuffixCount ← (implicit − 1)
32: /* '$' node from root is not considered */
33:     Remove '$' from text
34:     Remove '$' from all the existing suffixes of the tree using global reference
35:     Traverse tree to find the largest implicit suffix and update activePoint
36: /*the string to search is text.substring(start_pos=length(text)-
    implicit+1,end_pos=length(text))*/
37:     RETURN
38: procedure Insertion Module
39:     Call Increment_Tree ( ) /*To convert the tree from explicit to implicit and
    to set up the variables*/
40:     for each new symbol i from new input data stream do
41:         Add i to the end of text
42:         position ← (length(text) − 1)
43:         Call ExtendSuffixTree(position)
44:     RETURN
45: procedure Deletion Module
46:     if the tree is not in explicit form then
47:         Add '$' to the end of text
48:         position ← (length(text) − 1)
49:         Call ExtendSuffixTree(position)
50:     position ← leftmost position of text
51:     call Decrement_Tree(position))
52:     Shift start pointer to right by one move of text
53:     RETURN
```

3.3.6 Complexity Analysis of HDTS

According to the Ukkonen's algorithm the construction of the suffix tree takes $\mathcal{O}(n)$, where n is length of the string for which we construct the suffix tree. So if we reconstruct the tree k times due to modification events, then the total complexity will be $\mathcal{O}(k \times N)$, where N = length of the largest string, for which suffix tree was reconstructed in between k reconstructions.

Our framework *HDTS* focuses on modification rather than reconstruction. First our algorithm converts the explicit suffix tree to implicit suffix tree. Now suppose, for a string S of length n, due to addition of unique symbol, m implicit suffixes become explicit and always $m \leq (n-1)$ because at worst case if all the symbols of string S were same then due to the addition of unique symbol total $(n-1)$ suffixes will become explicit and m will be equal to $(n-1)$ otherwise m will always be less than $(n-1)$. So, the reverting complexity from explicit to implicit will be $\mathcal{O}(m)$ and we can save the updated active point while reverting. So the total complexity for single reverting and updating the new *active point* is $\mathcal{O}(m)$. Now the complexity for inserting new p symbols will be $\mathcal{O}(p)$. So total insertion module's complexity will be $\mathcal{O}(m+p)$.

Now we will analyse our deletion module's complexity. Suppose, we have a string S of length n and we will delete p characters from this sequence. So, in reconstruction the total complexity will be $\mathcal{O}(n-p)$. In our deletion module the complexity calculation is little bit difficult. The amortized complexity can be stated as $\mathcal{O}(p+\epsilon)$ where the upper bound of ϵ can be given as,

$$\epsilon << (n \times p - \frac{p \times (p-1)}{2} + 1) \tag{3.1}$$

The right side of equation 3.1 says the complexity if we had to traverse each symbol of each path to reach total p leaf nodes to delete. But due to edge label compression and compact structure of the tree this value ϵ is much much smaller and the dominating factor becomes p. So, $\mathcal{O}(p+\epsilon) \approx \mathcal{O}(p)$. So, the complexity of deleting one symbol is $\mathcal{O}(1)$ then. So, the complexity of reconstruction is $\mathcal{O}(n-p)$ and our complexity is $\mathcal{O}(p)$. Generally in real life data sets $p < \frac{n}{2}$ and in these cases our deletion module will perform very well. But if $p \geq \frac{n}{2}$ then the distinction will not be so significant.

So, now if we consider k modification events, then the total complexity will be $\mathcal{O}(k \times \mathbf{max}((M+P), Q)))$. M denotes the maximum number conversion from explicit to implicit due to removal of \$. P denotes the maximum number of characters

inserted and Q denotes the maximum number of characters deleted continuously. In real life time series, generally the effects of M during insertion events is not very significant. So, the total complexity actually becomes $\mathcal{O}(k \times \mathbf{max}(P, Q)))$. The total complexity actually depends on the number of largest modified symbols for some modification event. Let T be the length of largest modified symbols in some modification event. Hence, the total complexity analysis of this section summarizes that the total complexity of our framework in any module is $\mathcal{O}$(The Number of Modified Symbols). The main goal of our solution was to perform better in case of large sequence size or if small updates occurring frequently and the complexity analysis also proves this statement. Like, while reconstruction the complexity is $\mathcal{O}(k \times N)$ and during dynamically updating the complexity is $\mathcal{O}(k \times T)$ and from the mentioned arguments about the problems $\mathcal{O}(k \times N)$ $>\mathcal{O}(k \times T)$. So, our solution should perform better.

In the next section we will simulate our HDTS algorithm through an example.

3.3.7 Simulation of HDTS Algorithm

In this section, we will provide a simulation of our dynamic event handling algorithm. We will run ten successive dynamic events on the tree of figure 3.1. At odd number event, we will perform deletion on the tree and in even number we will perform insertion and after each event we will show the resultant tree along with the actions taken to dynamically update the tree.

a) **Event 1 (Deletion):**

- We will delete the largest suffix "abcababab c$" from the tree. So, to delete this suffix we will remove node 3. Then node 2 will have only one child (node 4). So, node 2 will be deleted and path from node 1 to node 4 will be merged. Resultant tree is shown in figure 3.14.

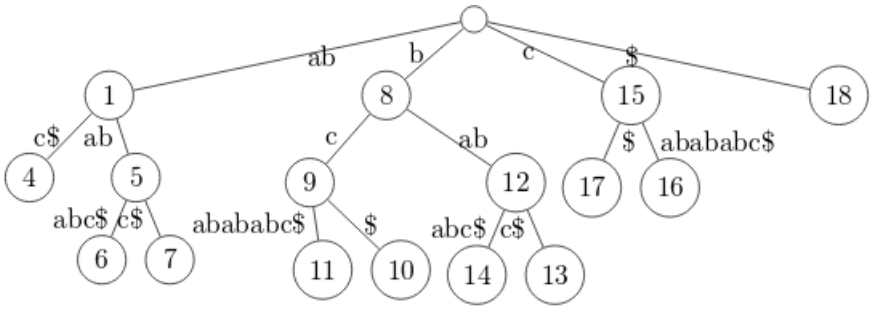

FIGURE 3.14: Event 1

b) **Event 2 (Insertion):**

- Our tree is in explicit condition. Before new insertion we will convert it to implicit. Resultant tree is shown in figure 3.15. where new *activeNode*, *activeLength*, *activeEdge* and *remainingSuffixCount* are 8, 1, 'c', 2 respectively. Reverted back nodes are 18, 15 and 9.

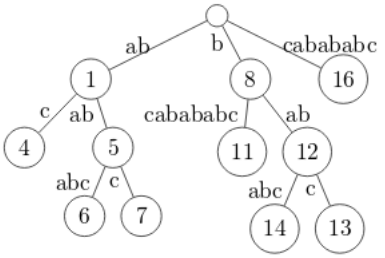

FIGURE 3.15: Event 2, Implicit Tree

- Now, we will add symbol 'a' in the tree. And after that we will add unique symbol '$'. Resultant explicit tree for data stream "bcabababca$" is given in figure 3.16.

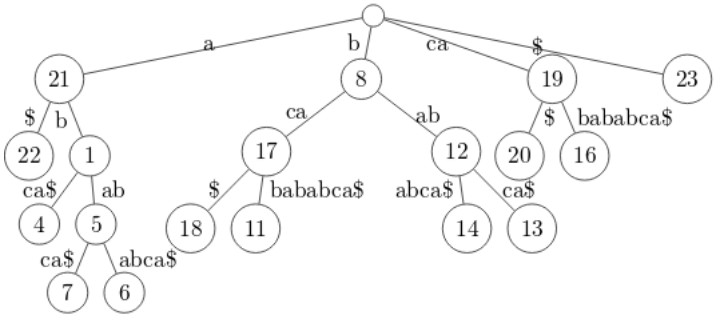

FIGURE 3.16: Event 2, Explicit Tree

c) **Event 3 (Deletion):**

- We will delete the largest suffix "bcabababca$" from the tree, resulting in deletion of node 11. But as node 17 will have only one child now, the path between node 8 and node 18 will merge. Resultant tree is shown in figure 3.17.

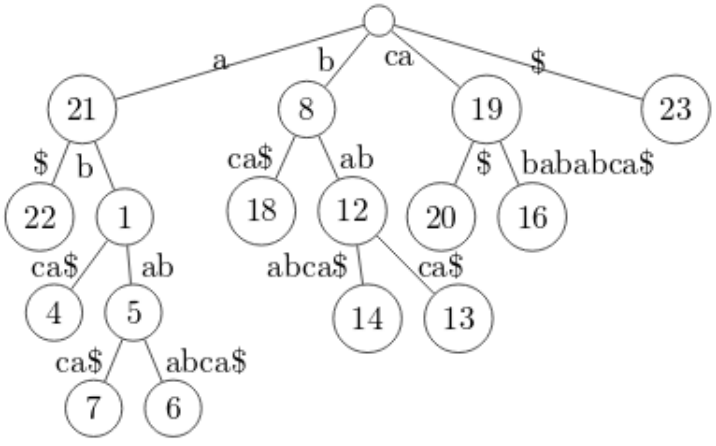

FIGURE 3.17: Event 3

d) **Event 4 (Insertion):**

- As our tree is in explicit form, first we will convert the tree to implicit. So node 23, 21 and 19's effect will be reverted. Resultant tree is shown in figure 3.18 where updated *activePoint*, *activeEdge*, activeLength and *remaining-SuffixCount* are *root*, 'c', 2, 2 respectively. Now, we add symbol 'b' and '$' to the tree resulting in the figure 3.19. Figure 3.19 is the implicit suffix tree of data stream "cababababcab$".

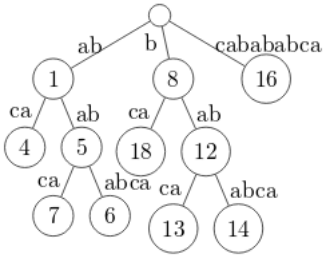

FIGURE 3.18: Event 4, Implicit Tree

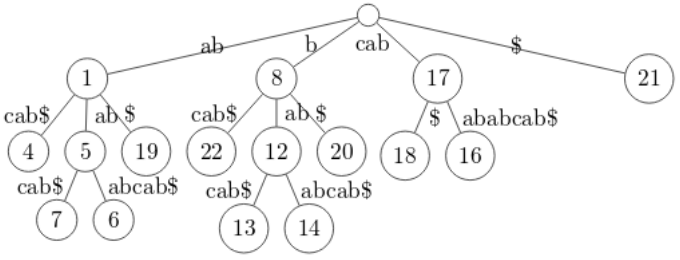

FIGURE 3.19: Event 4, Explicit Tree

e) **Event 5 (Deletion):**

- We will delete the largest suffix "cababababcab$" from the tree. So, node 16 will be deleted and due to node 17 having only one child, path from root to node 18 will merge. Resultant tree is shown in figure 3.20.

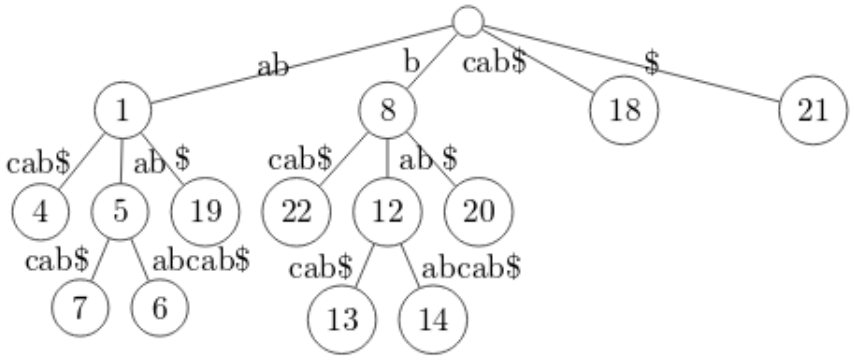

FIGURE 3.20: Event 5

f) **Event 6 (Insertion):**

- First we convert the explicit suffix tree to the implicit suffix tree. Resultant suffix tree is shown in figure 3.21 where new *activeNode*, *activeLength*, *activeEdge* and *remainingSuffixCount* are 1, 0, -1 and 2 respectively. Then we add new symbol 'c' to the tree and after that we add '$', resultant tree is shown in figure 3.22. It is the explicit suffix tree of data stream ababab-cabc$".

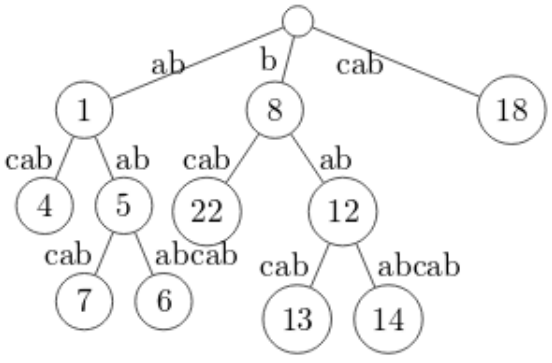

FIGURE 3.21: Event 6, Implicit Tree

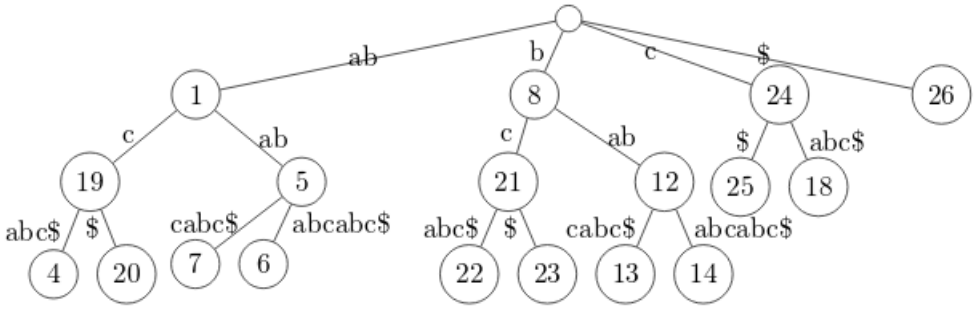

FIGURE 3.22: Event 6, Explicit Tree

g) **Event 7 (Deletion):**

- We delete the largest suffix "abababcabc$" from the tree resulting in deletion of node 6. But as node 5 will now have only one child (node 7), path from node 1 to node 5 and node 5 to node 7 will merge and node 5 will be deleted. Resultant tree is shown in figure 3.23.

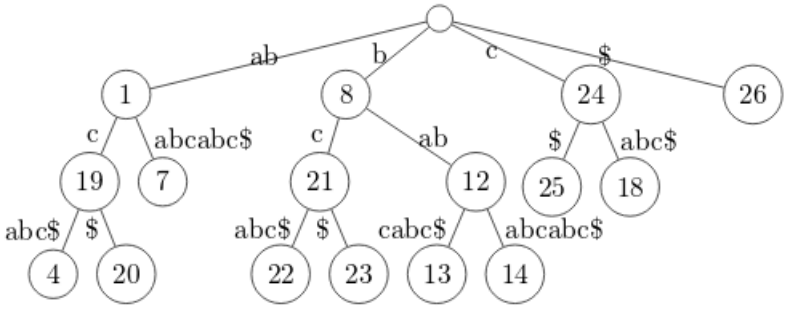

FIGURE 3.23: Event 7

h) **Event 8 (Insertion):**

- First we convert the explicit tree to implicit tree. The resultant implicit tree is shown in figure 3.24 where updated *activeNode*, *activeEdge*, *activeLength* and *remainingSuffixCount* are 1, c, 1 and 3 respectively.

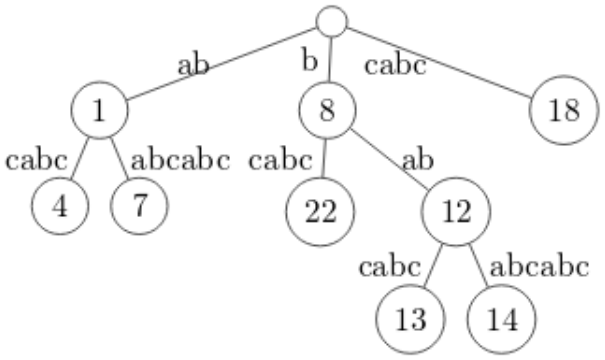

FIGURE 3.24: Event 8, Implicit Tree

- After that we add new symbol 'b' to the tree and following that we insert symbol '$' to the tree. Resultant tree of string "bababcabcb"is shown in figure 3.25.

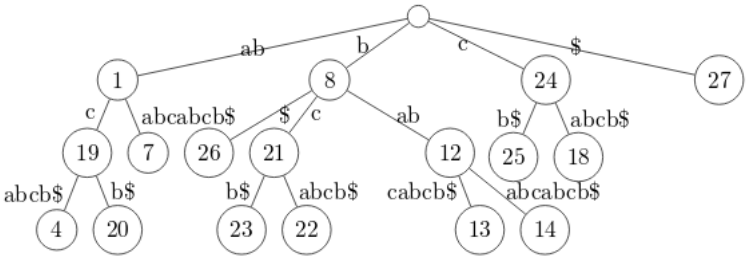

FIGURE 3.25: Event 8, Explicit Tree

h) **Event 9 (Deletion):**

- We delete the largest suffix "bababcabcb$" from the tree resulting in deletion of node 14. Then due to having single child node path from node 8 to node 12 and from node 12 to 13 will merge and node 12 will be deleted. Resultant suffix tree is shown in figure 3.26.

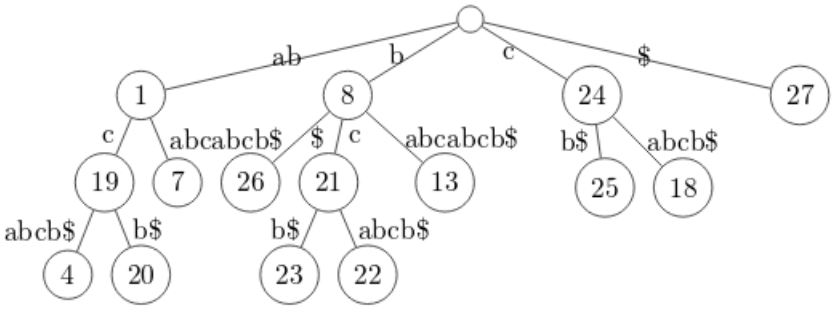

FIGURE 3.26: Event 9

h) **Event 10 (Insertion):**

- First we will convert the explicit tree to implicit tree. Resultant tree is shown in figure 3.27 where new *activeNode*, *activeLength*, *activeEdge* and *remainingSuffixCount* are 8, 0, -1 and 1 respectively.

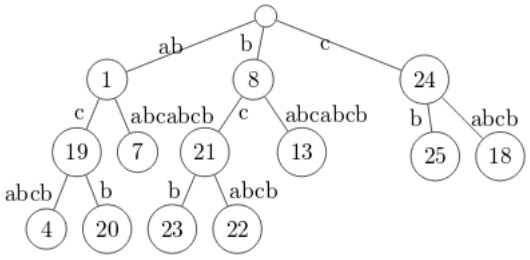

FIGURE 3.27: Event 10, Implicit Tree

- Now, we add symbol 'a' in the tree. After that, we insert symbol '$' in the tree. Resultant explicit tree for string "ababcabcba$" is shown in figure 3.28.

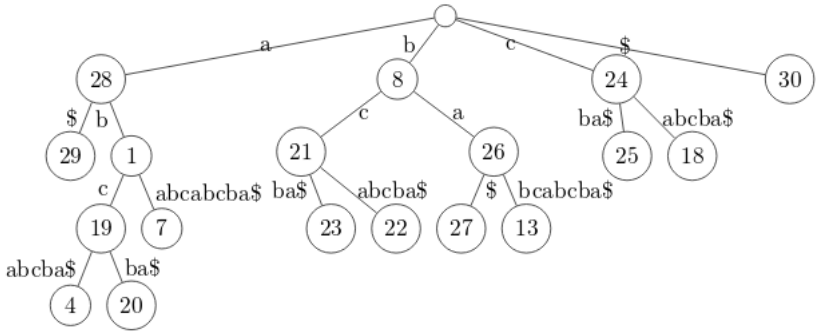

FIGURE 3.28: Event 10, Explicit Tree

3.4 Maximum Possible Weighted Support(MPWS) Pruning

Trivial Downward Closure Property does not hold in weighted pattern mining. The best known technique to optimize candidate generation process is to use maximum weight in the database(MaxW). We propose a new value MPWS for every node that gives a better performance in weighted time series pattern mining than MaxW Pruning. MPWS value of a node denotes Maximum Possible Weighted Support any pattern can achieve which ends in the subtree of that node.

Let's start with some definitions.

Definition 3.1. sumW(N)

sumW(N) denotes the sum of all the characters from root to node N. In Fig. 3.1, sumW(14) is the sum of weight of the characters 'b', 'a', 'b' and 'c'.

Definition 3.2. weight(P)

weight(P) s the average weight of the characters of pattern P. If P is "abac" and the weights of 'a', 'b' and 'c' are 0.8, 0.1 and 0.2 respectively, then weight(P) is $\frac{0.8+0.1+0.8+0.2}{4} = 0.475$

Definition 3.3. 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 3.4. weightedSupport(P)

$weightedSupport(P) = weight(P) \times support(P)$ where support(P) denotes the actual periodicity of the pattern P.

A pattern P is weighed frequent if, $weightedSupport(P) \geq \sigma$

Definition 3.5. cnt(A,B)

cnt(A,B) denotes number of characters encountered on the path from node A to node B.

In Fig. 3.1, cnt(8,13) is 5(ignoring '$').

Definition 3.6. 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. 3.1, maxW(8,13) is 0.8(ignoring '$') if the weights 'a', 'b' and 'c' are 0.8, 0.1 and 0.2 respectively.

Definition 3.7. sizeV(N)

sizeV(N) is the size of the occurrence of vector of node N.

Definition 3.8. subStr(A,B)

subStr(A,B) is the sub string encountered on the path from node A to node B.

In Fig. 3.1, subStr(8,13) is "ababc"(ignoring '$').

Definition 3.9. 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.

$$A = \frac{sumW(P) + maxW(P,N)}{cnt(R,P) + 1} \tag{3.2}$$

$$B = \frac{sumW(P) + maxW(P,N) \times cnt(P,N)}{cnt(R,P) + cnt(P,N)} \tag{3.3}$$

$$nodeW(N) = \max(A, B) \times sizeV(N) \tag{3.4}$$

Let us suppose,

$S_1 \leftarrow subStr(root, P)$

$S_2 \leftarrow subStr(P, N)$

$S_3 \leftarrow$ Any nonempty prefix of S_2

$S \leftarrow S_1 + S_3$

Lemma 3.10. *$nodeW(N) \geq weightedSupport(S)$ 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. 3.2 and Eqn. 3.3.

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. 3.2). Because Eqn. 3.2 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. 3.3) 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 3.11. MPWS(N)

MPWS(N) = is the Maximum value among the nodeW of all the nodes in the subtree(including node N) of node N.

nodeW(N) denotes the maximum possible weighted support a pattern ending just above that node can achieve. As MPWS(N) is the maximum of nodeW of all the nodes in the subtree, MPWS(N) is the maximum possible weighted support any pattern can achieve that ends in the subtree of node N.

3.4.1 Candidate generation

Candidate patterns can be generated by a breadth first search(bfs) in the suffix tree. When we encounter some node N, we do the following.

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

$S \leftarrow S_1 + S_3$

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 and test the occurrence vector of node N for periodicity detection.

3.4.2 Pruning Condition

In the suffix tree for a string of size N, the number of nodes will be around N. But the sum of the number of characters in the edges can be close to N^2. Thus, there can be around N^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.

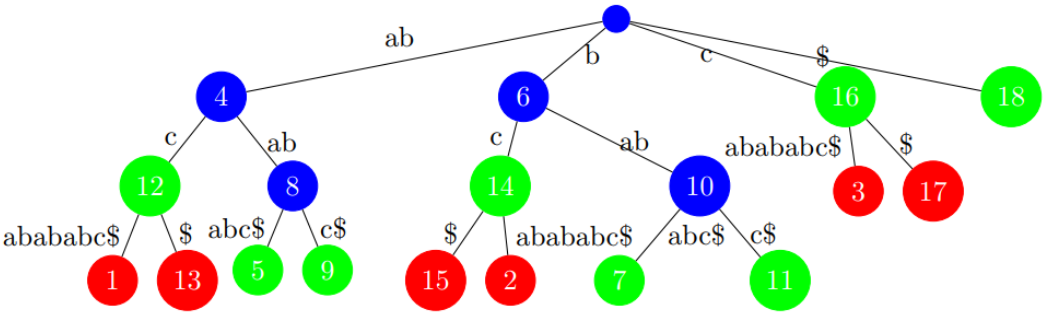

FIGURE 3.29: Example of MPWS Pruning

We can not use downward closure property here because even if some pattern P is not weighted frequent, it's supper pattern can be weighted frequent. The most used technique is to use the maximum weight(MaxW) in 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 3.12. *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 of the subtree in 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). □

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.

An example of our pruning approach is given in figure 3.29 and Table 3.1. figure 3.29 is the suffix tree of string "abcababababc$". Weight of symbols 'a', 'b', 'c' and '$' are 0.8, 0.1, 0.2 and 0 respectively and min_sup is 10%. Detailed calculation of mpws value is shown in Table 3.1.

There are 3 types of nodes in figure 3.29.

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 3.1: MPWS Necessary Values Calculated

- 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.
- A green node means, starting from that node, the whole subtree is unimportant and can be ignored during candidate generation.
- 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 the pruning. On the other hand, if we used MaxW Pruning, we had to test 10 patterns.

3.4.3 Pseudo code for Weighted Pattern Mining

In this section we will provide a pseudo code to understand our approach for mining the patterns. This section will use the definitions defined in the previous section. Besides those, we will define the following macros to make better understanding of our pruning technique and for simplicity of our algorithm.

Definition 3.13. edge(A,B)

Represents the edge between node A and node B.

Definition 3.14. stringConcat($String_1$**,** $String_2$**)**

Output will be concatenated string ($String_1 + String_2$).

Definition 3.15. nextSymbol(A,B)

Traversing through the edge between node A and B at a time we will concatenate single symbol from the edge(A,B) to the pattern and forward the pointer to the next symbol through the edge.

Now, let's dive into our pseudo code.

Algorithm 3 MPWS Pruning

1: **Input:** An explicit suffix tree *tree*, weight provided to the symbols, *min_sup*, input data stream *text*, each node having occurrence vector *occ_vec*.
2: **Output:** Generated patterns with their occurrences.
3: **Method:**
4: **procedure** DFS_MPWS_PRE_CAL(**SuffixTreeNode Reference** *node*)
5: **if** (*node* is not *root*) **then**
6: $edgeMax \leftarrow \mathbf{maxW}(node.parent, node)$
7: $sum \leftarrow node.parent.nodeW$
8: $A \leftarrow \frac{sum+edgeMax}{\mathbf{cnt}(root,node.parent)+1}$
9: $B \leftarrow \frac{sum+(edgeMax \times \mathbf{cnt}(node.parent,node))}{\mathbf{cnt}(root,node)}$
10: $node.nodeW \leftarrow \max(A, B)$
11: **else if** (node is root) **then**
12: $node.nodeW \leftarrow 0$

```
13:     node.mpws ← node.nodeW
14:     for each node i in node.child list do
15:         C ← DFS_MPWS_PRE_Cal(i)
16:         node.mpws ← max(C, node.mpws)
17:     RETURN node.mpws
18: procedure PatternMining
19:     Initialize Queue Q ← {}
20:     Initialize Set generatedPatterns ← {}
21:     for ( each node i in root.child list ) do
22:         if (i.mpws <σ) then
23:             Discard the whole subtree from i
24:         else if (i.mpws ≥σ) then
25:             pattern ← first symbol in the edge(root, i)
26:             Q.add_in_queue({i, pattern})
27:             if (weightedSupport(pattern) ≥σ) then
28:                 generatedPatterns.add_in_set({pattern, i.occ_vec})
29:     while Q is not empty do
30:         temp ← Q.top_from_queue()
31:         /*will have 2 elements temp.node,temp.pattern*/
32:         Q.pop_from_top()
33:         if (all symbols between edge(root,temp.node) been taken then
34:             for each node i in temp.node.child list do
35:                 if (i.mpws <σ) then
36:                     Discard the whole subtree from i
37:                 else if (i.mpws ≥σ) then
38:                     a ← temp.pattern
39:                     b ← first symbol in edge(temp.node, i))
40:                     pattern ← stringConcat(a, b)
41:                     Q.add_in_queue({i, pattern})
42:                     if (weightedSupport(pattern) ≥σ) then
43:                         generatedPatterns.add_in_set({pattern, i.occ_vec})
44:         else if (all symbols between edge(root,temp.node) not been taken then
45:             a ← temp.pattern
46:             b ← nextSymbol(temp.node.parent, temp.node)
47:             pattern ← stringConcat(a, b)
48:             Q.add_in_queue({temp.node, pattern})
```

```
49:             if (weightedSupport(pattern) ≥σ) then
50:                 generatedPatterns.add_in_set({pattern, temp.node.occ_vec})
51:     RETURN generatedPatterns
52: procedure MINING_MODULE
53:     Call DFS_MPWS_PRE_CAL(root)
54:     Call PATTERNMINING()
```

3.4.4 Complexity Analysis of Candidate Generation Using MPWS 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 by DFS_MPWS_PRE_CAL function. In that function we need the MaxW of each edge. We have determined it using RMQ in static data. The query complexity for each edge is $\mathcal{O}(1)$. Other than that, the DFS_MPWS_PRE_CAL function is just a Depth First Search(DFS) on the tree. So, the complexity is $\mathcal{O}(N)$.

After calculating the the MPWS values, we start generating candidates. There are L^2 possible patterns in a string of size L. We check the weighted support of the patterns using a Breadth First Search technique. But we visit each character of each edge. So the overall complexity was supposed to be $\mathcal{O}(L^2)$. But as we are using a heuristic value to prune, this will be much lesser than that. But that is dataset specific.

3.4.5 Optimization Tricks

We have provided the pseudo code to understand our approach in the previous section. Implementation ideas can vary, but two very interesting optimizations can be mentioned. First one is the use of the technique *Range Maximum Query on Static Data* to find the maxW for each node. It returns the maximum weighted character on a path in O(1) time. Insertion of each new character takes O(log(Size of The String)) time.

3.5 Snapshot of the Whole Process

In this study we proposed two algorithms. They are,

- **HDTS,** an algorithm to dynamically update the suffix tree due to modification events.
- **MPWS Pruning,** an efficient pruning technique to reduce the the number of patterns for candidacy test.

Our proposed solutions of two problems are completely independent of each other. So each of them have independent usage. But the most common example where both of them can be used is the *sliding window* problem in weighted time series framework. The largest length of string which we consider to construct our suffix tree is defined as the window size and the string upon which we we work is defined as the window in sliding window problem. Sliding window problem can be summarized as, when new input data comes, if it crosses the defined window size, then the window shifts to forget some older data and to embrace newer data to the window and if it does not cross the window size then newer data will be appended to the window and after that mining process will be performed on the window. So, sliding window problem gives two challenges.

- How to keep the data structure updated with the modified data stream.
- How to mine from the window (we are considering weighted framework due to our problem definition).

A snapshot of the whole process can be given as follows,

1. We convert the raw input to a discretized string by following any discretization technique. Discretized string brings implementation efficiency.
2. When a new input symbol appears, if it does not cross the window size we use our Insertion_Modeule to insert the symbol to the tree and if it crosses the window size then first we use Decrement_Module to delete the unnecessary older symbol and then Insertion_Module to insert the newer symbol to the tree.
3. To mine from the window, first we calculate occurrence vector of each node, then we use our Mining_Module to mine from the explicit suffix tree with an input min_sup. By this we will get the weighted frequent patterns from the window. But if we want to find weighted periodic patterns, then we can use periodicity detection algorithm on the occurrence vectors of the generated weighted frequent patterns.

3.5.1 Periodicity Detection Algorithm

Periodicity detection algorithm is given following.

Algorithm 4 Periodicity Detection

```
1: Input: A pattern X with occurrence vector occ_vec, min_sup, tolerance t.
2: Output: Determines X is periodically frequent or not.
3: Method:
4: procedure PERIODICITY_DETECTION(X, occ_vec)
5:     S ← occ_vec.size()
6:     for i := 0 ..... (S - 2) do
7:         cnt ← 1
8:         for j := i+1 ..... (S - 1) do
9:             period ← (occ_vec[j] − occ_vec[i])
10:            cnt ← cnt + 1
11:            for k := j+1 ..... S do
12:                for l := 0 ..... t do
13:                    diff_1 ← (occ_vec[k] − occ_vec[i] − l)%period
14:                    diff_2 ← (occ_vec[k] − occ_vec[i] + l)%period
15:                    if ( diff_1 equals 0 OR diff_2) equals 0 ) then
16:                        cnt ← cnt + 1
17:                        Break
18:            if (weighted_average(X) ×cnt) ≥σ then
19:                RETURN true
20:    RETURN false
```

3.6 Summary

In this chapter, we have discussed our proposed two techniques *HDTS* and *MPWS Pruning* along with pseudo code and simulations. We have also provided an application where our solution will outperform existing approaches. In the next chapter we will show the performance of our algorithms by running in various real life data sets and we will also provide a comparative analysis between our solution and existing solutions.

Chapter 4

Performance Evaluation

4.1 Introduction

In this chapter, we show performance analysis of our proposed algorithms over several datasets. Our proposed algorithm *HDTS* handles dynamicity(insertion and deletion events in the current suffix tree) in time series and *MPWS Pruning* prunes generation of candidate patterns using a heuristic named maximum possible weighted support. Performance of *HDTS* is compared with reconstruction of the suffix tree as there is no known technique. *MPWS Pruning* is compared to *MaxW Pruning*. We are providing the experimental results of *HDTS* and *MPWS Pruning* under various performance metrics.

4.2 Source of Datasets

We have ran our program on four real life datasets all taken from UCI Machine Learning Repository [1]. These are *Absenteeism at work Data Set*, *Appliances energy prediction Data Set*, *Individual household electric power consumption Data Set* and *Diabetes Data Set*. The datasets were discretized to turn them into stream of characters.

4.3 Environment Setup for Experiment

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.

4.4 Performance Analysis of *HDTS*

In sliding window system, when new batch of data appears inserting them into the current data structure is a O(Size of a batch) operation. But there is no known efficient technique of updating the suffix tree structure(both deletion and insertion in same framework) in time series. Rebuilding is the only way. Our proposed algorithm *HDTS* solves the problem by avoiding reconstruction. The performance of *HDTS* will be compared with reconstruction in this section.

We have converted our data(A string of length N) into into data streams in the following manner. We get window i from window $(i-1)$ by removing the first 50 characters from the (i-1)window and adding 50 new characters at the end of the same window. Basically we have divided the data into several batches each containing 50 characters. There are $\frac{N}{50}$ batches in the data set where N is the number of characters in the data set. If the window size is X, then each window consists of $\frac{X}{50}$ batches.

The figures in this section only show the analysis of the first 40 windows.

4.4.1 Runtime With Varying Window Size

We will analyze the performance of our algorithm with varying window size. When window size is large, reconstruction for every window will take big amount of time. Our algorithm does any insertion and deletion in O(Number of Modified Symbols) time.

In figure 4.1 window size is 100 and in figure 4.2 window size is 1000. We see that time taken by reconstruction is almost similar to *HDTS* in these cases. But in figure 4.3 window size is 10000 and in figure 4.4 window size is 30000.

As the window size increases more, our algorithm runs much much faster than reconstruction. So, in cases where window size is large, our proposed algorithm *HDTS* has a very useful application. Main purpose of our solution was how to perform well in cases like frequent modification events occurring or having large data streams to handle. If we have to rebuild the structure frequently then definitely performance will get worsen, but as our solution dynamically updates the structure performance improves to a huge extent. Figure 4.2, 4.3 and 4.4 are shown in the following pages.

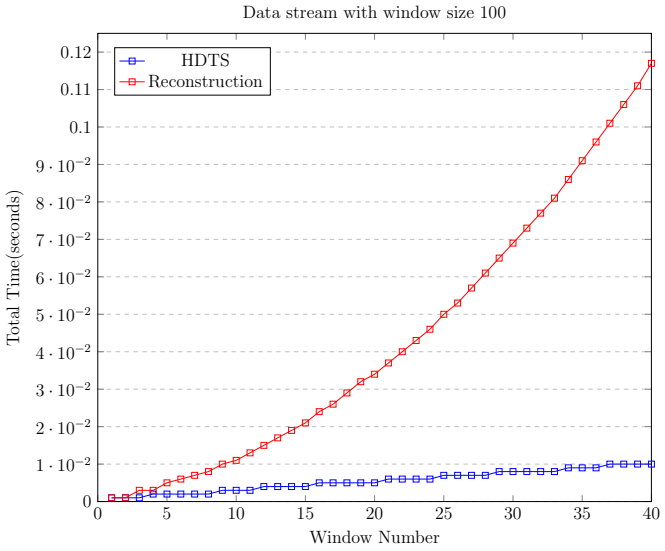

FIGURE 4.1: Runtime With Window Size 100 in Individual Household Electric Power Consumption Data Set.

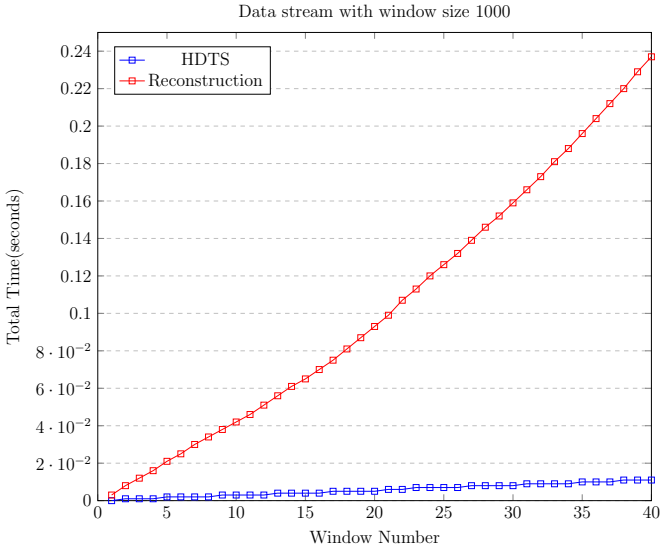

FIGURE 4.2: Runtime With Window Size 1000 in Individual Household Electric Power Consumption Data Set.

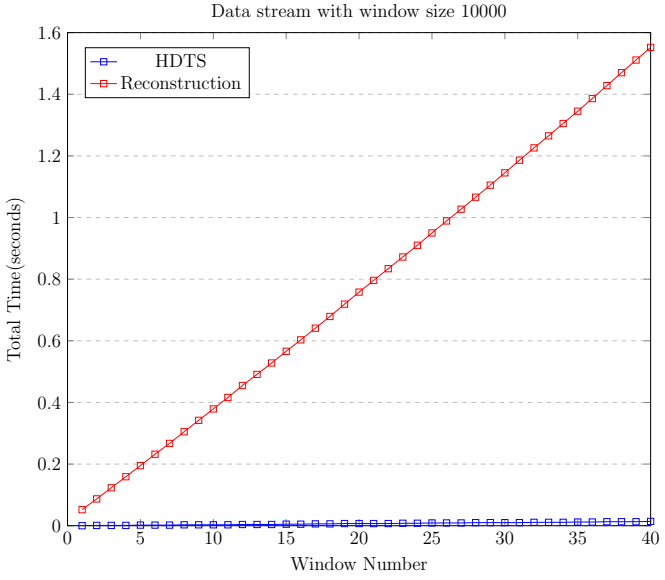

FIGURE 4.3: Runtime With Window Size 10000 in Individual Household Electric Power Consumption Data Set.

4.5 Performance Analysis of *MPWS Pruning*

The performance analysis of *MPWS Pruning* over several performance metrics is as follows.

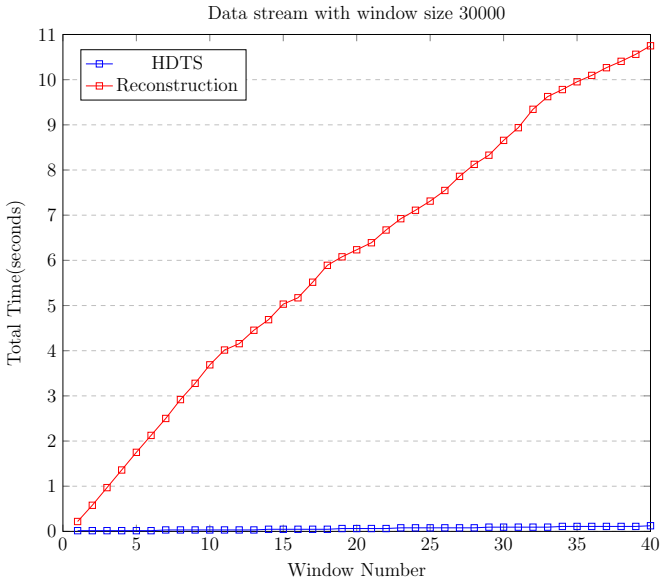

FIGURE 4.4: Runtime With Window Size 30000 in Individual Household Electric Power Consumption Data Set.

4.5.1 Number of Patterns Tested with Varying min_sup

Number of patterns tested varies with varying minimum weighted support threshold value. The number of patterns tested increases when the *min_sup* value is increased. But our goal is to show the difference between two pruning techniques. So, the difference in number of patterns tested is the main concern here.

In brute force candidate generation process, one would test every pattern if it can be a candidate. But this is not acceptable as there can be many patterns. Our main goal in *MPWS Pruning* is to avoid testing patterns that will not become a candidate pattern eventually.

For all the 4 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 datasets. From fig. 4.5, Fig. 4.6, Fig. 4.7 and Fig. 4.8 we see that *MPWS Pruning* tests much fewer patterns compared to *MaxW Pruning*.

For example, in Fig. 4.5, when the minimum support threshold is 0.005%, if we try to optimize the candidate generation process using only MaxW in the

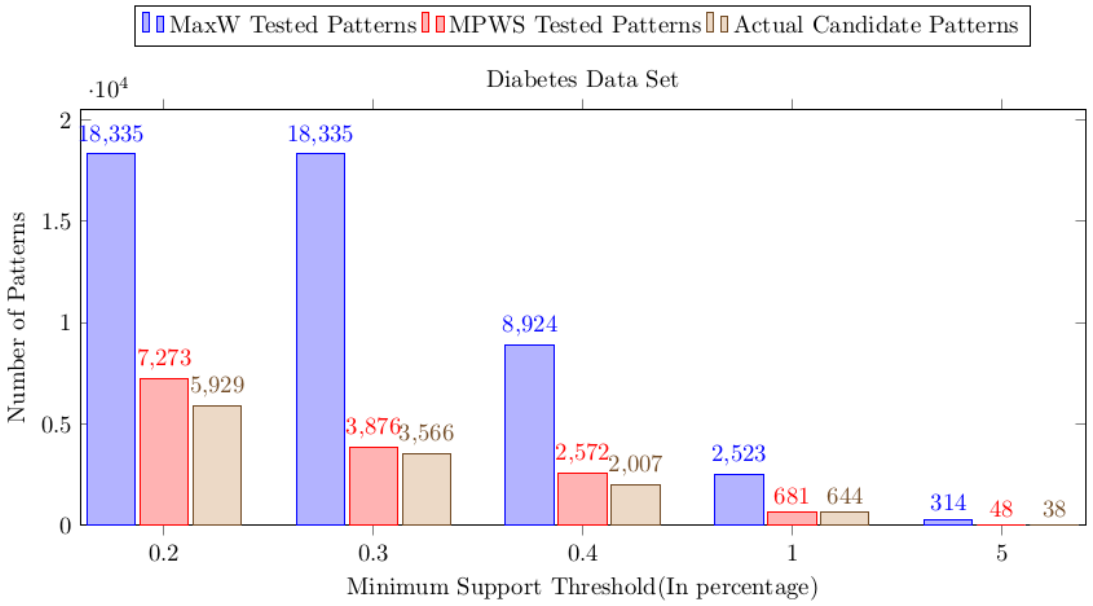

FIGURE 4.5: Number of Patterns Tested With Varying min_sup in Diabetes Data Set.

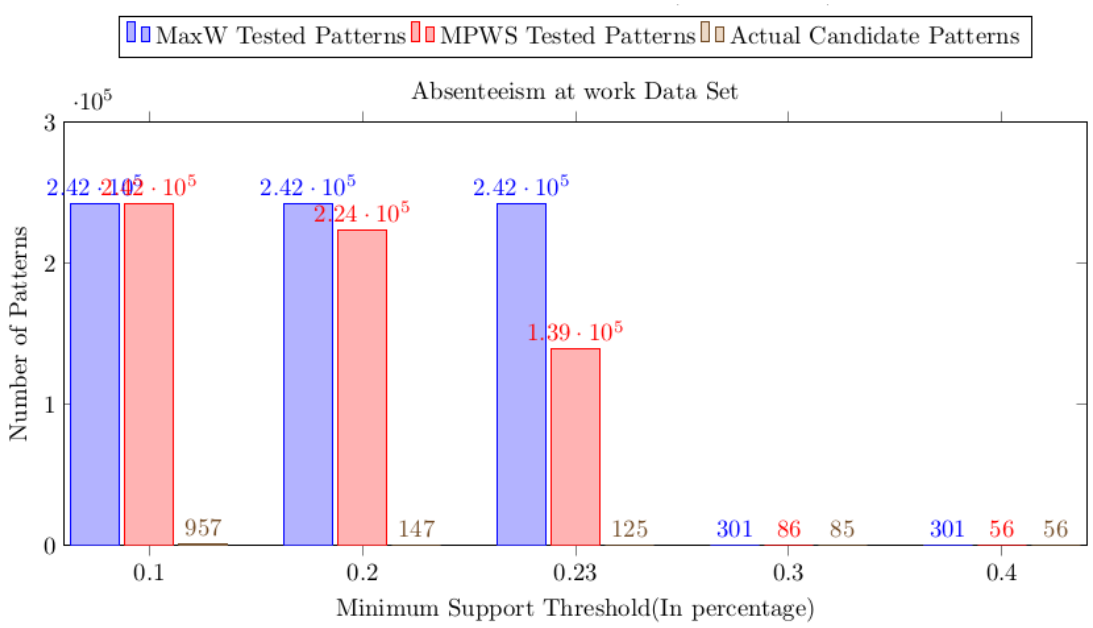

FIGURE 4.6: Number of Patterns Tested With Varying min_sup in Absenteeism at Work Data Set.

database, it checks 63490 patterns whereas *MPWS Pruning* checks 21592 patterns only. Eventually, 21408 patterns become candidate patterns. *MPWS Pruning*

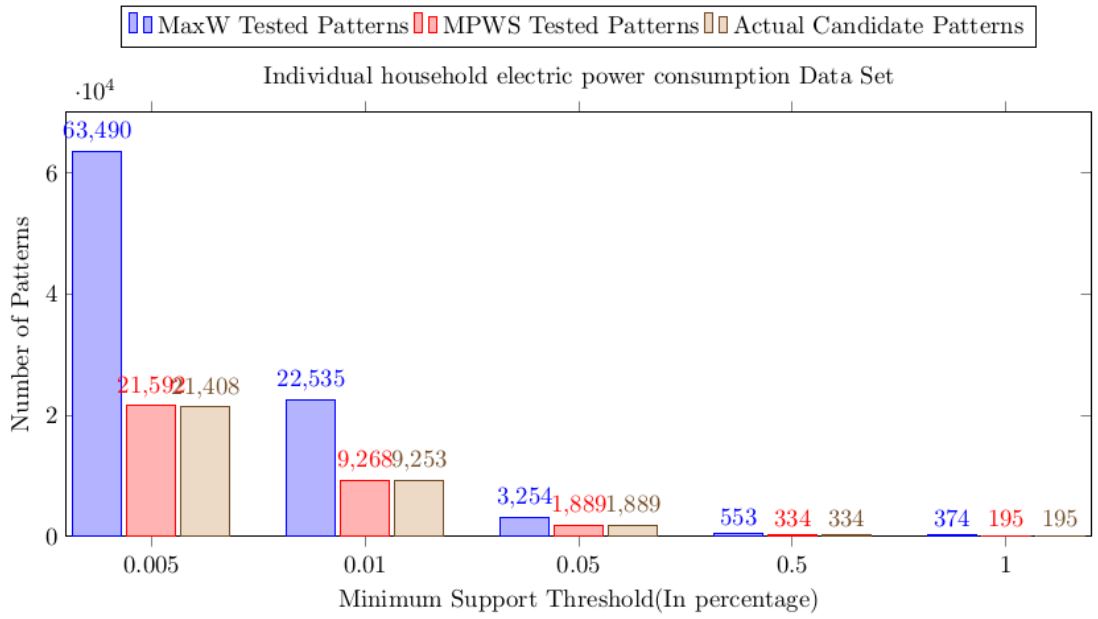

FIGURE 4.7: Number of Patterns Tested With Varying min_sup in Individual Household Electric Power Consumption Data Set.

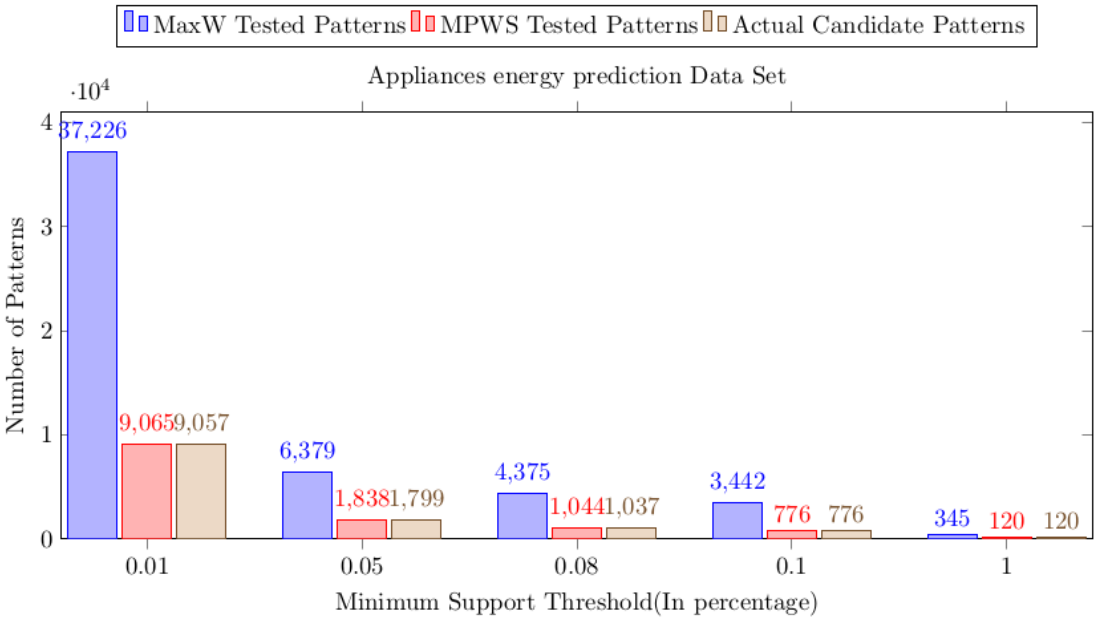

FIGURE 4.8: Number of Patterns Tested With Varying min_sup in Appliances Energy Prediction Data Set.

prunes the testing of almost all the unnecessary patterns.

Let N be the current number of characters in the suffix tree. The beauty

of MPWS Pruning is that all the pre processing needed for it can be done in O(number of nodes in the tree) and the number of nodes in the tree is proportional to N.

4.5.2 Runtime with Varying min_sup

With the decrease of *min_sup*, more patterns will be generated thus more time will be needed for the program to finish. So, the runtime will decrease when min_sup is increased.

For example, in Individual Household Electric Power Consumption Data Set, there are 21408 candidate patterns when min_sup is 0.005%. But when min_sup is 0.01%, the number of candidate patterns is only 9523. We can see in Fig. 4.5 that number of patterns tested for candidacy also decreases when min_sup increases thus resulting in a reduced runtime.

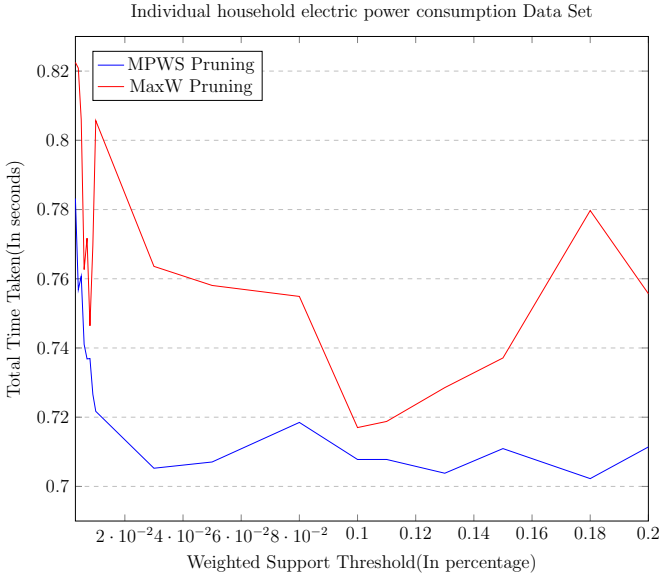

FIGURE 4.9: Runtime With Varying min_sup in Individual Household Electric Power Consumption Data Set.

Fig. 4.9, Fig. 4.10, Fig. 4.11 and Fig. 4.12 show the correspondence between runtime and min_sup. These figures also compare the runtime of *MPWS Pruning*

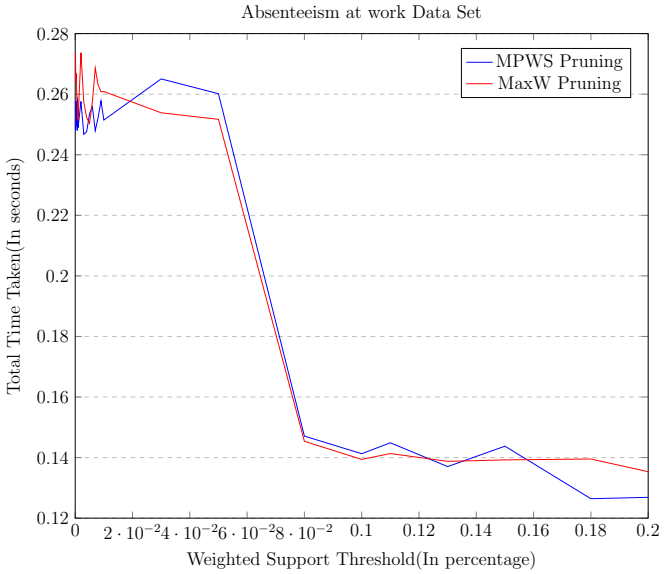

FIGURE 4.10: Runtime With Varying min_sup in Absenteeism at Work Data Set.

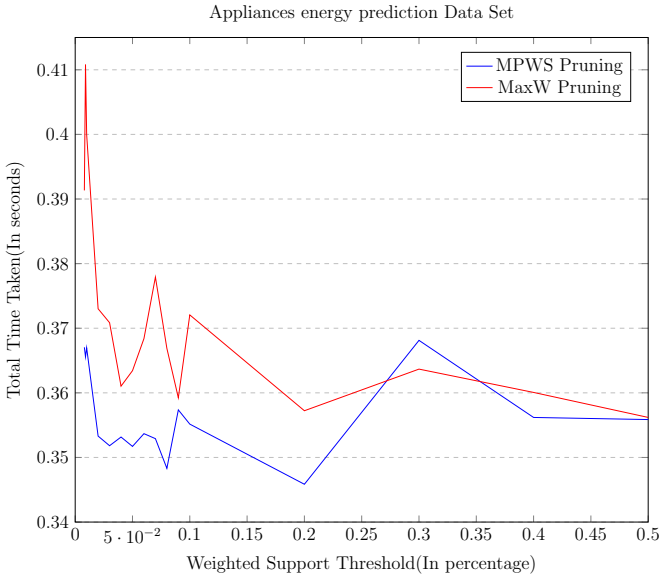

FIGURE 4.11: Runtime With Varying min_sup in Appliances Energy Prediction Data Set.

and *MaxW Pruning*. It's easily observable that when min_sup is low, the difference in runtime between *MPWS Pruning* and *MaxW Pruning* is significant but when min_sup is increased, the runtime of both of the techniques become similar. Because, the difference between runtime of the two techniques depends on the difference in number of patterns tested. When min_sup increases, the difference

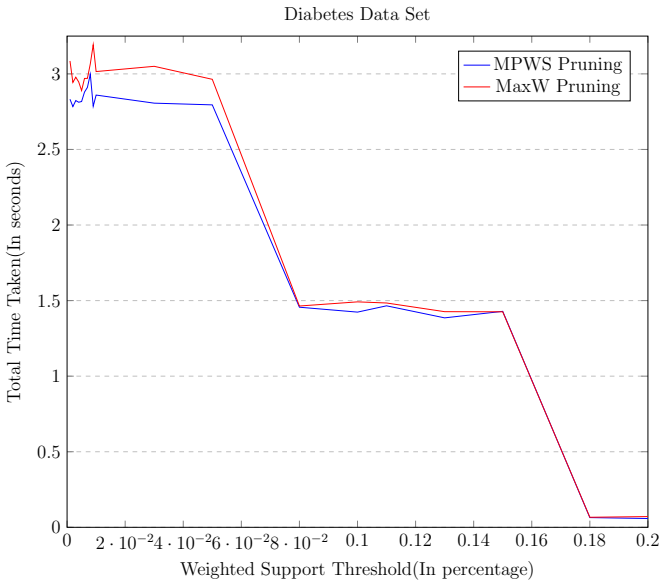

FIGURE 4.12: Runtime With Varying min_sup in Diabetes Data Set.

in the number of patterns tested becomes negligible.

4.5.3 Memory Usage with Varying min_sup

To analyse the memory usage of *MPWS Pruning*, we have compared memory usage of this technique with *MaxW Pruning*. Memory usage means maximum amount of memory the program uses at any given moment.

Fig. 4.13,Fig. 4.14,Fig. 4.15 and Fig. 4.16 show the comparison between MPWS and *MaxW Pruning*. The figures tells us that the memory usage of both the algorithms are almost similar. When, min_sup is low, MPWS performs a bit better than *MaxW Pruning*. As we have use a breadth first technique to generate patterns, the maximum size of the queue is the biggest factor here. When min_sup is increased, number of patterns tested decreases and thus less memory is used. *MaxW Pruning* tests more number of patterns than MPWS Pruing. So, more space is needed in the queue thus resulting in more memory used.

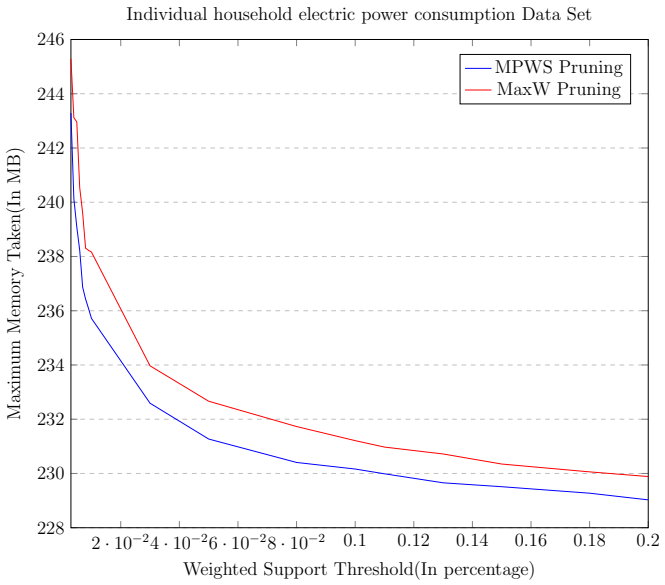

FIGURE 4.13: Memory Usage With Varying min_sup in Individual Household Electric Power Consumption Data Set.

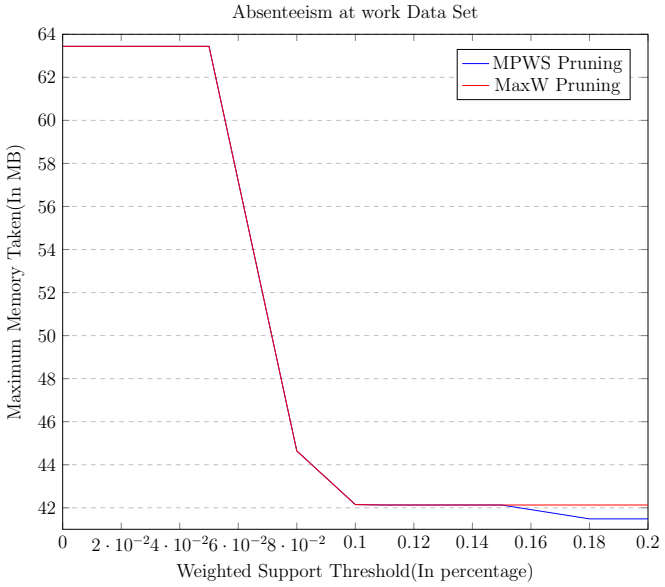

FIGURE 4.14: Memory Usage With Varying min_sup in Absenteeism at Work Data Set.

4.6 Feasibility Analysis

When it comes to feasibility of HDTS, being able to build the suffix tree for the current window in consideration is enough. A suffix tree takes only $\mathcal{O}(L)$ memory where L is the length of the window. As the complexity of deleting first

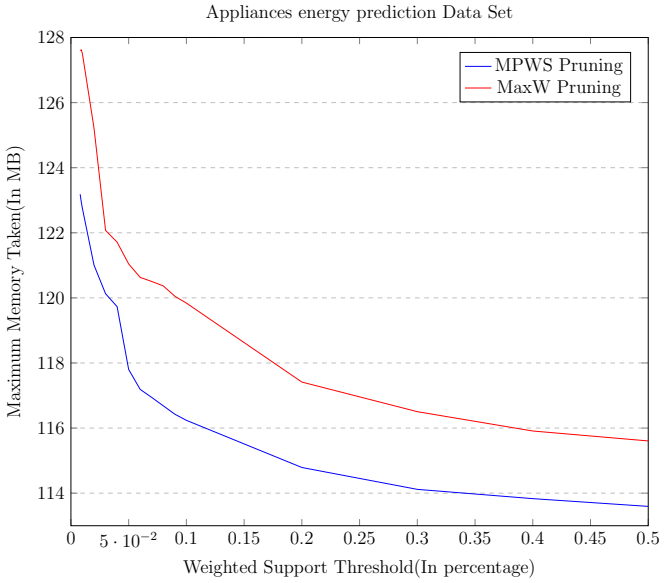

FIGURE 4.15: Memory Usage With Varying min_sup in Appliances Energy Prediction Data Set.

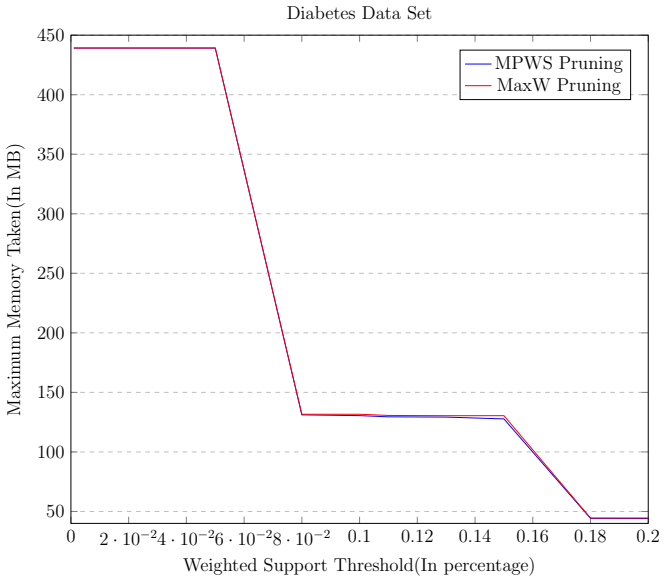

FIGURE 4.16: Memory Usage With Varying min_sup in Diabetes Data Set.

P characters in $\mathcal{O}(P)$ and inserting P new characters is also $\mathcal{O}(P)$, that's obviously better than building the whole tree again for each window and memory usage of HDTS is also $\mathcal{O}(N)$. So, there is no exponential growth in memory or runtime.

MPWS Pruning does a pre calculation for to prune as many subtrees as possible. To calculate the necessary values for the pruning, a depth first search is

needed. In the depth first search, a queries for maximum weight on some edge is done. Eache query is answered in $\mathcal{O}(1)$ complexity. Update of the data structure for edge weight query takes $\mathcal{O}(log(L))$ time. Which is not much of a problem for practical use of the whole structure.

4.7 Summary

The analysis of the whole chapter suggests that *HDTS* always performs better than reconstruction technique. *HDTS* can be useful specially in mining important patterns from time series where the window size is large and frequent modifications occurring.

There is no trivial downward closure property in weighted pattern mining. *MaxW Pruning* is used to optimize candidate generation. Our proposed *MPWS Pruning* always tests lesser number of patterns that *MaxW Pruning* technique which is also time and memory efficient.

Chapter 5

Conclusions

This section will include an overall discussion regarding our work. Our working domain was time series data stream. Time series is a very widely addressed topic under sequential pattern mining and has a huge number of real-life applications. In this study, we addressed two very important problems in dynamic time series data streams and proposed our solutions.

5.1 Summary of Research

The first problem which we addressed in this study is directly related to using sliding window protocol in time series data streams. The main challenge in that is to keep the underlying data structure consistent for the current window. Current works suggest reconstruction of the data structure for each new window which performs poorly when the window size is big.

To solve this problem, we proposed a technique *HDTS* which focuses on dynamically updating the tree in lieu of reconstruction each time. So, as we enforce on modification of the tree rather than reconstruction, our solution performs comparatively way better when we have to handle data streams having large window size. Our experimental analysis section also validate this statement where we

have shown our *tree modification technique* surpasses *tree reconstruction* in performance. Another important feature of our solution is, *its structure*. The core problem which we had to solve was, how to dynamically update the suffix tree in case of new insertion and deletion events [1]. So, our solution can fit in any other problem which uses suffix tree as the underlying data structure. Another important point to note is that this solution is connected to structure modification and have no consideration regarding special frameworks like weighted or unweighted. So, this solution is framework independent and can be applied to both weighted and unweighted versions.

The second problem which we address in this study is directly connected with weighted pattern mining. A traditional and very popular approach in weighted pattern mining to apply downward closure property is using *Max-Weight* or alike concepts as pruning technique. Main cliche part in any weighted pattern mining is, how to prune speed up the candidate generation process by using better heuristics. In our work, we propose a compact pruning technique, *MPWS Pruning* which provides better and more accurate heuristic than existing max-weight alike concepts. Our pruning technique gives better assumption regarding possible candidates and reduces pattern testing for candidacy. After running our pruning technique in various real-life data sets, we have also found the validity of our claim and provided a detailed analysis in our experimental analysis section.

Another important point to observe here is, as our working domain was time series, the whole weighted fragment has been discussed upon mining weighted sequential patterns from time series datastreams. But our proposed pruning technique can also be applied to other types of weighted pattern mining as well because the ideology lies in how we are approximating the possible candidates and reducing the testing. So, our solution framework can have other appliances like periodic pattern mining too.

[1] according to the definition of time series datastream

Time series domain has a vast range of applications in our day to day life. There are numerous fields like Weather Forecasting, Market Transaction Analysis, Match Performance Analysis, Symptom Observation to Detect Disease, Banking Scheme Decision etc. which closely represents time series applications. As dynamicity is the inherent characteristic of datastream, we believe our first solution *HDTS* can play a vital role in improving the latency of existing approaches. Ss weighted pattern mining in time series plays a very mesmerizing role by simulating characteristics scoped over a large time domain and our second technique *MPWS Pruning* plays a significant role to improve performance, we believe our contributions will help to ease many existing deficiencies in time series and will pave the way for solutions to many existing problems.

5.2 Future Work

In the previous section, we discussed the possibility of expanding our solutions to various domains and scenarios. While working, we discovered many interesting problems. One of the problems was about the introduction of *dynamic weights* in time series which is still undiscovered. In the modern world, the importance of an item varies with time and the same product can contain different amount of significance in a long range of time and this characteristic should be reflected while mining patterns. We believe our solutions are capable to solve the above-mentioned problem and we are planning to extend our work in this area in near future.

Bibliography

[1] Dua, D. and Karra Taniskidou, E. (2017). UCI Machine Learning Repository [http://archive.ics.uci.edu/ml]. Irvine, CA: University of California, School of Information and Computer Science.

[2] Agrawal, R., & Srikant, R. (1994). Fast Algorithms for Mining Association Rules in Large Databases. *Proceedings of the 20th International Conference on Very Large Data Bases, 487-499.*

[3] Zaki, M. J. (2001). SPADE: An Efficient Algorithm for Mining Frequent Sequences. Machine Learning, 42 (1-2), 31-60. doi:10.1023/A:1007652502315

[4] Han, J., Pei, J., & Yin, Y. (2000). Mining Frequent Patterns Without Candidate Generation. *ACM SIGMOD Record, 29*(2), 1-12. doi:10.1145/335191.335372

[5] Agrawal, R., & Srikant, R. (1995). Mining Sequential Patterns. *Proceedings of the Eleventh International Conference on Data Engineering.* doi:10.1109/icde.1995.380415

[6] Aggarwal, C. C., & Han, J. (2014). *Frequent Pattern Mining.* Cham: Springer International Publishing.

[7] Lee, C., Lin, C., & Chen, M. (2005). Sliding Window Filtering: An Efficient Method for Incremental Mining on a Time-variant Database. *Information Systems, 30*(3), 227-244. doi:10.1016/j.is.2004.02.001

[8] Srikant, R., & Agrawal, R. (1996). Mining Sequential Patterns: Generalizations and Performance Improvements. *Advances in Database Technology EDBT '96 Lecture Notes in Computer Science*, 1-17. doi:10.1007/bfb0014140

[9] Han, J., Pei, J., Mortazavi-Asl, B., Chen, Q., Dayal, U., & Hsu, M. (2000). FreeSpan. *Proceedings of the Sixth ACM SIGKDD International Conference on Knowledge Discovery and Data Mining - KDD '00.* doi:10.1145/347090.347167

[10] Pei, J., Han, J., Mortazavi-Asl, B., & Pinto, H. (2001). PrefixSpan: Mining Sequential Patterns Efficiently by Prefix-Projected Pattern Growth. *ICDE '01 Proceedings of the 17th International Conference on Data Engineering*, 215.

[11] Rasheed, F., Alshalalfa, M., Alhajj, R., Efficient Periodicity Mining in Time Series Databases Using Suffix Trees, *IEEE Transactions on Knowledge and Data Engineering, Volume: 23, Issue: 1, Jan. 2011, pp 79 - 94.*

[12] Leung, C.K.-S, Khan,Q.I., DSTree: A Tree Structure for the Mining of Frequent Sets from Data Streams, *In proc. of ICDM'06, 18 - 22 Dec. 2006, Pages 928 - 932.*

[13] Ahmed, C.F, Tanbeer, S.K., Jeong, B. -S, Lee, An Efficient Algorithm for Sliding Window-Based Weighted Frequent Pattern Mining over Data Streams, *IEICE Transactions on Information and Systems,92 - D(7): 1369 - 1381, July.*

[14] Ahmed, C.F.,Tanbeer, S.K., Jeong,B.-S., Handling Dynamic Weights in Weighted Frequent Pattern Mining, *IEICE - Transactions on Information and Systems, Volume E91-D, Issue 11, November 2008, Pages 2578 - 2588.*

[15] Ukkonen, E., Algorithmica (1995) 14: 249. On-line construction of suffix trees, Algorithmica, September 1995, Volume 14, Issue 3, pp 249260.

[16] Tari, Z. et. al., MicroGRID: An Accurate and Efficient RealTime Stream Data Clustering with Noise, *In Proc. of PAKDD 2018,pp 483 - 494.*

[17] Almusallam, N. et. al., UFSSF: An Efficient Unsupervised Feature Selection for Streaming Features. *In Proc. of PAKDD 2018,pp 495 - 507.*

[18] Chanda, A. K., Saha, S., Nishi, M. K., Samiullah, M., Ahmed, C. F., An Efficient Approach to mine Flexible Periodic Patterns in Time Series Databases, Engineering Applications of Artificial Intelligence, Volume 44, September 2015, Pages 46-63.

[19] Chanda, A. K., Ahmed, C. F., Samiullah, M., Leung, C.K.-S, A New Framework for mining Weighted Periodic Patterns in Time Series Databases, Expert Systems With Applications, Volume 79, 15 August 2017, Pages 207-224.

[20] Yun, U. (2008). A New Framework for Detecting Weighted Sequential Patterns in Large Sequence Databases. *Knowledge-Based Systems, 21*(2), 110-122. doi:10.1016/j.knosys.2007.04.002

[21] Cui, W., & An, H. (2009). Discovering Interesting Sequential Pattern in Large Sequence Database. *2009 Asia-Pacific Conference on Computational Intelligence and Industrial Applications (PACIIA).* doi:10.1109/paciia.2009.5406655

[22] Kollu, A. (2013). Incremental Mining of Sequential Patterns Using Weights. *IOSR Journal of Computer Engineering, 14*(5), 70-73. doi:10.9790/0661-1457073

[23] Garofalakis, M., Rastogi, R., & Shim, K. (2002). Mining Sequential Patterns with Regular Expression Constraints. *IEEE Transactions on Knowledge and Data Engineering, 14*(3), 530-552. doi:10.1109/tkde.2002.1000341

[24] Han, J., Dong, G., & Yin, Y. (1999). Efficient Mining of Partial Periodic Patterns in Time Series Database. *Proceedings 15th International Conference on Data Engineering (Cat. No.99CB36337).* doi:10.1109/icde.1999.754913

[25] Mannila, H., Toivonen, H. & Verkamo, A. I. (1997). Discovery of Frequent Episodes in Event Sequences. *Data Mining and Knowledge Discovery, 1*, September 1997, Volume 1, Issue 3, pp 259 - 289.

[26] Bing, L. (2011). Association Rules and Sequential Patterns. *Web Data Mining*, 37-47.

[27] Elfeky, M.G., Aref, W.G., Elmagarmid, A.K., Periodicity Detection in Time Series Databases, *IEEE Trans. Knowledge and Data Eng., vol. 17, no. 7, pp. 875-887, July 2005.*

[28] Gaber, M. M., Zaslavsky, A., Krishnaswamy, S., Mining Data Streams: A Review, *ACM SIGMOD Record, Vol. 34, No. 2, June 2005, pp. 18 - 26.*

[29] Chi, Y. et al., Moment: maintaining closed frequent itemsets over a stream sliding window. *In Proc. ICDM 2004, pp. 5966.*

List of Notations

- *node.child* - A list which saves the child nodes' reference of *node*.
- *node.suffixLink* - the reference of the node which is directed as suffix link of *node*.
- *node.start* - start index (from input) of the string represented by the edge that connects the *node* to it's parent.
- *node.end* - integer reference of the string end represented by a node.
- *node.nodeW* - the best possible weighted support of a pettern ending the edge that connects the *node* to it's parent.
- *node.mpws* - the best possible nodeW in the subtree of node.
- *min_sup* - minimum weighted support threshold(In Percentage).
- cnt(A,B) - number of characters between node A and node B.
- maxW(A,B) - max weight in the edge between node A and node B.
- edge(A,B) - represents the edge between node A and node B.