Noname manuscript No.
(will be inserted by the editor)

A New Tree-Based Approach to Mine Sequential Patterns

Redwan Ahmed Rizvee · Chowdhury Farhan Ahmed · Md. Fahim Arefin · Carson K. Leung

the date of receipt and acceptance should be inserted later

Abstract Sequential Pattern Mining is a popular research domain due to its increasing range of applications. In this article, we have introduced a new tree-based solution to the sequential pattern mining problem, including two sets of novel solutions for static and incremental sequential databases. We have proposed two new structures, *SP-Tree* and *IncSP-Tree* and designed two efficient algorithms, *Tree-Miner* and *IncTree-Miner* to mine the complete set of sequential patterns from static and incremental databases. The proposed novel structures provide an efficient manner to store the complete sequential database maintaining "build-once-mine-many" property and give scope to perform interactive mining. Additionally a new breath-first based support counting technique has been designed to efficiently identify the infrequent patterns at the early stage and a new heuristic pruning strategy to reduce pattern search space. We have also designed a new pattern storage structure *BPFSP-Tree* to store the frequent patterns during successive iterations in incremental mining to reduce the number of database scans and to remove the infrequent patterns efficiently. A novel structure named Sequence Summarizer is also introduced to efficiently calculate and update the co-occurrence information of the items, especially in an incremental environment. Experimental results over various real-life and synthetic datasets demonstrate the efficiency of our work in comparison with the related state-of-the-art approaches.

Keywords Sequential Pattern · Tree-based mining · Incremental mining · Breadth-first based pruning · Pattern Storage

Redwan Ahmed Rizvee, Chowdhury Farhan Ahmed, Md. Fahim Arefin
Department of Computer Science and Engineering
University of Dhaka, Bangladesh
E-mail: rizveeredwan.csedu@gmail.com, farhan@du.ac.bd, f.arefin8@gmail.com

Carson K. Leung
Department of Computer Science
University of Manitoba, Canada
E-mail: kleung@cs.umanitoba.ca

1 Introduction

The idea of pattern mining problem was introduced to discover interesting characteristics or behavior from the database. Due to the wide variation of database characteristics pattern mining problem has been divided into numerous sub-domains, among which Sequential Pattern Mining (SPM) problem stands out because of its wide range of variations. SPM problem targets to discover frequent sequential patterns from an ordered or sequential database. In Table 1, we have shown an example of a camera market sequential database where each entry denotes a customer's purchase history. Items within same bracket denotes a transaction for that customer. So, a record is a collection of ordered transactions for that customer. SPM problem will try to discover different types of ordered relationships from this dataset, such as the generic SPM problem will try to discover the ordered item clusters which are frequently purchased. Being first introduced in [33] based on market basket analogy, the SPM problem has found its usage in numerous applications, e.g., web usage mining, customer behavior analysis, DNA sequence mining, etc.

Due to having numerous applications, a wide range of literature has addressed SPM problem and provided solutions which can be broadly categorized into two groups: apriori based and pattern growth based. Apriori based approaches follow candidate generation and

testing paradigm and pattern growth approaches follow the projected database or database shrink concept with pattern's gradual extension. Pattern growth approaches are significantly faster compared to apriori approach [2]. In traditional itemset mining it has been shown that tree structure alike approaches provide more control over the database which ultimately helps improve the mining runtime[20,2] and incorporate new strategies. But due to the problem complexity, the tree alike structures of the itemset mining were not suitable for SPM problem and a structural solution was yet to be proposed. Based on this motivation, in this study, we have proposed a novel tree structure *SP-Tree* to represent the sequential database in a structured format and an efficient mining algorithm *Tree-Miner* to mine sequential patterns using the node properties of *SP-Tree*. The advantage of the proposed tree structure is, it provides efficient structured control over the database and pattern space which ultimately leads to a faster generation of the patterns. An important motivation behind designing such a structure was, if we had a structured format of the database, it would have given the advantage to adopt new pruning strategies and control the manipulation of the database when the database can change based on other parameters, e.g, incremental database, stream database, etc. Our incremental solution is the result of prior motivation.

Generic SPM problem focuses on mining frequent sequential patterns from the static sequential databases [30,33,36,28,12,4,29,26,16]. But in real-life applications, most of the time, the database is found not to be static; rather gets increased time to time with more information [24]. With increased database size, patterns' distribution can vary significantly which necessitates the urge to mine again over the updated complete database. But, mining sequential patterns over the complete database is a very costly operation. So re-mining over the updated database again from scratch creates several performance and resource bottleneck. From these motivations the problem of Incremental Sequential Pattern Mining (ISPM) was introduced in [34] to tackle the challenge of re-mining over the complete database rather than focusing more over the efficient handling of the new incremental database or increased part of the database [24,32,13,19].

In Table 2, we have shown an example of incremental version of our prior static database shown in Table 1. Here, two sequences newly appeared. First one as an appended sequence to the existing sequence with identifier (*sid*) 1 and the fifth one is an inserted sequence with a new *sid*(5) which increases database length. ϕ means there was no new sequence as appended or inserted for the corresponding *sid*. Here the updated incremental database will be the concatenation of two databases. The solutions to the ISPM problems focus on developing new strategies to efficiently discover the complete set of updated frequent patterns from this modified database rather than re-mining from scratch.

There are several crucial challenges in ISPM problems due to the problem's nature. For example - handling the modification of existing sequences (*Append*), the addition of completely new appearing sequences (*Insert*), ratio of the incremental database vs existing database, updating the existing data structures, the change in the frequent patterns' distribution or concept drift, empirically setting the extra introduced parameters to control the candidate buffers [6,9] etc. In summary, several issues exist which control the efficiency, applicability and complexity of the solution to approach the ISPM problems and thus numerous literature have addressed this problem [21,6,23,9,10]. In this study, we have also proposed a new tree-based solution, *IncTree-Miner* based on *IncSP-Tree* to approach the ISPM problem which provides an efficient manner and structural advantage to implicitly track the incremental database and the patterns which are affected by it.

The usage of co-occurrence information can significantly reduce the search space which has been discussed in many literature[12,13,10]. Our solutions have also adopted this concept. This information states the relationship among the items which guides during pattern extension. But it becomes a challenge on how to efficiently update this information for a gradually increasing database. To solve this issue, we have proposed a novel structure *Sequence Summarizer* which helps calculate such information efficiently especially in an incremental environment. This novel structure is also helpful to perform the *Append* operation over the existing sequences.

The main challenges of any SPM problem are reducing the number of database (DB) scans, making the DB scans faster, reducing the search space, and detecting the infrequent patterns early during support calculation. In our proposed structures we have introduced the idea of *next_link* which makes the DB scans significantly faster. Also utilizing the tree properties we have developed two new pruning strategies: a breadth-first based support counting technique which helps detect the infrequent patterns early before calculating complete support and a heuristic strategy to reduce the search space.

As, with database increment, the support of the patterns gets updated in each iteration, popular literature maintain a tree alike structure to keep the patterns' support [5,23,9]. In this study, we further investigate

sid	Sequence
1	<(camera, kit lens) (50 mm prime lens) (tripod) >
2	<(camera, kit lens) (85 mm prime lens) (tripod) >
3	<(camera, kit lens) (tripod) >
4	<(camera, kit lens) (50 mm prime lens) (85 mm prime lens) >

Table 1: Initial Camera Market Dataset

sid	New Sequence	Type
1	<(ND Filter, Reverse Ring) >	Append
2	ϕ	-
3	ϕ	-
4	ϕ	-
5	<(camera, kit lens) >	Insert

Table 2: Additional update in Database

this approach and design a new Bi-directional Projection Pointer Based Frequent Sequential Pattern Tree (*BPFSP-Tree*) which keeps the frequent sequential patterns, their support, and projection pointers using the node structure of *IncSP-Tree*. It helps reduce the number of DB scans and provides an efficient mechanism to remove the non-frequent patterns which were previously frequent.

Because of various crucialities of the ISPM problem, different literature have adopted different types of strategies to solve them. Among them, many introduced additional parameters or concepts such as negative border [38], semi-buffer [6], pre-large with upper and lower thresholds [9] etc. The main problem of additional parameters is that the solution's performance and complexity largely depend on the appropriate selection of these parameters and their mutual dependency and it is difficult to estimate database characteristics prior. Also, these approaches are severely affected due to concept drift and create resource misuse and bottleneck. Based on these observations, we wanted to reduce the parameters' dependency and stick to the single traditional support threshold parameter. Besides, our approach is a new take to generic SPM problem. So, it is also flexible to other modules. Moreover, our proposed structure stores the complete database in an efficient format. So, it is also able to handle the absence of prior database in stream mining and runtime threshold parameter change. In summary, our main contributions are as follows -

1. We have proposed two new tree-based solutions, an efficient *Tree-Miner* algorithm based on a novel tree structure *SP-Tree* and an efficient *IncTree-Miner* algorithm based on a novel tree structure *IncSP-Tree* to solve the SPM problem for static and incremental databases, respectively.
2. Based on the tree properties, we have designed two new pruning strategies: an efficient breadth-first based support counting technique and and a heuristic pruning strategy.
3. A novel structure *Sequence Summarizer* is proposed to efficiently calculate and update the co-occurrence information and perform *Append* operation during database increment.
4. A new structure *BPFSP-Tree* to store the frequent sequences along with projection pointers to reduce the DB scan and efficiently remove the infrequent patterns.
5. A Discussion regarding efficiency and effectiveness of our proposed solutions and issues related to implementation.

The preliminary version of the current study has been published in [30]. These two literature focuses on designing new tree-based solutions to approach the SPM problem by providing a new viewing angle. The prior study proposed a tree-based solution to solve static SPM problem while the current literature has added the following new materials,

- More detailed and comprehensive discussion with additional examples to discuss *Tree-Miner* algorithm based on SP-Tree to solve the static SPM problem.
- Addition of two new pruning strategies based on SP-Tree structures.
- A novel approach to solve the incremental SPM problem with a new incremental mining algorithm *IncTree-Miner* over an extended SP-Tree structure, *IncSP-Tree* with additional discussion about various crucial aspects related to implementation and incremental environment.
- More detailed and extensive experimental results to discuss the novelty of the proposals along with more examples, analysis and discussion.

The rest of the paper is organized as follows. Related works are discussed in Section 2. We formulate our addressed problem in section 3 and discuss our proposals in Section 4. In Section 5, we evaluate our solutions based on various metrics by conducting experiments on both real-life and synthetic datasets and finally we con-

clude this study with an overall summary and the possibility of future extensions in Section 6.

2 Related Work

In this section, we will provide a short discussion regarding the literature related to our problems and investigate the issues we have focused on.

Being a very important problem domain, a good amount of literature has addressed numerous issues related to SPM [13,14] and provided a wide range of solutions, being first introduced in [33]. The proposed approaches mainly fall into two categories, apriori based and pattern growth based. Pattern-growth based approaches are efficient compared to apriori approaches because they use the concept of database's continuous space reduction with the pattern's gradual extension. The main key factors to improve the SPM algorithms' performance lie in faster support calculation during pattern extension and incorporating different pruning strategies[12,28,36,30]. PrefixSpan [28] is one of the most popular and efficient techniques to solve SPM problem which proposed the idea of database projection. SPAM[1] used bit based representation to calculate patterns' support incorporating mutual dependence based search space pruning technique. FAST[31] improved the support counting technique of SPADE[36] using sparse id-lists. LAPIN[35] showed the importance of the last event's items during pattern extension. CM-SPADE and CM-SPAM[12] are improvements over SPADE and SPAM by incorporating the idea of co-occurrence information.

Up to now, only the literature that proposed new techniques to mine the core support based SPM problem have been briefly discussed. Besides these, there lie a set of literature that have addressed different variations and applications of SPM problems. In [7], they proposed a new type of SPM problem named as Discriminative SPM. There they separated the database as positive and negative class labels and embedded the effect of support variation of the same pattern having two types of class labels also incorporating multiple testing correction in this regard to reduce the number of false positive patterns. In [11], they proposed a hybrid recommendation system for e-learning purpose. In their proposal, they had to discover context aware information in such regard where they used SPM techniques. In the aforementioned articles, both used GSP to generate the sequential patterns and over that they added their contribution. In [8], they addressed constraint-based SPM problems. In such problems, some additional properties and challenges are discussed, e.g, sum, median, max, average utilities etc along with support constraint. They proposed a node based structure in such regard where the nodes hold the rich information in such regard. Their proposed solution could not handle multiple itemed events also they needed minimum support threshold value given during the data structure construction.

Our proposed SP-Tree is a novel tree-based technique to efficiently represent the sequential database. It provides a structural advantage to control the manipulation of the database which ultimately helps to adopt newer pruning strategies and perform faster pattern generation. Our proposed algorithm, Tree-Miner, uses SP-Tree to mine the complete set of sequential patterns. Tree-Miner has adopted all the popular pruning techniques and introduced some newer ones making it a very efficient algorithm.

The problem of mining sequential patterns in the incremental database has also attracted the researchers because of its wide variety of challenges and applications [24,32]. Different literature raised different factors and provided their solution based on it. To solve ISPM problem, one of the earliest solutions was given in [34]. They designed a suffix tree-based solution to approach the problem which maintained the substring w.r.t their address rather than positions. The critical performance issues of this approach were the suffix tree's dependency over the database size and the sensitivity of the position where the update occurs. In ISM [27], the authors adopted a vertical mining approach along with maintaining a negative border information to determine the part of the original database which needs to be scanned again. Negative border included those infrequent sequences whose subsequences were frequent. But the size of the negative border creates a severe memory bottleneck by keeping the information of a huge number of unnecessary patterns. ISE [25] algorithm adopted a level-wise apriori approach for mining and reused the information gathered from the previous passes and had the problem of multiple DB scans. MFS+ and GSP+ were introduced in [37] to provide a solution for the incremental database. The goal of the MFS algorithms is different from us because they targeted to mine maximal frequent sequences.

In [21] IncSP algorithm was proposed. They provided two completely novel ideas for efficient counting and implicit merging and was found to be very efficient but had the problem of level-wise candidate generation and test paradigm because of adopting the apriori approach. In [6], one of the most popular approaches for solving ISPM problem was given, known as IncSpan. It introduced the idea of semi-buffer concept to reduce DB scan and an additional parameter to control the buffer. But, this solution had some critical issues. E.g., the ad-

ditional parameter can control the solution's efficacy along with the complexity to a great extent. Because of keeping a semi-buffer and depending on the parameters, a huge number of infrequent patterns might need to be kept in the main memory resulting in huge memory misutilization. In general cases, a group of patterns slowly become infrequent to frequent and vice versa because in most of the updates, the size of the incremental database is much less compared to the existing database and we wanted to focus on this property and make proper memory utilization to reduce the DB scan. We also wanted to reduce the dependency over the parameters. This approach also induced some additional pre-computation. In PBIncSpan[5], it was shown that IncSpan is not complete because of some wrong conditioning in the algorithm. They corrected the solution and added two new pruning concepts named as width pruning and depth pruning and provided a prefix tree data structure to store the frequent patterns. It was very efficient but still had a good complexity of projecting the database. In ISPBS [23], they proposed a modified version of PBIncSPan. They suggested keeping all the patterns' (frequent and infrequent) information in the tree to reduce DB Scan. The number of patterns in a database can be combinatorially explosive and therefore, keeping all the patterns' information in tree is not practical and the result is far from optimal in the databases of medium to large size.

Being motivated by the FP-Tree[15] structure to solve the itemset mining problem, FUSP-Tree was developed to store frequent sequential patterns. Then the concept of pre-large sequences was discussed in [17,18, 9]. This concept added two new parameters, named as lower threshold and upper threshold. These parameters acted as a buffer, and using them, the FUSP-Tree structure was updated. Pre-large sequences are those sequences that are nearly large but not truly large. These approaches also have a problem similar to buffer like concepts of dependency over multiple parameters, over computation, and concept drift. The algorithm's performance will largely depend on the appropriate selection of the parameters which is difficult to estimate prior and in case of concept drift, for the streaming databases it will cause a good amount of memory misuse and unnecessary pre-computations. Our proposed solution IncTree-miner based on IncSP-Tree enforces more importance over the frequent patterns and their frequency transition properties. As it is based on a single parameter, the solution does not perform unnecessary pre-computation and is not affected due to concept drift. Also, as it holds all the information in a compact manner, it is not affected due to the information absence in stream databases. Researchers also introduced some hybrid and environment based solutions. For example, in [22], the authors provided an apriori and maximal pattern-based approach, in [3], the authors gave a solution to mine closed patterns from the incremental database, in [10] the authors provided a distributed solution based on MapReduce framework, etc. But as our approach discovers all the frequent sequential patterns from the current updated database with their support from a single machine environment, these literature do not match with our key performance issues.

Our proposed IncTree-Miner based on IncSP-Tree is an incremental version of Tree-Miner based on SP-Tree to solve the ISPM problem which develops a set of strategies to efficiently capture the change in pattern space rather than re-mining. Based on our novel tree properties we have also designed two new pruning strategies to efficiently detect the pattern's infrequency and a new pattern storage structure. We have also discussed the supporting summarizer structure to calculate and update the co-occurrence information especially in an incremental environment.

3 Problem Definition

In this section, we will discuss the necessary terms and provide formal definitions of our approached problems.

Let, there be a set of items $I = \{i_1, i_2, i_3, .., i_n\}$. An event or itemset e is a set of items such that $e \subseteq I$. A sequence $s =< e_1, e_2, e_3, .., e_n >$ is a collection of ordered events. A sequence Database D consists of sequences where each sequence, s_i, has an unique identifier aka *sid* ($s_{i_{sid}}$). The support of a pattern (or a sequence) P is the number of sequences in D which contain it. A sequence or pattern P is considered to be frequent if its support, S_P, satisfies a minimum support threshold parameter *min_sup* set by users. In the static database problem, the database is fixed and the formal definition can be stated as follows,

Definition 1 (Sequential Pattern Mining Problem, SPM) Given a sequence database D and a minimum support threshold parameter *min_sup*, discover all the frequent sequences P whose support value satisfy threshold *min_sup* ($S_P \geq min\_sup \times |D|$).

In the incremental mining problem, the database is not fixed. In each iteration a new incremental database *db* is provided and our original or old database D gets updated to $D' = D \cup db$. The main challenge of incremental mining is not to re-mine over the complete database rather to focus only on those patterns which are affected due to the addition of *db*. *db* consists of two types of sequences, $db = D_{Append} \cup D_{Insert}$. D_{Append}

sid	sequence
1	$<(f)(a)(b)(c)(d)(abc)>$
2	$<(ab)(cde)(a)>$
3	$<(abc)(de)>$
4	$<(b)(c)(abc)>$
5	$<(d)(e)(a)>$
6	$<(d)(e)(ab)(c)>$
7	$<(ab)(ce)(ab)>$
8	$<(b)(c)(a)(cd)>$
9	$<(cd)(abc)>$
10	$<(cd)(ef)(ac)>$

Table 3: Static Database

sid	pass 1	pass 2
1	$<(f)(a)(b)(c)(d)>$	$<(abc)>$
2	$<(ab)(cde)(a)>$	$<\phi>$
3	$<(abc)(de)>$	ϕ
4	$<(b)(c)>$	$<(abc)>$
5	$<(d)(e)(a)>$	ϕ
6	$<(d)(e)(ab)(c)>$	$<\phi>$
7	ϕ	$<(ab)(ce)(ab)>$
8	ϕ	$<(b)(c)(a)(cd)>$
9	ϕ	$<(cd)(abc)>$
10	ϕ	$<(cd)(ef)(ac)>$

Table 4: Incremental Database

consists of those sequences whose *sid*s were already present in D, $D_{Append} = \{s|s_{sid} \in D\}$. These sequences do not increase the database length rather increase existing sequences' size because they will be appended at the end to their corresponding sequences. These sequences perform *Append* operation over D. D_{Insert} introduces new sequences in the database and increase its size, $D_{Insert} = \{s|s_{sid} \notin D\}$. These sequences perform $Insert$ operation over D. Our minimum support threshold parameter min_sup will be fixed at the beginning. But, with each increased database size, the minimum support requirement for a pattern to be frequent gets increased. In each pass, we need to mine those sequential patterns, P which have support, $S_P \geq min\_sup \times |D'|$. The formal definition of ISPM problem is given in 2.

Definition 2 (Incremental Sequential Pattern Mining Problem, ISPM) Given an original database D, an incremental database db and a minimum support threshold parameter min_sup, discover all the frequent sequences P from the updated database $D' = D \cup db$, where $S_P \geq min\_sup \times |D'|$.

In Table 3, a static database has been shown which will be used as an example for the remaining discussion of static mining and in Table 4, an incremental database having two iterations has been shown which will be used for the remaining incremental mining discussion. The following incremental database is an incremental version of the prior static one. In the incremental database, in each iteration, for a row, ϕ means, no sequence is given for this row's sid. The final updated database for each pass is, the concatenation of all the sequences up to now. If, we have $min\_sup = 30\%$, then the minimum support value for a pattern to become frequent after first and second iteration will be $2(6 \times 0.3 = 1.8 \approx 2)$ and $3(10 \times 0.3 = 3)$ respectively.

4 Our Proposals

In this section, the proposals of this literature will be presented. First, we will discuss the proposed data structures leading to the pattern generation concepts and the proposed pruning strategies. Finally, the two mining algorithms Tree-Miner and IncTree-Miner will be discussed.

4.1 Proposed Structures

This section will include discussion regarding the proposed structures to represent the information along with some visualizations and pseudocodes to draw a clear picture.

4.1.1 SP-Tree and IncSP-Tree: Node Attributes

SP-Tree is a tree-based structure to represent the static sequential database and IncSP-Tree is a modified version of SP-Tree to represent the incremental database. In the incremental mining problem, it is essential to track the incremental database efficiently and IncSP-Tree is specially designed to solve this challenge. It provides some structural advantages to implicitly track the incremental database or modified subtrees. Sequences of the database are represented by our tree branches through its nodes. First, we will talk about the common attributes of both trees. Then, we will introduce the additional attributes of IncSP-Tree.

1. Item(I): Each node will represent an item of some sequence. Due to the overlapping characteristics, the same node can represent items of one or more sequences. E.g., in $<(abc)(ac)>$, a is an item, b is an item, etc.
2. Event no(ev): Each sequence consists of ordered events. So, in each sequence each item has a specific event number to represent it. E.g., in sequence $<(abc)(ac)>$,

first a's event number is 1 and second a's event number is 2. To represent this, each node will have an attribute named event no(ev).

3. Child node(ch): Similar to normal tree structures each non leaf node of the tree will have child nodes for $\{I, ev\}$ combination.
4. Next link(nl): Next links for an item α from a node V denote the first occurrences for that item in different disjoint subtrees of V. Using these links faster tree traversals are performed leading to faster generation of the patterns.
5. Parent info(p_{inf}): Parent info for a node contains the information of the items which are in the same event as it found in its ancestor nodes. We suggest for the bit based representation of parent info by numbering the items in the database which leads to memory compactness along with operational efficiency. E.g., in sequence $< (ab)(abc) >$, c's (node which represents second event's c) parent info will contain $\{a, b\}$ represented as 11 by numbering $a = 1$ and $b = 2$ and setting 1 in 1^{th} and 2^{nd} position.
6. Maximum achievable local support(S^{MAL}): This attribute of a node V represents the maximum achievable local support for an item α from the underlying subtree. It helps detect the infrequent patterns early and reduce pattern searching cost.

   Up to now, we have talked about the common attributes. Now, we will talk about the attributes which were added in the IncSP-Tree to efficiently control the manipulation of the incremental database (IncDB).
7. Present count(C) and Previous count(C'): Count related attributes represent the number of sequences that have overlapped this node. Present count and Previous count denote the total number of overlapping of this node up to the current pass and previous pass(last mined pass) respectively. These attributes are used to track the change in support from the last mined iteration for a pattern. For SP-Tree, as it is a static version, we only need a single count attribute. So, Present count is enough.
8. Created at(cr) and Modified at(md): These attributes are used to store the information that in which pass the node was first created and last modified respectively.
9. Modified next link(nl_m): This attribute has similar characteristics like next link except that it only tracks the modified nodes found as the first occurrences in its disjoint subtrees. This attribute is used to faster traverse in the modified nodes or subtrees. Modified nodes are those nodes that are affected due to the addition of the IncDB.

4.1.2 Sequence Summarizer(seq_sum)

Co-occurrence information [12,13] is helpful to guide during pattern extensions. To store this information, we maintain a table named Co-Existing Item Table (CETable) in our solution. This table has two columns, $CETable_s$ and $CETable_i$ which holds the information for an item α that which items β can perform sequence $((\alpha) \rightarrow (\alpha)(\beta))$ and itemset $((\alpha) \rightarrow (\alpha\beta))$ extension over it respectively. We have shown the CETable for the complete database of Table 3 in Fig. 4 (b). It gives idea regarding two items' combinations'$((\alpha)(\beta)$ or $(\alpha\beta))$ support value over the complete database. It is an important challenge how to efficiently calculate this information, especially during database increment. Another important challenge is during *Append* operation it is important to know in which node of the IncSP-Tree the sequence ended in the previous iteration so that we can directly reach that node and extend further which will also reduce branch searching cost for long sequences.

To solve these issues, we propose a structure to summarize information for each sequence named as *Sequence Summarizer*(seq_sum). A theoretical definition of this structure for each sequence is given in Table 5. Using this structure, we update global $CETable_s$ and $CETable_i$ and perform *Append* operation. Through last node reference(*) we denote the node where sequence s ended in IncSP-Tree. s_α^{st} denotes the first event where item α was found in s, $s_\alpha^{en\_previous}$, $s_\alpha^{en\_current}$ denote the last event where α was found in previous and current passes respectively. Through *same event info*, we keep the information for α that which items $\beta(\beta <_{order} \alpha)$ were found with α in the same event and keep as bitset representation for faster calculation.

For sequence with sid 1 from Table 3, after pass 2 the *positional information* and *same event info* of c will be $(4, 4, 6)$ and $(0, 11)$[1]. After updating CETable, we update summarizer's information for the following iterations. In Fig. 1, we describe the idea of updating $CETable_s$ and $CETable_i$ from a sequence summarizer structure for an updated sequence s based on the newly appended or inserted part. After *Append* or $Insert$ operation, this function is executed over the concerned sequence. In the algorithms, capital and bold *AND, OR, XOR* represent bitwise and, or and xor operations respectively. This convention is maintained throughout the discussion. The main advantage of this structure is, during database increment it only considers the additional part and eventually updates the complete co-occurrence information. In Fig. 4(c), the discussed example for $sid = 1$ is visualized by showing the status

[1] order wise $a = 1, b = 2$, so we set bit in the 1^{st} and 2^{nd} position getting $11_2 = 3_{10}$

sid	last node	positional info	same event info
s_{sid}	*(last) node reference	for each item $\alpha \in s$ $\alpha : (s_\alpha^{st}, s_\alpha^{en\_previous}, s_\alpha^{en\_current})$	for each item $\alpha \in s$ $\alpha : (bitset_\alpha^{previous}, bitset_\alpha^{current})$

Table 5: Definition of Sequence Summarizer

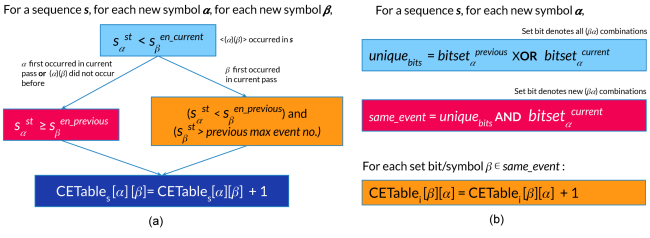


Fig. 1: (a) $CETable_s$ Calculation, (b) $CETable_i$ Calculation

after both iterations. Always only the final update is kept in the table. Also, in Fig. 4(c) it has been shown, how new combinations of items can generate due to the update in the summarizer table based on the algorithm discussed in 1.

4.1.3 Pseudocode for Tree Construction

As up to now we have talked about the tree attributes and Sequence Summarizer, now we will provide the algorithm to construct these structures from the sequences. The construction method of SP-Tree and IncSP-Tree are similar except IncSP-Tree has some additional attributes. In Algorithm 1, we provide the pseudo-code to insert a sequence into the tree along with the update of corresponding (track with *sid*) sequence summarizer (*seq_sum*) based on the input sequence (s) and in Algorithm 2, we provide the pseudo-code to update the attributes of the concerned branch of the tree related to the sequence.

For compactness and operational efficiency issues, we have used bitset representation for saving item information, but other representations will also work. The newly used variables are initially provided in the first two lines of the Algorithm 1. In *Append* operation, a prefix of the corresponding sequence (s_{sid}) already exists and ev_p is used to denote that previous maximum event no. In *Insert* operation, it is considered as -1. In Line 11, the actual item (I) and event no (ev') is calculated. When a desired child node is not found it is calculated and set (line 12-16). *seq_sum* is updated in line 18-27. Bit based operations are performed to update Same Event Info of *seq_sum* in line 26-27. Finally recursion is called to insert the remaining items in line

Algorithm 1 Insert sequence into IncSP-Tree

1: **Variables:** pass pa, node N, sequence s, previous maximum event ev_p, running bitset bit to update for each N, Item no I_{no}, Event No ev, Actual event no ev'.
2: **Globals:** pa, s, ev', seq_sum
3: **Additional functions:** $len(s)$ returns the number of event in s and $len(s[ev])$ denotes the number of items in event ev of s.
4: **Output:** s inserted, last node returned, seq_sum updated.
5: **procedure** InsIncSPTree(N, ev, I_{no}, bit)
6: **if** $ev = \text{len}(s)$ **then** ▷ All events been inserted
7: Return N
8: **if** $(I_{no} > \text{len}(s[ev]))$ **then** ▷ Add next event
9: ▷ 0 Based Indexing
10: Return InsIncSPTree(N,$ev+1$,0,0)
11: $I \leftarrow s[ev][I_{no}]$, $ev' = ev + ev_p$
12: **if** $(N.ch[\{I, ev'\}] = \{\})$ **then** ▷ No child found
13: ▷ create a child node n and initialize the attributes
14: $n.I \leftarrow I, n.ev \leftarrow ev', n.cr \leftarrow pa, n.C \leftarrow 0, n.C' \leftarrow 0, n.p_{inf} \leftarrow bit$
15: $n.md \leftarrow 0$ ▷ md will be set as pa in Algorithm 2
16: $N.ch[\{I, ev'\}] \leftarrow n$ ▷ Setting n as child node
17: $n \leftarrow N.ch[\{I, ev'\}]$
18: **if** $seq\_sum[s_{sid}][I] = \{\}$ **then**
19: ▷ Positional Info update
20: $seq\_sum[s_{sid}][I]_{st} \leftarrow ev'$,
21: $seq\_sum[s_{sid}][I]_{en\_previous} \leftarrow ev'$
22: $seq\_sum[s_{sid}][I]_{en\_current} \leftarrow ev'$
23: **else** ▷ Ending position update
24: $seq\_sum[s_{sid}][I]_{en\_current} \leftarrow ev'$
25: ▷ set bit/items are in same event
26: $bit' \leftarrow seq\_sum[s_{sid}][I]_{bitset_{current}}$ **OR** bit
27: $seq\_sum[s_{sid}][I]_{bitset_{current}} \leftarrow bit'$ ▷ Same Event Info update)
28: Return InsIncSPTree(n,ev',$I_{no}+1$,bit' **OR** bit)
29: ▷ bit' **OR** bit: Setting bit for I for next call

28 with child node n. Recursion stopping conditions are given in line 6.

Using Algorithm 2, we update the attributes of the tree. As SP-Tree has a subset of attributes from IncSP-Tree, it can follow the same procedure. But whereas IncSP-Tree needs to run it after each sequence insertion, SP-Tree can run only once after complete insertion of the database. Using bottom up recursive traversal with the usage of last node reference of *seq_sum* for each sequence s the nodes' attributes are calculated (line 35). For each N, its nl, nl_m and S^{MAL} attributes are updated using the information sent from underlying nodes in lines (25-26), (10-12) and (13-14) respectively. Also, for N the corresponding global lists L_{nl}, L_{nl_m} and L_{SMAL} are updated to send its information to its ancestor nodes in lines (27-30), (15-17), (21-24) respectively. In *Insert* operation, all the nodes' concerning s count attributes are incremented (lines 18-19) whereas in *Append* operation only the newly considered nodes' for s count attributes get incremented (lines 21-22). In

Algorithm 2 Updating Attributes of IncSP-Tree

Globals: Pass pa, list containing nodes for next links and modified next links as L_{nl} and L_{nl_m} respectively, Operation type as t, list containing underlying S^{MAL} information L_{SMAL}.
Output: Update of the concerned branch's attributes.
procedure UPDATE PATH(N)
  if (N is **None**) **then**
    Return ▷ All the node's attributes are updated
  $md' \leftarrow N.md$
  if $N.md < pa$ **then** $N.md \leftarrow pa$ ▷ newly modified node
    ▷ Support Tracking, Runtime memory clear
    $N.C' \leftarrow N.C$, $N.nl_m \leftarrow \{\}$
  for item $i \in L_{nl_m}$ **do**
    ▷ Tracking underlying modified nodes
    $N.nl_m[i] \leftarrow N.nl_m[i] \cup L_{nl_m}[i]$
  for item $i \in L_{SMAL}$ **do**
    $N.S^{MAL}[i] \leftarrow N.S^{MAL}[i] + 1$ ▷ to reduce loop
  if $md' \neq N.md$ **then** ▷ N was not tracked
    $L_{nl_m}[N.I] \leftarrow N$ ▷ Need to track N from ancestors for I
  else Delete $L_{nl_m}[N.I]$ ▷ Already Been Tracked
  if $t = Insert$ **then**
    $N.c \leftarrow N.c + 1$,$L_{SMAL} \leftarrow L_{SMAL} \cup \{N.I\}$
  else ▷ Append Operation
    if N is newly considered node for s **then**
      $N.c \leftarrow N.c + 1$, $L_{SMAL} \leftarrow L_{SMAL} \cup \{N.I\}$
    else ▷ Old nodes, Already Tracked
      $L_{SMAL} \leftarrow L_{SMAL} - \{N.I\}$
  for item $i \in L_{nl}$ **do** ▷ Underlying nl update
    $N.nl[i] \leftarrow N.nl[i] \cup L_{nl}[i]$
  if $N.cr = pa$ **then** ▷ Tracking new nodes for next link
    if $md' = N.md$ **then** ▷ Already Tracked for next link
      Delete $L_{nl}[n.I]$
    else $L_{nl}[N.I] \leftarrow N$ ▷ Tracking node for nl
  else Delete$L_{nl}[N.I]$ ▷ Already Tracked by ancestors
  UpdatePath($N.parent$)

Algorithm 2, it has been assumed that the mine operation is performed after each batch of data increment. But the proposed framework has no such dependency and can be slightly tweaked to mine anytime based on users' requests.

4.1.4 SP-Tree and IncSP-Tree: Visualization

In Fig. 2 (a), the complete SP-Tree has been shown after inserting all the sequences of the static database into the tree. For simplicity and discussion purpose, it has been assumed that the sequences are inserted according to the increasing order of the sids. Nodes are numbered (blue color) to depict their creation order. In Fig. 2, the complete commentary has been stated to understand how each sequence is inserted, new nodes are created or existing nodes get prefix shared, etc. E.g., due to the insertion of $sid = 2$ nodes 9-14 are created and for $sid = 3$, nodes 9-10 get prefix shared and nodes 15-17 get newly created. To represent p_{inf} for each node (shown in red color), bit based representation is used and the corresponding numbering scheme has also been shown. For each node, only a single count attribute is used as per the discussion stated in 4.1.1.

In Fig. 3, the proposed IncSP-Tree has been shown for the exampled incremental database for the two iterations. Similar to static mining, sequences are inserted according to the increasing order of *sid*s and nodes' numbers (blue color) depict their creation order. As per the attributes' discussion, for each node two count related attributes have been shown. Inside each node, we have shown four attributes, I, ev, C' and C respectively. Before updating an existing branch, first we update the previous count with the present to track the support change between two iterations. Then we perform update where in $Insert$ operation every node's count increases but in *Append* operation only the nodes which were newly considered due to the new items' arrival in the sequence get incremented. For example nodes (20-22) get incremented but nodes (1-5) do not get incremented after the 2nd pass. This is an *Append* operation over $sid = 1$.

In Fig. 7 (b), we have shown the next links(nl) and modified next links(nl_m) for node 6 and 7 where $nl_m \subseteq nl$. This separation is useful to faster traverse in the modified subtrees of IncSP-Tree. When a subtree is modified the corresponding path up to the root is also considered modified so that the ancestor nodes can detect the update in the successor nodes through modified links and count attributes' variation. There are basically two types of modified nodes, those nodes where are newly created (filled in blue color) and those old nodes which (filled in red color) have modified nodes underneath. Uncolored nodes are the unmodified nodes. Patterns consist of the modified nodes are actually the affected patterns due to database increment. In Fig. 4 (c) we have shown the sequence summarizer for $sid = 1$ after both iterations.

4.2 Pattern Generation from SP-Tree Structures

Our pattern mining approach follows the pattern growth technique with suffix extension. In the literature mainly two types of pattern extensions are conducted, *Sequence Extension (SE)* means adding a new item as an event in the current pattern and *Itemset Extension (IE)* means adding a new item in the last event of the pattern where the added item is lexicographically (or based on some ordering scheme) larger compared to the existing items of the concerned event.

Definition 3 (Pattern Formation) Based on the Fig. 4 (f), pattern formation concepts can be pointed as follows -

- Patterns are generated through SP-Tree node concatenations, i.e., $N'_P = \{ (V, V_{1,1}, V_{1,1,1}), (V, V_{1,2}, V_{1,2,2}), (V, V_{2,1}, V_{2,1,1}), (V, V_{3,1}, V_{3,1,1})\}$.
- Each node concatenation $n \in N'_P$ -
  1. Represents a pattern occurrence in a subtree.

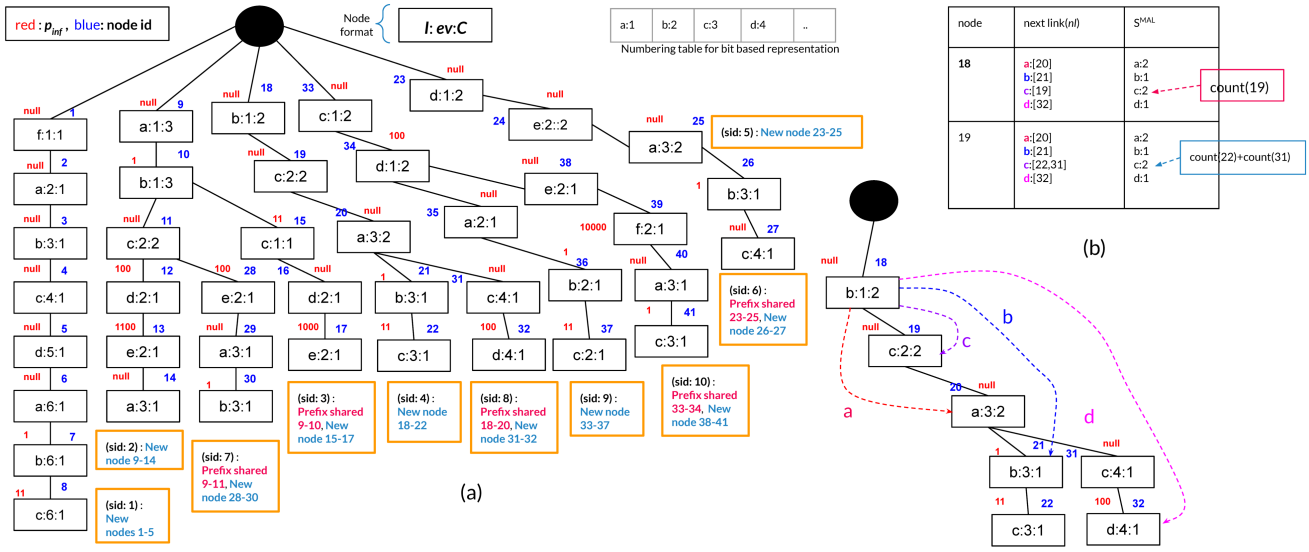


Fig. 2: (a) Complete SP-Tree, (b) Next links for node 18 and 19

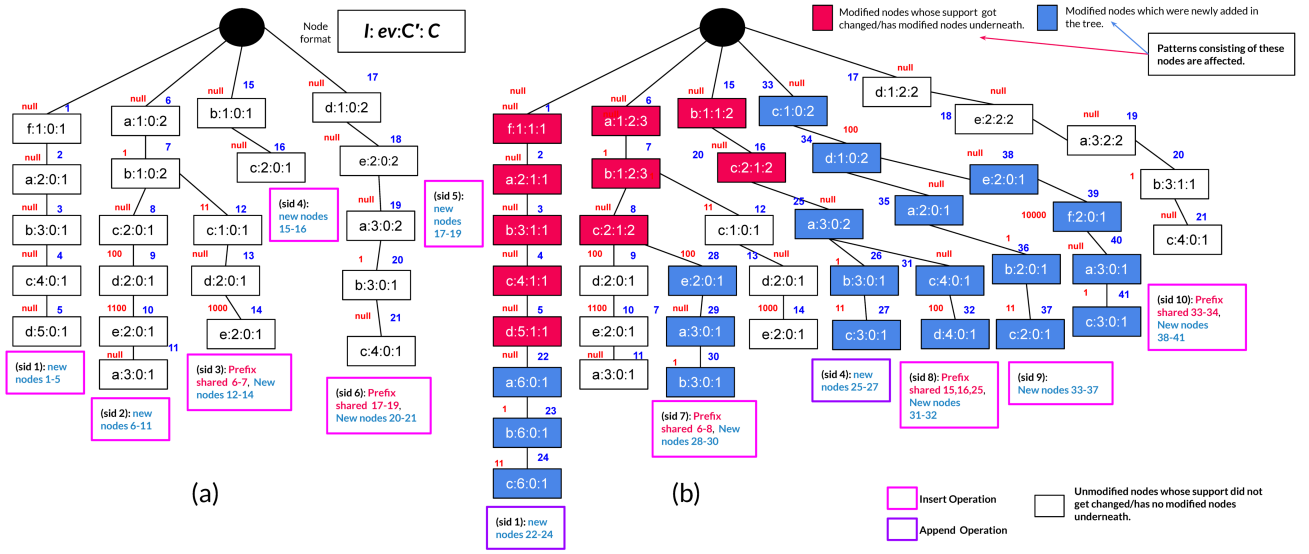


Fig. 3: (a) IncSP-Tree after iteration 1, (b)IncSP-Tree after iteration 2

2. Ends in different disjoint subtrees compared to other $n_j \in N'_P$.
3. Always the first occurrences in different subtrees are considered.

So, a pattern P can be represented by the nodes where it ends, N_P={ $V_{1,1,1}$, $V_{1,2,2}$, $V_{2,1,1}$, $V_{3,1,1}$}.

– Support of P is $S_P = \sum_{n \in N_P} C(n)$. C denotes the count attribute of each node.
– For each $n \in N_P$, we can say that, we have completed searching up to that node. During following iterations, we will search only in their underlying subtrees.

In Fig. 5, we have shown an example of pattern formation which states how the nodes' concatenation can form a pattern and for each occurrence the last node position can represent the pattern in a subtree.

Suppose, we have a pattern $P =< (\alpha\beta) >$ ending at nodes $N_P = \{V_i, V_j, V_k, .., V_n\}$. We want to find nodes $N_{P\gamma}$ which will extend P for γ. For each $V \in N_P$, we follow similar extension procedure (SE/IE) each having two cases. Based on Fig. 4 (d), we can discuss the extensions as follows -

Definition 4 (SE in SP-Tree, $P \rightarrow P\{\gamma\}$**)** Two cases are -

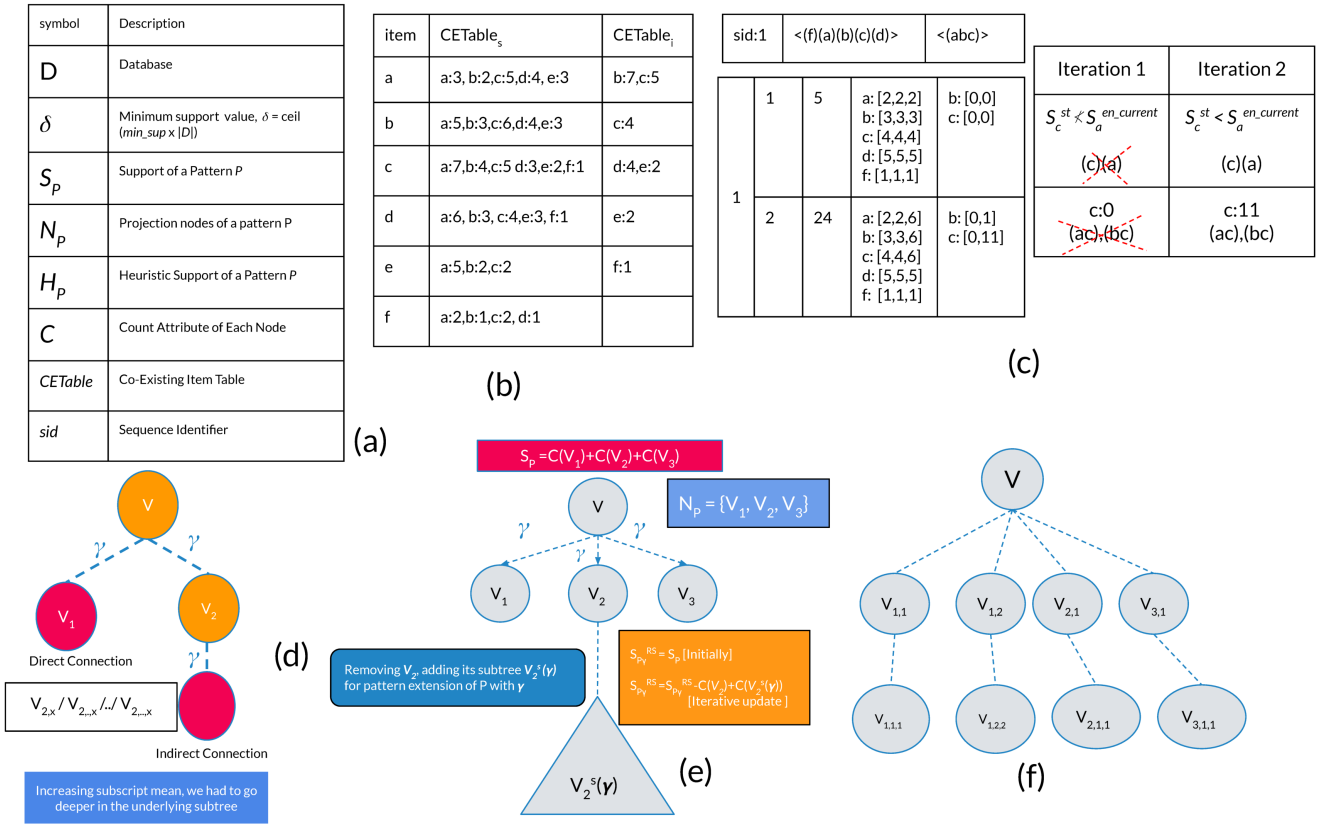


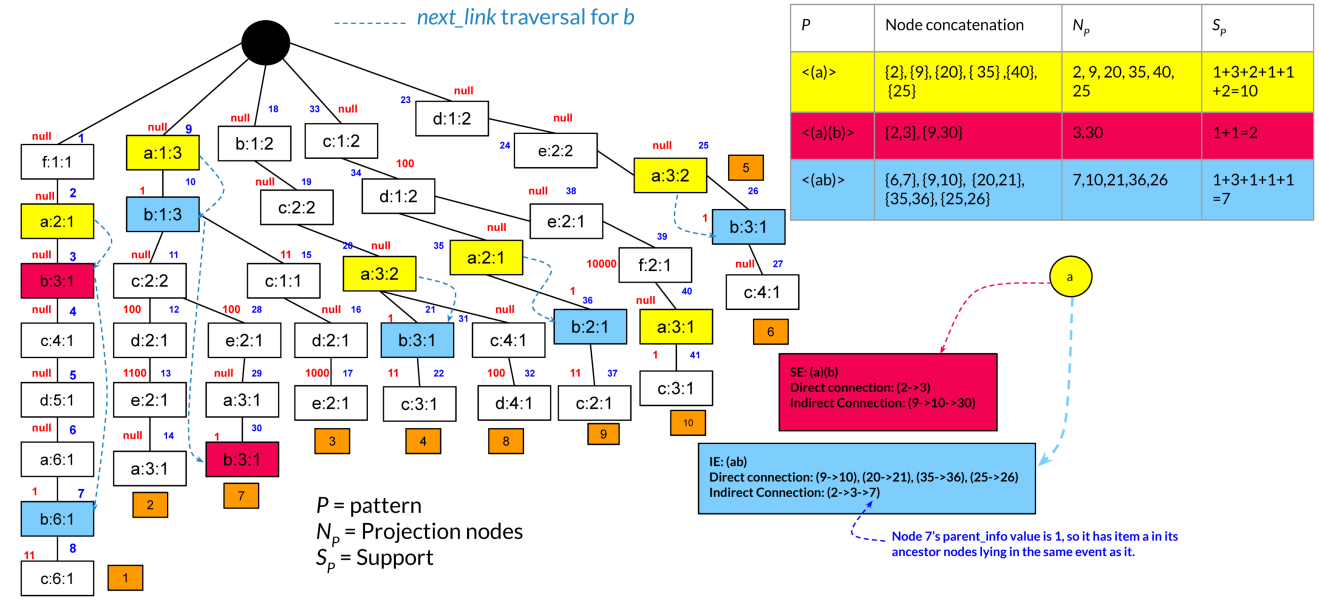
Fig. 4: (a) Terminologies for static mining (b) Co-Existing Item Table, (c) Example of sequence summarizer structure for $sid = 1$, (d) Pattern Generation Example, (e) Breadth-First based support counting Technique, (f) Pattern Formation Concepts

1. Direct Connection: For a node $V \in N_P$ suppose through next link for γ, we reach node $V_1 (V.nl[\gamma] = \{V_1\})$. If $V.ev \neq V_1.ev$, then V_1 can perform SE over V. E.g., $(a) \rightarrow (a)(b)$ with nodes $\{(2) \rightarrow (3)\}$ (Fig. 5).
2. Indirect Connection: For a node $V \in N_P$ suppose through next link for γ, we reach node $V_2 (V.nl[\gamma] = \{V_2\})$. If $V.ev = V_2.ev$ then V_2 can not perform SE over V. Then, we need to search in underlying subtree of V_2 to find such node(s) $V_{2,\ldots,x}$ where $V.ev \neq V_{2,\ldots,x}.ev$. In this case, we need to perform two level next link traversals for γ from V. E.g., $(a) \rightarrow (a)(b)$ with nodes $\{(6) \rightarrow (7) \rightarrow (30)\}$ (Fig. 5).

Definition 5 (IE in SP-Tree, $P \rightarrow \{P\gamma\}$) Two cases are -

1. Direct Connection: For a node $V \in N_P$ suppose through next link for γ, we reach node V_1. If $V.ev = V_1.ev$, then V_1 can perform IE over V. E.g., $(a) \rightarrow (ab)$ with nodes $\{(6) \rightarrow (7)\}$ (Fig. 5).
2. Indirect Connection: For a node $V \in N_P$ suppose through next link for γ, we reach node V_2. If $V.ev \neq V_2.ev$ then V_2 can not perform IE over V. Then, we need to search in underlying subtree of V_2 to find such node(s) $V_{2,\ldots,x}$ which have items α,β in the same event as it found in its predecessor nodes. From a node's p_{inf} we get the items which are in the same event as it and can be calculated through bit-wise AND operation. E.g., $(a) \rightarrow (ab)$ with nodes $\{(2) \rightarrow (3) \rightarrow (21)\}$ (Fig. 5).

To search in a node's underlying subtree we take all the nodes reached through next link and simulate each connection cases. In Fig. 2 (b), we have shown a small visualization of traversing through next links. Our node extension procedure works in level by level manner through our proposed breadth-first based support counting mechanism.



P	Node concatenation	N_p	S_p
<(a)>	{2}, {9}, {20}, { 35} ,{40}, {25}	2, 9, 20, 35, 40, 25	1+3+2+1+1 +2=10
<(a)(b)>	{2,3}, {9,30}	3,30	1+1=2
<(ab)>	{6,7}, {9,10}, {20,21}, {35,36}, {25,26}	7,10,21,36,26	1+3+1+1+1 =7

Fig. 5: Pattern Extension Examples

4.3 Pruning Techniques

In this section, we will discuss the applied pruning techniques that help to detect the redundant patterns early and reduce the search space. Suppose, we have a pattern $P =< (\alpha\beta) >$ with support S_P ending at nodes $N_P = \{n_1, n_2, n_3, ..., n_k\}$ and corresponding $sList = \{\alpha, \beta, \gamma, \delta\}$ and $iList = \{\gamma, \delta\}$. $sList$ and $iList$ give idea regarding which items may extend P as SE and IE respectively. Let $\delta = \lceil min\_sup \times |D| \rceil$. First, we will discuss the support downward closure. The terminologies used in static mining discussion has been shown in Fig. 4(a).

Lemma 41 (Support Downward Closure) *From a node $n_i \in N_P$, suppose we reach a set of nodes $M = \{m_{1,i}, m_{2,i}, ..., m_{x,i}\}$ through next link for any item ϵ, where each $m \in M$ are in disjoint subtrees and $m_{j,i}$ denotes the j^{th} branch in n_i's subtree. Then $C(n_i) \geq \sum_{j=1}^{x} C(m_{j,i})$.*

Proof This holds because of the tree's node overlapping characteristics.

Now, we will discuss the pruning strategies based on their execution order.

4.3.1 CETable Based Pruning

First, we will provide the definition of this strategy stated in a lemma. Following that, we will give some examples to understand it.

Lemma 42 (CETable Based Pruning) *An item γ can extend P as sequence extension iff $CETable_s[\alpha][\gamma] \geq \delta$ and $CETable_s[\beta][\gamma] \geq \delta$. Similarly $\gamma(\gamma > \beta > \alpha)$ can extend P as itemset extension $CETable_i[\alpha][\gamma] \geq \delta$ and $CETable_i[\beta][\gamma] \geq \delta$. For each item I belongs to last event of P this constraint is checked.*

Proof CETable holds the co-occurrence information of the items and for any super pattern to satisfy min_sup constraint, its sub patterns also need to satisfy. This was adopted in our solution from [12].

For example, suppose, $\delta = 3$, now pattern $< (a)(c)(b) >$ can not be frequent, because $< (a)(b) >$ is not frequent (in $CETable_s$ column of Fig. 4(b), $S_{(a)(b)} = 2 < \delta$). Similar pruning strategy can also be applied based on $CETable_i$ column.

4.3.2 Breadth-First Based Support Counting Technique

Now, we will talk about our proposed new *Breadth-First Based Support Counting Technique* which is stated in Algorithm 3. We have used comment to understand the underlying logic behind the statements. This technique is valid because of the Lemma 41. When we remove a node and go deeper in the subtree the maximum possible support for the extended pattern will stay same or reduce. Theoretically the strategy is explained in Lemma 43 using Fig. 4 (e).

Lemma 43 (Breadth-First Based Counting Strategy) *Suppose, from a pattern P we have to check extension*

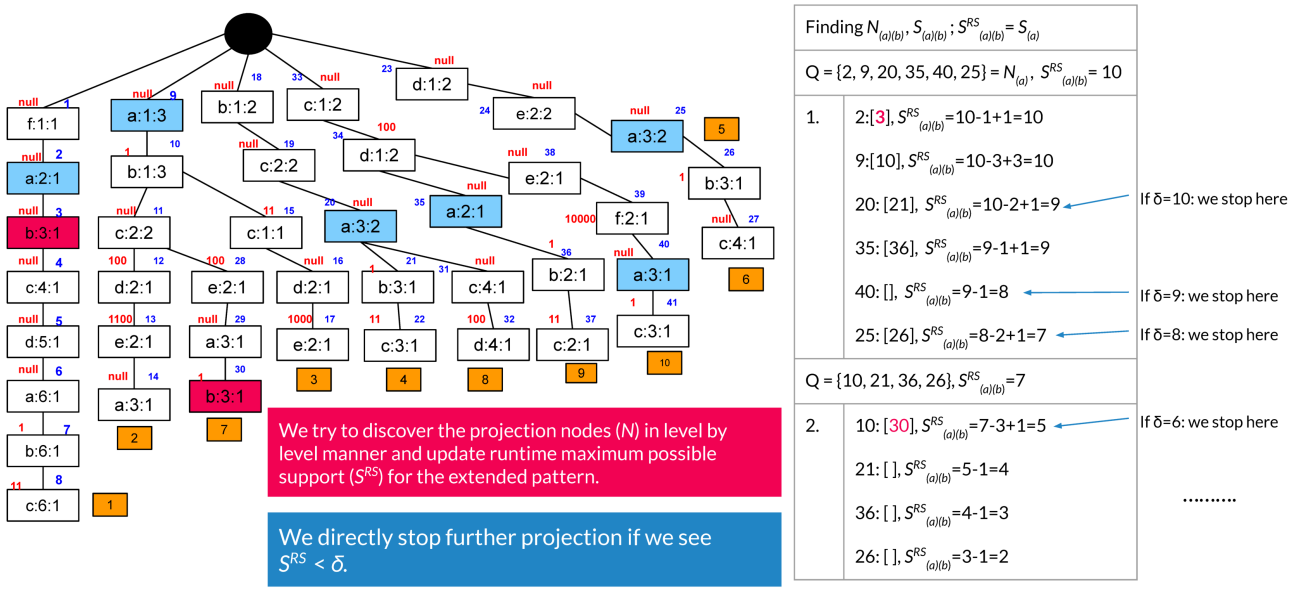


Fig. 6: Simulation of Breadth-First Based Support Counting Technique

Algorithm 3 Breadth-First Based Support Counting

Globals: $\delta = min\_sup \times |D|$
procedure BREADTHFIRSTSUPPORTCOUNTING(P, N_P, γ, S_P)
▷ Extended Nodes, Actual Support,Queue
$N_{P\gamma} \leftarrow \{\}, S^{RS}_{P\gamma} \leftarrow S_P, Q \leftarrow N_P$
for each node $n_i \in Q$ **do**
▷ Extending P for $\gamma(P\gamma) : (\alpha\beta)(\gamma)$ or $(\alpha\beta\gamma)$
if $(S^{RS}_{P\gamma} - C(n_i) + n_i.S^{MAL}[\gamma] < \delta)$ **then**
Return Infrequent
▷ Removing Node, Will check in subtree
$Q \leftarrow Q - \{n_i\}$, $S^{RS}_{P\gamma} \leftarrow S^{RS}_{P\gamma} - C(n_i)$
for each node $m_j \in n_i.nl[\gamma]$ **do**
▷ Maximum possible support can be found in $S^{RS}_{P\gamma}$
$S^{RS}_{P\gamma} \leftarrow S^{RS}_{P\gamma} + C(m_j)$
if (pattern extension constraint satisfied) **then**
$N_{P\gamma} \leftarrow N_{P\gamma} \cup m_j$
else ▷ Need to check further in underlying subtrees
$Q \leftarrow Q \cup \{m_j\}$
if $(S^{RS}_{P\gamma} < \delta)$ **then** Return Infrequent
Return $N_{P\gamma}, S^{RS}_{P\gamma}$ ▷ Frequent: Return extension nodes with actual support

for $P\gamma$ from a set of projection nodes, $N_P = \{V_1, V_2, V_3\}$ where initially current maximum possible support for the extended pattern is $S^{RS}_{P\gamma} = S_P$. For each $n \in N_P$, it is removed, its underlying subtree($n^s(\gamma)$) is added for iterative searching and the runtime maximum possible support gets updated to $S^{RS}_{P\gamma} = S^{RS}_P - C(n) + \sum C(n^s(\gamma))$. The extension nodes for $N_{P\gamma}$ is discovered in level by level manner.

Proof As, $C(V) \geq \sum C(V^s_2(\gamma))$ [Lemma 41], so $S^{RS}_{P\gamma} \leq S_P$.

The main advantage of this technique is, we may detect the infrequency of a pattern early without performing its complete projection. A small simulation to understand this technique has been shown in Fig 6. In Fig. 6 we have shown how we will calculate the projection nodes for the pattern $< (a)(b) >$. We discover the nodes in level by level manner and may stop projecting at any time based on the maximum possible support.

4.3.3 Heuristic iList Pruning

This is our second proposed pruning strategy and stated in Lemma 44.

Lemma 44 (Heuristic *iList* Pruning) *Suppose, We have an item $\epsilon (\epsilon \in sList \cap iList)$. During support calculation as SE for ϵ, we collectively reach nodes $M = \{m_{1,1}, m_{2,1}, .., m_{1,2}, m_{2,2}, .., m_{1,x}, .., m_{2,y}, .., m_{k,z}\}$ from $n.nl[\gamma]; \forall n \in N_P$. Let $H_{P\{\epsilon\}} = \sum_{m \in M} C(m)$. If heuristic support, $H_{P\{\epsilon\}} < \delta$, then ϵ can not pefrorm IE over P and can be removed from iList.*

Proof Intuition lies behind Lemma 41. Actual support for IE, $S_{\{P\epsilon\}} \leq H_{P\{\epsilon\}} < \delta$. Because we need to go deeper to find the valid nodes for the extension.

To understand this strategy, an example can be seen using SP-Tree of Fig. 2. Here $N_{(a)} = \{2, 9, 20, 35, 40, 25\}$. For b, $< (a) >$ will be extended. Now, the first level nodes reached through next links for b from each $n \in N_{(a)}$ are $\{3, 10, 21, 36, 26\}$ and total heuristic support $H_{(a)(b)}$ is 7 $(C(3) + C(10) + C(21) + C(36) + C(26))$. Now, if $H_{(a)(b)} < \delta$, then without any further checking b can be removed from both *sList* and *iList*.

4.3.4 Recursive sList and iList pruning

This strategy is stated in Lemma 45. The lemma 45 is a very widely addressed pruning strategy which has been adopted in our solution and it comes from the apriori property that if $< (\alpha)(\gamma) >$ is not frequent, then $< (\alpha)(\beta)(\gamma) >$ will never be frequent.

Lemma 45 (Recursive *sList* and *iList* pruning) *Suppose, after support calculation the found shrinked lists are* $sList' = \{\alpha, \beta, \gamma\}$ *and* $iList' = \{\delta\}$ *which will extend* P*. Now during recursive extensions for each* $\theta \in sList'$ *as SE the corresponding sList and iList will be* $sList'$ *and* $\{\epsilon | \epsilon \in sList' \cap \epsilon >_{order} \theta\}$ *respectively. Similarly to perform IE for each* $\theta \in iList'$ *the corresponding sList and iList will be* $sList'$ *and* $\{\epsilon | \epsilon \in iList' \cap \epsilon >_{order} \theta\}$ *respectively.*

4.3.5 Loop Reduction

This implementational strategy is also designed using the node properties and theoretically stated in Lemma 46 and applied in Algorithm 3. Before searching for an item in the underlying subtree, first the corresponding value in S_{MAL} is checked and the possible maximum support is heuristically calculated. If this heuristic value fails to support δ, then there's no need to perform more projection.

Lemma 46 (Loop Reduction) *Suppose, we want to extend from node* $n \in N_P$ *for an item* ϵ *and current maximum possible support for the extended pattern is* $S^{RS}_{P\epsilon}$*. If,* $(S^{RS}_{P\epsilon} - C(n) + n.S^{MAL}[\epsilon]) < \delta$*, then* $P\epsilon$ *can be detected as infrequent.*

Proof For any P's projection node N_P for any item ϵ, $N_P.S^{MAL}[\epsilon] = \sum_{n \in N_P.nl[\epsilon]} C(n) \geq \sum_{n' \in N_{P\epsilon}} C(n')$.

4.4 BPFSP-Tree: Bi-Directional Projection Pointer Based Frequent Sequential Pattern Tree

In ISPM problem, after each iteration, we need to report the current list of frequent patterns with their support. So, to store the frequent patterns traditional literature maintain a tree-like structure [5,23]. In our study, we propose a new tree-based structure named *Bi-Directional Projection Pointer Based Frequent Sequential Pattern Tree (BPFSP-Tree)* to store the patterns with their support and projections using *IncSP-Tree* node pointers. An important point to note that generally in most of the cases (or after some iterations) the size of the incremental database gets quite smaller compared to the size of the existing database. Thus why, generally after some iterations, a pattern's transition from frequent to infrequent or vise-versa occurs slowly. Moreover, it is the normal characteristic that either an existing frequent pattern's super pattern will get infrequent to frequent or a current frequent pattern along with its some of the sub-patterns will get frequent to infrequent also with the addition of some completely new patterns with new prefix as frequent. So, it is important to give focus on the frequent patterns' representation because they might control the DB scan. The main features and characteristics of our proposed *BPFSP-Tree*(B) is as follows -

1. *Pattern, Support and Projection Pointer:* BPFSP-Tree stores the frequent patterns(P) with their corresponding support(S_P) and projection nodes(N_P) through prefix sharing. So, for each node B_P a frequent pattern(P) can be formed concatenating the items of the nodes from root(B_{root}) to it(B_P) in order. B_P's support and projection pointers will denote P's support(S_P) and projection nodes(N_P). Projection information reduces the number of DB scans by directly reaching this node and getting the projection info for further extensions ($N_P \rightarrow N_{P\gamma}$).
2. *Non-Frequent Item Buffer(NIB):* Suppose during pattern extension for a frequent pattern P, $P \rightarrow P\gamma$'s projection is completely calculated. Then to mitigate this cost, only $S_{P\gamma}$ is stored in an item buffer in B_P to use it in the successive mining iterations by implicitly merging modified and unmodified nodes' support. This buffer is called *Non-Frequent Item Buffer(NIB).*
3. *Bottom up pruning using end-link*: After a mining iteration some previously frequent patterns can become infrequent. To remove such patterns from BPFSP-Tree, bottom up strategy is followed which leads to traverse lesser nodes. To traverse from one leaf node to another in BPFSP-Tree, a link is maintained named as *end_link*.
   A small example of BPFSP-Tree is shown in Fig. 7 (c) with *end_link*s for traversing through the leaf nodes using SP-Tree node references of Fig. 3 (b).

4.5 Mining Algorithms

In this section, based on the above discussions, we will describe our proposed mining algorithms. Tree-Miner based on SP-Tree and IncTree-Miner based on IncSP-Tree are two algorithms to mine the complete set of frequent patterns for static and incremental database respectively. The main challenge of an incremental mining algorithm is how to efficiently track those patterns which are affected due to the addition of IncDB rather

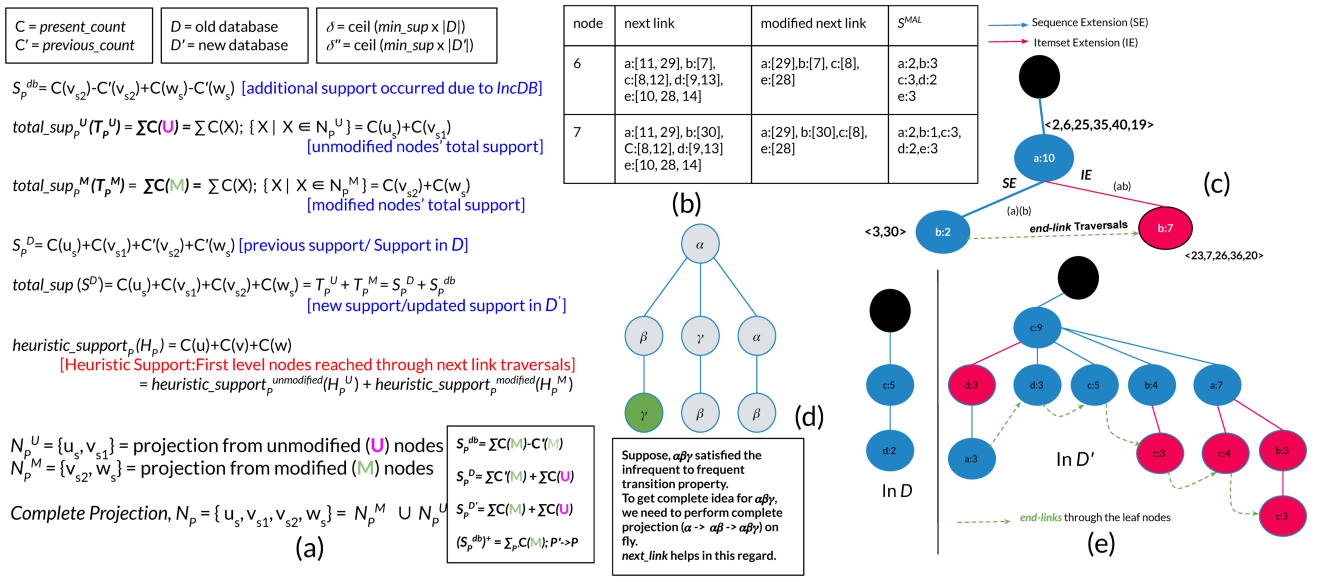


Fig. 7: (a) Pattern Generation Approach for IncTree-Miner, (b) Example of next links and modified next links, (c) Example of BPFSP-Tree, (d) Memory Resilient IncTree-Miner, (e) BPFSP-Tree for patterns with prefix $< c >$ after iteration 1 and 2 for $\delta = 2, \delta' = 3$.

Algorithm 4 Tree-Miner

```
1: Globals: δ = ⌈min_sup × |D|⌉
2: Output: Complete set(F) of frequent sequential patterns.
3: procedure TreeMiner(P, N_P, sList, iList)
4:                                          ▷ CETable based pruning
5:     Reduce sList and iList based on CETable.
6:                                          ▷ Nodes which will perform extension
7:     N_SE ← {}, N_IE ← {}
8:     for each item γ ∈ sList do                          ▷ SE checking
9:         ∀n ∈ N_P perform SE for γ(P{γ})
10:        if (S_P{γ} < δ) then
11:            sList ← sList − {γ}                         ▷ Infrequent Pattern
12:            if (H_P{γ} < δ and γ ∈ iList) then
13:                iList ← iList − {γ}                     ▷ 4.3.3
14:        else          ▷ Frequent Pattern, tracking extension nodes
15:            N_SE[γ] ← N_P{γ}
16:    for each item γ ∈ iList do                          ▷ IE checking
17:        ∀n ∈ N_P perform IE for γ({Pγ})
18:        if (S_{Pγ} < δ) then
19:            iList ← iList − {γ}                         ▷ Infrequent Pattern
20:        else          ▷ Frequent Pattern, tracking extension nodes
21:            N_IE[γ] ← N_{Pγ}
22:    for (each item γ ∈ sList) do                        ▷ Recursive SE
23:        TreeMiner(P{γ}, N_SE[γ], sList, {α|α ∈ sList ∩ α > γ})
24:    for (each item γ ∈ iList) do                        ▷ Recursive IE
25:        TreeMiner({Pγ}, N_IE[γ], sList, {α|α ∈ iList ∩ α > γ})
```

than complete mining. IncTree-Miner is efficiently designed to solve this challenge.

4.5.1 Tree-Miner: Static Mining Algorithm

Our mining approach is based on forward mining with suffix extension. We start with a single item pattern P with the information in which nodes N_P it ends along with two supporting lists, $sList$ and $iList$, which give idea regarding the items which might perform SE and IE over it. To extend P for an item γ, we find the desired nodes for each $n \in N_P$ in the underlying subtrees and recursively perform the extensions with the shrunken $sList$ and $iList$ through our proposed support counting technique and pruning strategies. We have provided the pseudo-code for Tree-Miner in Algorithm 4. Here, S_P and H_P denote a pattern P's actual and heuristic support respectively. In Fig. 8, we have shown how patterns are recursively discovered in Tree-Miner. To visualize, we have shown the generated patterns with prefix $< (a) >$ and considered $\delta = 2$. For each pattern we have shown its support, its initial $sList$,$iList$, shrinked $sList'$, $iList'$ and corresponding projection nodes. Node numbers can be mapped to the exampled SP-Tree.

4.5.2 IncTree-Miner: Incremental Mining Algorithm

IncTree-Miner algorithm finds the complete set of frequent patterns from the updated database. After each iteration, basically three types of events can occur - some infrequent patterns become frequent (patterns with completely new prefix or new suffix with the existing ones), some frequent patterns become infrequent and some frequent patterns' support get incremented. Main goal of the IncTree-Miner is to handle these operations efficiently. There are some important observations to highlight in IncTree-Miner,

Definition 6 (Incremental Support) Suppose, we have a pattern P ending at nodes $N_P = \{n_1, n_2, n_3, n_4, n_5\}$

Step	P:S	sList, iList	sList', iList'	N_p
1	<(a)>:10	sList = {a, b ,c, d, e} iList = { b, c, d, e }	sList' = {a, b ,c, d, e } iList' = { b, c}	{2, 9, 20, 35, 40, 25}
2	<(a)(a)>:3	sList = {a, b ,c, d, e} iList = { b, c, d, e}	sList' = { }, iList = {b}	{6, 14, 29}
3	<(a)(ab):2>	sList = { }, iList = { }	sList' = { }, iList = { }	{7, 30}
4	<(a)(b)>: 2	sList = {a, b ,c, d, e} iList = { c, d, e }	sList' = { }, iList' = { }	{3,30}
5	<(a)(c)>:4	sList = {a, b, c, d, e} , iList = {d,e}	sList' = {a, b }, iList' = {d,e}	{4, 11, 31, 27}
6	<(a)(c)(a)>:3	sList = {a, b}, iList = {b}	sList' = {}, iList' = { b}	{6, 14, 29}
7	<(a)(c)(ab)>:2	sList = { }, iList = { }	sList' = { }, iList' ={ }	{7, 30}
8	<(a)(c)(b)>:2	sList = {a, b} , iList = { }	sList' = {}, iList' = { }	{7, 30}
9	<(a)(cd)>:2	sList = {a,b} ,iList = {e}	sList' = {}, iList' = {}	{12, 32}
10	<(a)(ce)>:2	sList = {a,b} ,iList = {}	sList' = {a}, iList' = {}	{13, 28}
11	<(a)(ce)(a)>:2	sList = {a} ,iList = {}	sList' = {}, iList' = {}	{14,29}
12	<(a)(d)>:4	sList = {a, b ,c, d, e}, iList = {e}	sList' = {a}, iList' = {e}	{5, 12, 16, 32}
13	<(a)(d)(a)>:2	sList = {a}, iList = {}	sList' = { },iList' = {}	{6, 14}
14	<(a)(d)>:4	sList = {a, b ,c, d, e}, iList = {e}	sList' = {a}, iList' = {e}	{5, 12, 16, 32}
15	<(a)(d)(a)>:2	sList = {a}, iList = {}	sList' = { }, iList' = {}	{6, 14}
16	<(a)(de)>:2	sList = {a},iList = {}	sList' = {}, iList' = {}	{13, 17}
17	<(a)(e)>:3	sList = {a, b ,c, d, e} , iList = { }	sList' = {a }, iList '= {}	{13, 28, 17}
18	<(a)(e)(a)>:2	sList = {a} , iList = { }	sList' = {}, iList' = {}	{14, 29}
19	<(ab)>:7	sList = {a, b ,c, d, e} iList = { c }	sList' {a, c, d, e } iList' = { c}	{7, 10, 21, 36, 26}
20	<(ab)(a)>:2	sList = {a, c, d, e} , iList = { c,d,e }	sList' = {}, iList' = {}	{14, 29}
21	<(ab)(c)>:3	sList = {a, c, d, e} , iList = { d,e}	sList' = {a} ,iList' = {e}	{11, 27}
22	<(ab)(c)(a)>:2	sList = {a} ,iList = {}	sList' = {} ,iList' = {}	{14, 29}
23	<(ab)(ce)>:2	sList = {a} ,iList = {}	sList' = {a} ,iList' = {}	{13, 28}
24	<(ab)(ce)(a)>:2	sList = {a} ,iList = {}	sList' = {} ,iList' = {}	{14, 29}
25	<(ab)(d)>:2	sList = {a, c, d, e} , iList = {e}	sList' = { }, iList' = {e}	{12, 16}
26	<(ab)(de)>:2	sList={}, iList={}	sList'={}, iList'={}	{13, 17}
27	<(abc)>:4	sList = {a, c, d, e} , iList = {}	sList' = { }, iList' = { }	{8, 15, 22, 37}
28	<(ac)>:5	sList = {a, b, c, d, e}, iList = { }	sList' = { }, iList' = { }	{8, 15, 22, 37, 41}

Fig. 8: Simulation of Tree-Miner for the patterns with prefix $< (a) >$

P	N_P^M	T_P^M	N_P^U	T_P^U	S_P^{db}	$N_P^{D'}$	$S_P^{D'}$
$< (ab) >$	$\{23, 7, 26, 36\}$	6	$\{20\}$	1	4	$\{23, 7, 26, 36, 20\}$	7
$< (ce) >$	$\{28\}$	1	$\{10\}$	1	2	$\{28, 10\}$	2

Table 6: Example of Projection in modified and unmodified subtrees

where $N_P^U = \{n_1, n_3, n_5\}$ are unmodified and $N_P^M = \{n_2, n_4\}$ are modified nodes. Modified nodes mean those nodes which were affected due to the addition of the IncDB. So, $S_P^{D'} = C(n_1)+C(n_2)+C(n_3)+C(n_4)+C(n_5)$ denotes the total updated support of the pattern in D' and $S_P^{db} = C''(n_2)+C''(n_4)$ denotes the additional support of the pattern occurred due to IncDB (db) where C''=$C - C'$ for each modified node.

Definition 7 (Infrequent to Frequent Transition Property) Suppose, we had an infrequent pattern P (up to previous pass), minimum support threshold value min_sup, previous database D and updated database D'. The previous minimum support of a pattern to be frequent was $\delta = \lceil min\_sup \times |D| \rceil$ and current is $\delta' = \lceil min\_sup \times |D'| \rceil$. So, the additional support of P (S_P^{db}) from the modified subtrees or IncDB needs to be $\geq (\delta' - \delta + 1)$ for P to be frequent. For the first pass ($\delta = 0$), the constraint is $S_P^{db} \geq (\delta')$ as D is empty.

IncTree-Miner's basic mining procedure is similar to Tree-Miner except the key point is, it separates the projection nodes of a pattern into two groups, modified nodes and unmodified nodes. It calculates the result separately and merges them and that helps track the changed patterns efficiently. In Fig. 7 (a), we have shown the idea of our pattern mining approach regarding the extensions in modified and unmodified subtrees along with the used terminologies and mathematical relations. Similar to Tree-Miner, our extension functions work based on the proposed breadth-first technique and the complete projection nodes for the patterns are stored in BPFSP-Tree(B). In Table 6, we have shown two examples of how, IncTree-Miner performs

step	D_{status}	D'_{status}	P	N_P^M	N_P^U	$S_P^{D'}$	S_P^D	$sList^M$, $iList^M$	$sList^{M'}$, $iList^{M'}$
1	F	F	<(c)>	{4, 8, 16, 33}	{12,21}	9	5	{a, b, c, d,e}, {d,e}	{a,b,c,d}, {d}
2	NF	F	<(c)(a)>	{22, 25, 29,35, 40}	{11}	7	1	{a, b, c, d}, {b, c,d}	{}, {b,c}
3	NF	F	<(c)(ab)>	{23, 26,36}	{ }	3	0	{ }, {c}	{ }, {c}
4	NF	F	<(c)(abc)>	{24, 27, 37}	{ }	3	0	{ }, { }	{},{}
5	NF	F	<(c)(ac)>	{24, 27, 37,41}	{ }	4	0	{ }, { }	{},{}
6	NF	F	<(c)(b)>	{23, 30, 26, 36}	{}	4	0	{a, b, c, d},{c,d}	{},{c}
7	NF	F	<(c)(bc)>	{24, 27, 37}	{}	3	0	{ }, { }	{ }, { }
8	NF	F	<(c)(c)>	{24, 27, 31, 37, 41}	{}	5	0	{ a, b,c, d}, {d}	{ }, { }
9	F	F	<(c)(d)>	{5, 32}	{13}	3	2	{ a, b,c, d}, { }	{ }, { }
10	NF	F	<(cd)>	{34}	{9}	3	1	{a, b, c, d}, { }	{a}, { }
11	NF	F	<(cd)(a)>	{35, 40}	{11}	3	1	{a}, { }	{ }, { }

Fig. 9: Simulation of IncTree-Miner for the patterns with prefix $< (c) >$

projection in modified and unmodified subtrees separately and then merges them to calculate the complete results for each pattern. To refer the nodes, the IncSP-Tree shown in Fig. 3 has been used.

We represent the pattern formation technique in Algorithm 5 and the incremental mining technique in Algorithm 6. A pattern $P\gamma$ can be in three states. It Was previously frequent(Algorithm 5, line 3-10) or its previous complete support is in NIB(lines 11-20) or it was previously infrequent(lines 21-32). We simulate each case separately. We always first project in modified subtrees(line 2) and based on the results we take decision to perform projection in unmodifed subtrees. We start with the items which were found in IncDB to make the initial $sList^M$ and $iList^M$(Algorithm 6), gradually shrink them and track the patterns which were effected or had modifications in underlying subtree. We clear the NIB for the items for which no projection was done in current iteration to maintain consistency. Finally, we remove the unmodified previously frequent but currently infrequent patterns from BPFSP-Tree in a bottom-up manner(Algorithm 6, lines 27-28). In Fig. 9, a small visualization of IncTree-Miner has been shown where it can be seen, how IncTree-Miner recursively discovers the patterns with prefix $< (c) >$. For each pattern, first its status in old and new database, then its corresponding modified and unmodified projection nodes, its total support calculated from modified and unmodified subtrees, its corresponding two lists and shrinked lists are shown. In Fig. 7 (e), the status of the BPFSP-Tree for the patterns with prefix $< (c) >$ in the old and new database is shown which also reflects how pattern distribution can significantly vary due to database modification. To refer the nodes, the IncSP-Tree shown in Fig. 3 has been used.

IncSP-Tree provides structural advantage to perform faster pattern search in the database. So, it can also help to improve mining performance in buffer based solutions by performing faster pattern generation. Moreover We can also develop a memory resilient IncTree-Miner by not keeping the projection information and performing runtime necessary pattern search using IncSP-Tree. A short visualization of memory resilient version is shown in Fig. 7 (d). In the figure, we have shown that, when we need to know the frequency status of $< (\alpha)(\beta)(\gamma) >$ we perform the projection of it on fly using our tree structure's link properties which help us to perform faster traversals alongside other strategies. Our general proposal focuses on keeping only the frequent pattern's information (pattern and support) to avoid the problem of concept drift. But we can also embed some over computing buffer based approaches here.

Algorithm 5 Pattern Formation Block of IncTree-Miner

procedure PATTERNFORMATION($P\gamma$, N_P^M, B_P)
  $\forall n \in N_P^M$ perform implicit projection in modified subtrees for γ
  ▷ previously frequent, $B_{P\gamma}$ node exists
  if $P\gamma \in B_P.child$ **then**
    if $S_{P\gamma}^{D'} \geq \delta'$ **then** update support and projection in $B_{P\gamma}$.
    else ▷ Got Infrequent
      Remove $B_{P\gamma}$ and its subtree, adjust end-links.
      if (complete projection done) **then**
        ▷ Save support in NIB
        $B_P.NIB[\gamma] \leftarrow S_{P\gamma}^{D'}$
  else if $\gamma \in B_P.NIB$ **then**
    ▷ Previous complete support in NIB
    if $S_{P\gamma}^{D'} \geq \delta'$ **then**
      Perform projection in unmodified subtrees for γ.
      Create node $B_P.child[\gamma]$, adjust end-links, remove $B_P.NIB[\gamma]$
    else ▷ Still Infrequent
      if (complete projection done) **then**
        $B_P.NIB[\gamma] \leftarrow S_{P\gamma}^{D'}$
      else ▷ Not full projection calculated
        Delete $B_P.NIB[\gamma]$
  else if $P\gamma \notin B_P.child \cap \gamma \notin B_P.NIB$ **then**
    if $S_{P\gamma}^{db} \geq \delta' - \delta + 1$ **then**
      ▷ Infrequent to Frequent Transition
      Perform projection in unmodified subtrees for γ.
      if $S_{P\gamma}^{db} + S_{P\gamma}^{D} = S_{P\gamma}^{D'} \geq \delta'$ **then**
        Create $B_P.child[\gamma]$,adjust links
      else ▷ Infrequent Pattern
        if (complete projection done) **then**
          $B_P.NIB[\gamma] \leftarrow S_{P\gamma}^{D'}$
        if $H_{P\gamma}^{M} + H_{P\gamma}^{U} < \delta'$ **then**
          ▷ Heuristic Pruning during SE
          Remove γ from $iList$ of pattern P
  if $P\gamma$ is frequent $\cap$ $N_{P\gamma}^M \neq \{\}$ **then**
    ▷ $P\gamma$ has update in subtree, need further checking
    Return $N_{P\gamma}^M$
  else Return *null* ▷ infrequent/no update in underlying subtree for $P\gamma$

Algorithm 6 IncTree-Miner

Globals: minimum support value to be frequent in D' and D is δ' and δ respectively.
procedure INCTREEMINER(P,N_P^M,$sList^M$,$iList^M$,B_P)
  Reduce $sList^M$ and $iList^M$ based on CETable
  ▷ Initiating Next iteration nodes
  $N_{SE} \leftarrow \{\}, N_{IE} \leftarrow \{\}$
  ▷ SE, Tracking the modified patterns
  for each item $\gamma \in sList^M$ **do**
    $N_{SE}[\gamma] \leftarrow$ PatternFormation($P\{\gamma\}$,N_P^M, B_P)
    ▷ Infrequent/No modifications
    if $N_{P\{\gamma\}}^M = \{\}$ **then** $sList^M \leftarrow sList^M - \{\gamma\}$
  ▷ IE, Tracking the modified patterns
  for each item $\gamma \in iList^M$ **do**
    $N_{IE}[\gamma] \leftarrow$ PatternFormation($\{P\gamma\}$,N_P^M, B_P)
    ▷ Infrequent/No modifications
    if $N_{\{P\gamma\}}^M = \{\}$ **then** $iList^M \leftarrow iList^M - \{\gamma\}$
  ▷ For Consistency over mining iterations
  if $P\gamma$ is not projected **then**
    Remove all γ from $B_P.NIB$
  ▷ Reduced lists
  $sList' \leftarrow sList^M$,$iList' \leftarrow iList^M$
  ▷ Recursive SE for effected patterns
  for each modified item $\gamma \in N_{SE}$ **do**
    IncTreeMiner($P\{\gamma\}, sList', \{\epsilon | \epsilon \in sList' \cap \epsilon > \gamma\}, B_{P\{\gamma\}}$).
  ▷ Recursive IE for effected patterns
  for each modified item $\gamma \in N_{IE}$ **do**
    IncTreeMiner($\{P\gamma\}, sList', \{\epsilon | \epsilon \in iList' \cap \epsilon > \gamma\}, B_{\{P\gamma\}}$).
procedure REMOVEINFREQUENTPATTERNS
  Traverse through *end-link*s of leaf nodes in BPFSP-Tree and remove the infrequent patterns ($S_P^{D'} < \delta'$) in bottom up manner.

5 Performance Evaluation

In this article, we have proposed two novel tree-based solutions for static and incremental mining problems, namely Tree-Miner based on SP-Tree and IncTree-Miner based on IncSP-Tree. So, during comparison, we considered those popular and state-of-the-art algorithms which were designed as a new mining technique to solve this problem. We also designed a new support counting strategy which helps detect patterns' infrequency early along with some additional pruning strategies.

To compare Tree-Miner, we have considered three popular state-of-the-art mining algorithms, PrefixSpan[28], CM-SPAM and CM-SPADE[12] and to compare IncTree-Miner we considered IncSP[21] and PBIncSpan [5]. We have chosen IncSP and PBIncSpan because they are two of the most prominent mining techniques to solve ISPM problem. As these are based on a single minimum support threshold parameter and has similar key factors, there is a solid ground to compare with. We have introduced a new mining approach, so we considered those which did the same. All the implementations were in Python language and the experiments were conducted on a 64-bit machine having Intel Core i5-8265U CPU @ 3.90GHz × 8, 32 GB RAM, Windows 10 OS.

5.1 Dataset Description and Parameters

We have experimented our proposed solutions over various real-life and synthetic datasets. The performance was consistent and matched our intuition. To discuss the performance we will use the results of the datasets shown in the Table 7. We have shown the necessary and important information for each dataset in Table 7. All the real-life datasets are collected from SPMF: A Java Open Source Data Mining Library[2] and synthetic datasets are generated using *IBM Generator* by applying different parameters. In the real-life datasets, all the events had only one single item, so we randomly merged consecutive events to construct multiple itemed events. In Table 7, we have also shown the average number of itemsets in each sequence (C) and average number of items per itemset (T) for each dataset after the described modification.

To evaluate our incremental solution, we had to create an incremental mining scenario. To construct the

[2] `https://www.philippe-fournier-viger.com/spmf/index.php?link=datasets.php`

incremental database and represent the common phenomena, we followed the approach mentioned in the relevant literature [21]. Initially, We had the raw database D_{raw}, we divided it into two databases, old database D and incremental database db. Final updated database was D'. We have given description of the used variables in Table 8. To understand how the variables work here, we can construct an example using Table 3 and 4. Here Table 3 will work as D_{raw}, first iteration of Table 4 will work as D and second iteration of Table 4 will work as db. D_{Append} contains 2 sequences or sids ($sid = 1$ and $sid = 4$) and D_{Insert} contains 4 sequences ($sid = 7, 8, 9, 10$). So here $R_{new} = \frac{4\times100}{10}\% = 40\%$, $R_{com} = \frac{2\times100}{10}\% = 20\%$ and $R_{prev} = \frac{(\frac{5}{8}+\frac{2}{5})\times100\%}{2} = 51.25\%$. Here $\frac{5}{8}$ comes from sequence 1 ($sid = 1$) which means in the total length of 8 items 5 items appeared in the first pass and the remaining 3 items in the second pass. Same concept applies for $\frac{2}{5}$ for the sequence 4. For conducting experiments, for each dataset, we chose $R_{new} = 10\%, R_{com} = 50\%$ and $R_{prev} = 80\%$ similar to [21].

We have defined density ratio of a dataset as $d^* = \frac{\text{avg. sequence length}\times100}{(\#)\text{unique items}}$ which is the combination of average sequence length and number of unique items. To evaluate our proposals we have used datasets of both types, highly densed (with high ratio of d^*) and comparatively much lesser densed.

5.2 Tree-Miner

In this section, we will evaluate Tree-Miner based on various metrics and analyze the results.

5.2.1 Runtime, Memory Usage and Scalability

In Fig. 10 (a)-(g) we present the runtime performance of Tree-Miner. From the corresponding figures, it is obvious that our proposed Tree-Miner based on SP-Tree performs comparatively better than other static mining algorithms. SP-Tree is a compact representation of the database which provides huge structural control over it. Through the next links, we perform efficient traversals in the database which ultimately helps to generate the patterns faster. We also use a set of pruning techniques that significantly reduce the search space. We adopted all the previous pruning techniques and also incorporated some newer ones which makes it a very efficient mining algorithm. In the figures, in lower thresholds, our performance improvement is quite visible because we need to encounter a huge number of patterns and Tree-Miner efficiently discovers them whereas in upper thresholds the performance is quite close because the number of patterns is very small. The performance gets improved in dense datasets through node overlapping characteristics and in sparse datasets through faster database reduction and projection though next links. Our SP-Tree structure is enough to calculate the complete support of the patterns having no necessity to maintain additional structure for each pattern which is also an important factor to improve performance. In Fig. 10 (a)-(g) we have shown the results for Sign, Bible, BmsWebView1, MSNBC, Kosarak, L20k-C10-T5-N1K and L50k-C10-T2.5-N10K datasets respectively.

up to now, the comparison has been shown with the base mining approaches as the proposed solution was designed as a new mining algorithm for SPM problem. We have also tried to apply the proposed solution in some different types of SPM problems which are mostly application centric and borrows concepts from base pattern mining techniques.

In this regard, we compared our proposal with [7] and [11] two recent literatures. In [7], they mined discriminative sequential patterns using significance threshold. Here they first generate all the frequent patterns using GSP, then conduct multi level correlation analysis in such regard. In [11], they designed a context based e-learning recommendation system using the utility of SPM algorithms where GSP was used in such regard. We applied our proposal in these literatures and could directly embed it without any modification. As pattern growth approaches are significantly faster compared to GSP, our Tree-Miner improved the performance of the pattern generation level to a significant amount leading to an overall improvement. Some experimental results have been provided to support the claim in Fig 11. For the comparisons, all the base datasets have been used here, multiple single itemed sequential events.

In the figure (a) and (b) shows the results of Tree-Miner embedded performance improvement where Disc denotes the applied solution in literature [7] whereas Tree-Miner with Disc denotes our solution. For the comparison here, significance level (α) as per stated in the article is fixed and the support threshold value is varied.

Similarly, some results have been shown after comparing with [11]. In this article, a recommendation system is built based on context awareness for e-learning purpose. For this purpose, they needed relevant web click stream data, users' knowledge profile and ratings for each resource upon which they calculated the suggestion metrics. To bring the context aware recommendation, they used GSP to filter out the suggestions after generating a set of frequent sequential patterns or web logs visited by the users. Keeping all their contributions align, the comparison comes here is how the pattern generation process can be made faster using our

Name	Description	Sequence Count	Unique Items	Avg. sequence Length	C	T	d^*
Bible	Conversion of Bible	36,369	13,905	21.64	10.97	15.49	0.16
MSNBC	Click Stream	31790	17	13.23	7.83	10.81	77.82
BMSWebView1	Click Stream	59601	497	2.43	2.19	6.2	0.49
Sign	Sign Language	730	267	51.997	25.99	28.49	19.47
Kosarak	Click Stream	990000	41270	8.1	3.7	6.23	0.02
L20k-C10-T5-N1K	Synthetic Dataset	20000	1000	44.21	10	5	4.42
L50k-C10-T2.5-N10K	Synthetic Dataset	50000	10000	21.79	10	2.5	0.22

Table 7: Dataset Summarized Information

Variable	Description
D	Old database
db	Incremental database, $db = D_{Insert} \cup D_{Append}$.
D'	Updated database, $D' = D \cup db$.
R_{new}	*New sequence Ratio:* Percentage of sequences completely removed from D_{raw} to put in D_{Insert}. $\lvert D_{Insert}\rvert = R_{new} \times \lvert D_{raw}\rvert$
R_{com}	*Common Sequence Ratio:* Percentage of common sids picked from D_{raw} to put into D and D_{Append}. $\lvert D_{Append}\rvert = R_{com} \times \lvert D_{raw}\rvert$.
R_{prev}	*Previous Appearing Ratio:* Avg. splitting ratio for the common sequences lying in D and D_{Append}. If $R_{prev} = 80\%$, then for each common sequence (in avg.) 80% items will be in D and 20% will be in D_{Append}.
L	Total Number of sequences
C	Avg. number of itemsets per sequence
T	Avg. Number of items in each itemset
N	Number of distinct Items

Table 8: Variables' Description

proposal. In Fig. 11 (c) and (d), we have shown the performance of GSP with Tree-Miner in two click stream datasets. Here we have used the base datasets, single event with multiple itemed events for the comparison.

We have also compared our solution to [8], a novel Mining with Decision Diagram (MDD) based approach to solve constraint-based sequential pattern mining. But to make a fair comparison with the literature's solution, we had to modify our algorithm and datasets to some extent. A summary of such modifications are shown as below,

- In the literature, how multiple itemed event will be handled was not proposed. So, for the comparisons, all the base datasets have been used here, multiple single itemed sequential events.
- The literature constructed their proposed MDD nodes over an initially given support threshold value by pruning nodes. This idea can easily be embedded in our solution as ours is more generic. When the next links are calculated, we can apply the breadth-first and upper bound based pattern generation concept to reduce many next link connection.
  For example, we want to generate $\alpha \rightarrow \alpha\beta$. Here $N_\alpha = \{2, 3, 4\}$, $S_\alpha = 9$, $C(2) = C(3) = C(4) = 3$, $\delta = 8$. We want to calculate next links for β from each $v \in N_\alpha$. Let from 2 we need to make a next link connection with 5 for β where $C(5) = 1$. So using breadth-first technique, we can see $S_{\alpha\beta} = 9-3+1 < \delta$. Thus we can stop constructing all the next links for $\alpha \rightarrow \beta$ and its super patterns for all the remaining nodes along with pruning this pattern from the generation.
- Two solutions' comparisons are done only over support threshold constraint. The result is shown in Fig. 12.

From the figures, it can be observed that Tree-Miner with SP-Tree performs comparatively better. The main reasons lie in its larger flexibility, e.g., multiple event handling, easily modifiable, bit based usage advantages, etc and pruning strategies. The results for Kosarak, MSNBC and BMSWebView1 have been shown in the figure. As previously stated, using next links we can perform faster jumps in the database along with using the rich pruning techniques a good amount of pattern searching can be omitted.

Due to the structured representation of the information, our solution needs comparatively more memory. We maintain SP-Tree and co-occurrence information which gives control over the database and helps significantly during pattern extension. Prefixspan uses database projection technique, CM-SPADE uses lattice alike structures and CM-SPAM uses bit based array structures to perform projection and calculate the sup-

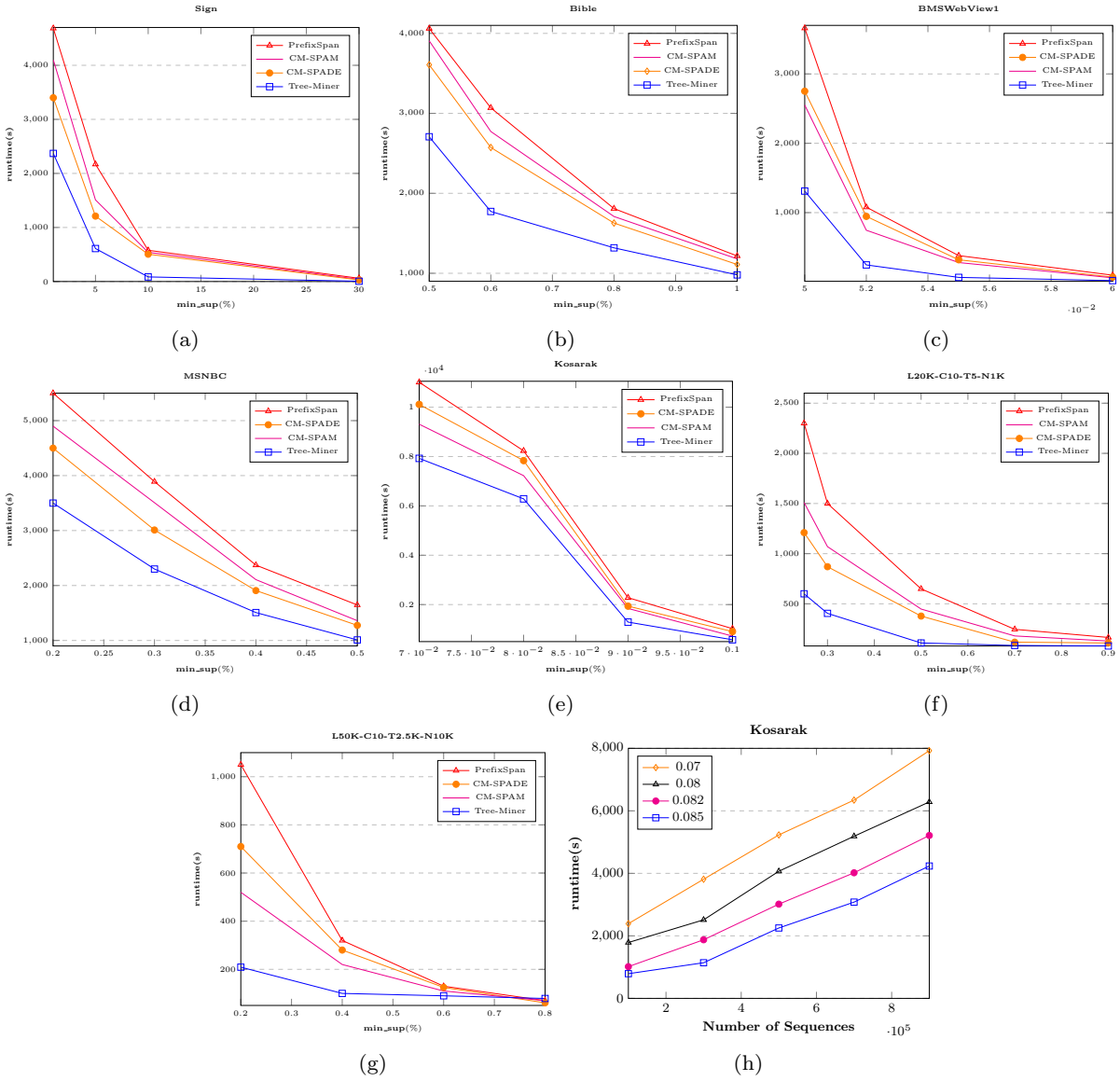


Fig. 10: (a) - (g): Runtime Comparisons, (h) Scalability Comparison for Tree-Miner

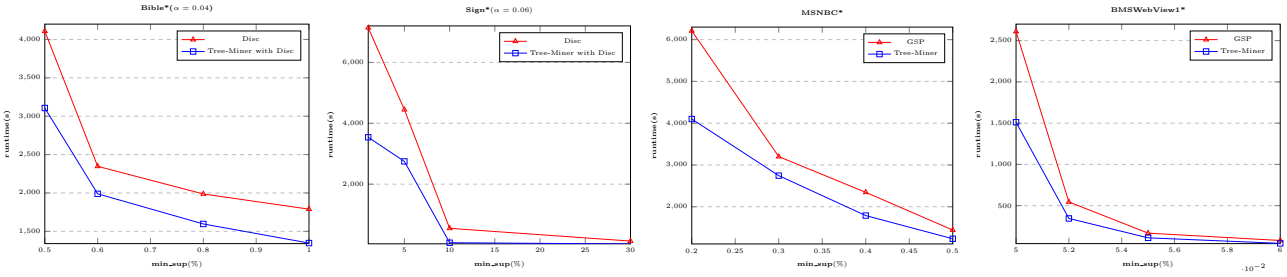
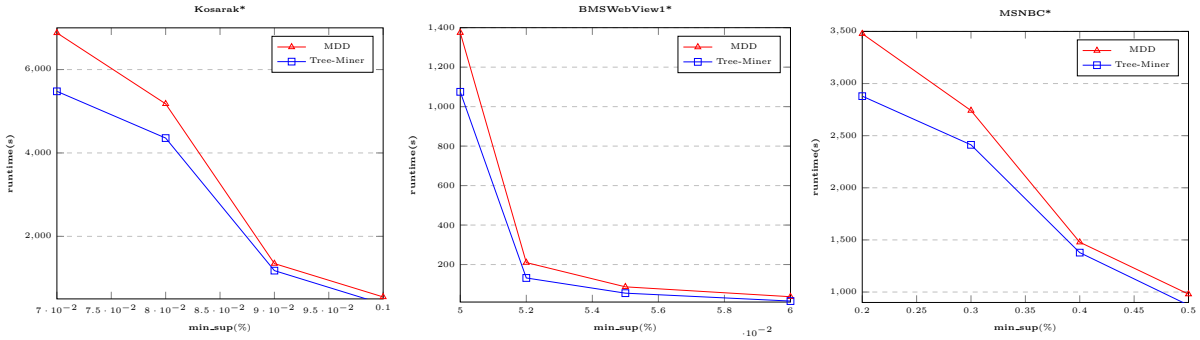


Fig. 11: Runtime Improvement using Tree-Miner over GSP based literature

port of a pattern. During pattern extension, the compared algorithms create and maintain some sort of additional supporting structures in runtime whereas our proposed solution uses the base tree structures' nodes for this purpose. Through some additional memory usage, we gain significant amount of improvement in min-



(a) (b) (c)

Fig. 12: Runtime Comparisons between Tree-Miner and MDD

ing which we will show shortly and considering the runtime improvement this should be tolerable. We present an analysis in Fig. 13 for two datasets. From the figure, it can be seen that our solution comparatively needs some additional memory and we have already discussed the underlying reasoning behind it. The difference will be comparatively closer in denser datasets due to tree's node overlapping characteristics.

We have also conducted a scalability analysis of Tree-Miner and presented the result in Fig. 10 (g) experimenting over the lage *Kosarak* dataset. We started with 100000 transactions, gradually increased it and recorded the performance for various *min_sup*. The corresponding figure shows the linear scalability of the solution. Another important concern can be, as our solution is based on defined structures, do their construction times create any bottleneck during mining. For this purpose we present an analysis in Table 9 where we can see that, the total construction time is quite insignificant compared to the total mining time and does not create bottleneck. In the last three columns of Table 9 we have also shown the corresponding runtime performance improvement in mining over the comparing algorithms (*Pr*: PrefixSpan, *SP*: CM-SPADE, *SM*: CM-SPAM). This basically states the significance of how much we can improve mining performance with very small time cost in pre-processing.

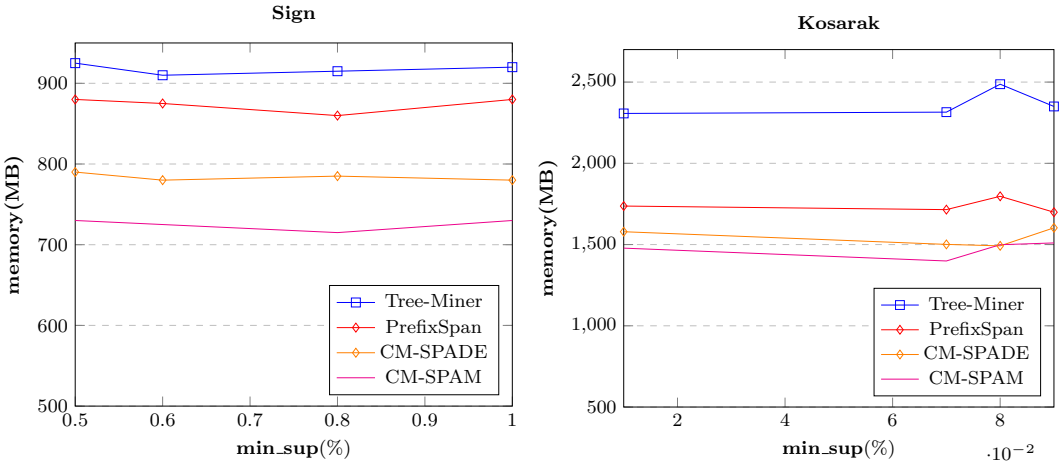
5.3 IncTree-Miner

In this section, we will analyze the performance of IncTree-Miner based on various metrics.

5.3.1 Runtime, Memory Usage, Scalability

Our IncTree-Miner contains all the novelty of Tree-Miner and designs a set of concepts to implicitly track the incremental database which ultimately helps to efficiently mine those patterns which are affected due to the addition of incremental database. It separates the modified and unmodified subtrees, performs projection separately and combines the results to provide the final output. Using modified next links and support count attributes we efficiently detect the modified subtrees and the updated patterns with their changed support. During pattern extensions, first we project in the incremental database and from there using *infrequent to frequent* transition property we first detect which infrequent patterns have chance to be frequent and then decide to perform projection in the remaining database for them. This approach also helps to efficiently update the frequency of the existing frequent patterns which are affected due to the incremental database also leading to the detection of the previously frequent and currently infrequent patterns. We also use BPFSP-Tree as pattern storage which keeps the projection information of the patterns using IncSP-Tree nodes. It reduces the number of DB scans and efficiently removes the infrequent patterns using the bottom-up strategy. It also maintains NIB buffer which makes use of the cost to calculate the support of an infrequent pattern by giving an idea regarding previous support during the following iterations. Sequence Summarizer also helps to incrementally update the co-occurrence information.

In Fig. 14 (a)-(g), we have shown the runtime performance of IncTree-Miner with $PBIncSpan$ and $IncSP$. From the figure, it is obvious that our proposed algorithm improves runtime by a significant amount. IncSP is based on candidate generation and testing paradigm and it is bound to be slow compared to pattern growth algorithms. PBIncSapn is an efficient solution that follows the pattern growth approach along with applying two efficient pruning mechanisms, width and depth pruning. But these are basically a subset of pruning mechanisms we maintain. PBIncSpan saves the projec-



(a) (b)

Fig. 13: Memory Comparisons for Tree-Miner

Dataset	Construction Time (Sec.) (C)	Mining Time (Sec.) (M)	min_sup (%)	$\frac{C}{M}$ (%)	M vs Pr (%)	M vs SP (%)	M vs SM (%)
Bible	40	2907	0.5	1.3	29	20	26
BMSWebView1	5	1321	0.05	0.3	64	52	45
Sign	16	2448	1	0.6	48	28	40
MSNBC	14	3700	0.2	0.3	33	18	24
Kosarak	130	7925	0.07	1.6	29	22	15
L20-C10-T5-N1K	10	1012	0.025	1	56	17	33
L50-C10-T2.5-N10K	20	2010	0.02	0.9	76	64	54

Table 9: Construction Time Vs Mining Time

tion information using pseudo projections of the database to reduce the DB scan, whereas we use the compact IncSP-Tree node pointers based BPFSP-Tree for this purpose. Similar to Tree-Miner our performance improvement is quite visible at lower thresholds. As the number of patterns (both frequent and updated) is very small at higher thresholds, the improvement is not much differentiable.

Upto now, we have compared with the base proposals related to incremental mining that aligns with the motivation of the proposed solution. Now, we will provide some analysis through comparing with a prominent literature that does not completely aligns with the motivation of the proposed work but goes closely with it, to understand our solution's efficacy in improving runtime.

For the comparison MR-INCSPM[10] has been chosen, a Map Reduced framework for ISPM problems. As the proposed solution works for a single machine environment, here for MR-INCSPM such environment has been considered for fair ground analysis. MR-INCSPM worked in backward extension with its own co-occurrence table definition and pruning strategies. So, these factors also provided important points to conduct a critical analysis.

In Fig.15, we have shown the results for some of the datasets. From the figures, it can be noted that, the performance of IncTree-Miner is comparatively better. The main issue, in single machine based MR-INCSPM is, it generates patterns with sort of prefixspan or database projection alike technique but with backward extension. Here the main improvement factor of ours is faster traversal through links compared to scan based moves. Alongside, their proposed pruning strategies are a subset of the strategies we use for our pruning. So, altogether the performance improvement is achieved.

In [9], pre-large concept was proposed where additional set of patterns which are not exactly frequent are pre-computed, so that the number of database re-scans can be reduced. There the authors use additional set of thresholds, upper(s_u) and lower(s_l) minimum support thresholds to perform the additional patterns' pre-computation. The performance of database re-scans can be improved using the proposed IncSP-Tree struc-

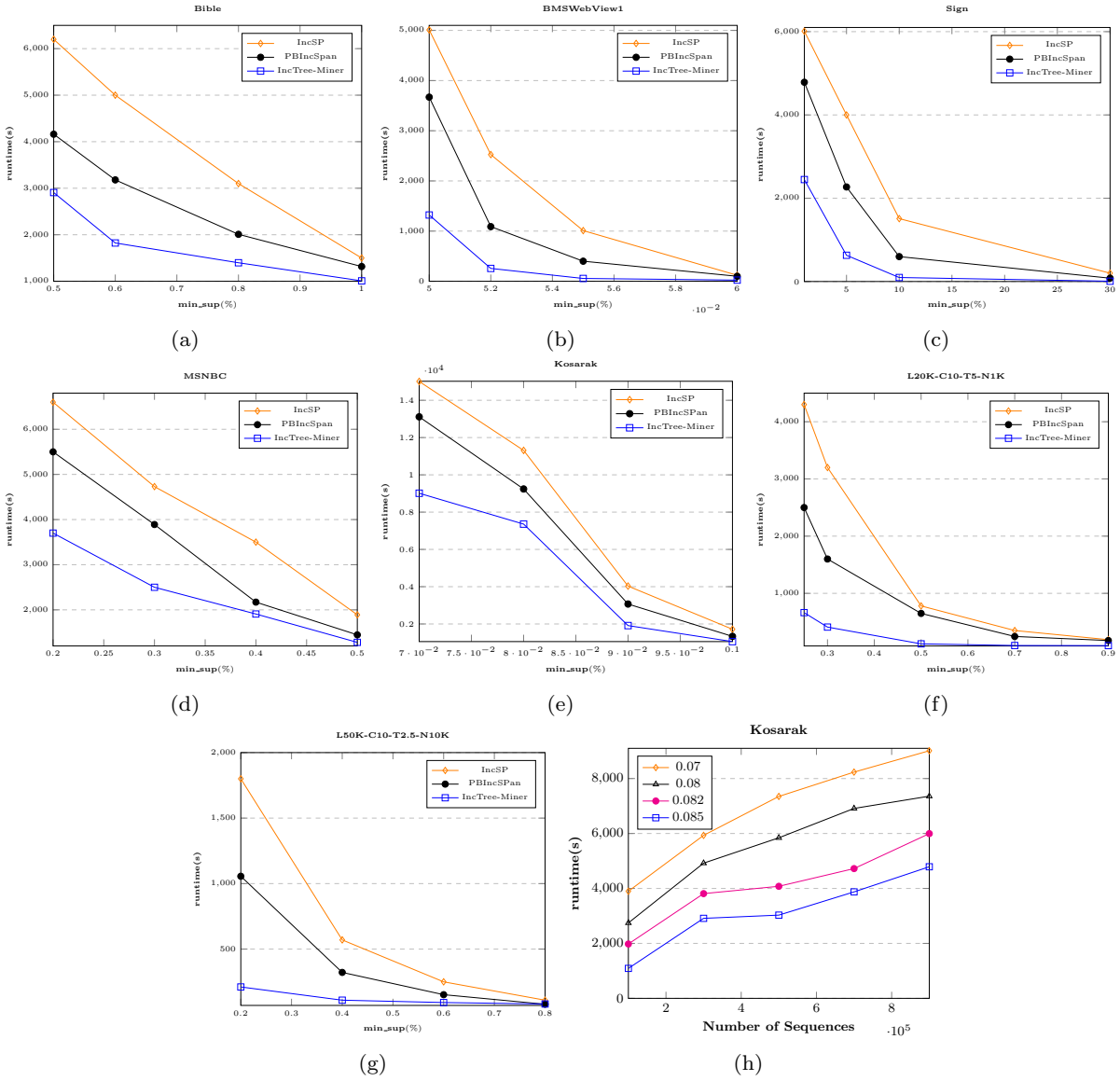
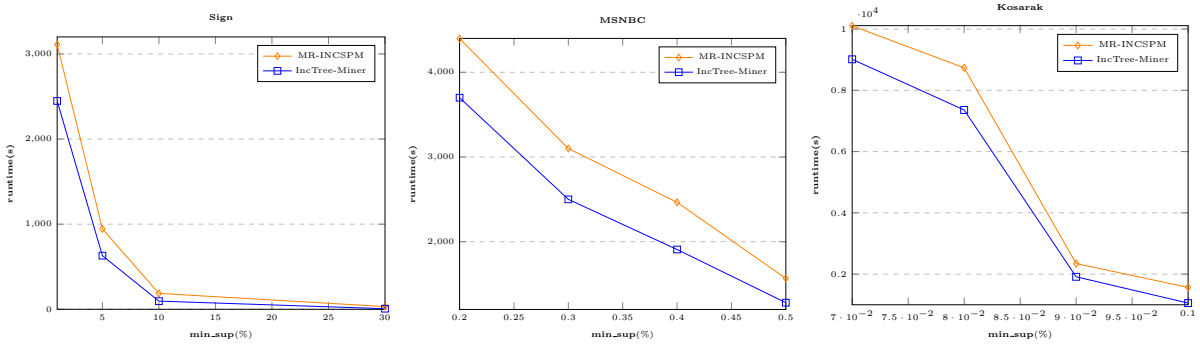


Fig. 14: (a)-(g): Runtime Comparison, (h) Scalability Comparisons for IncTree-Miner

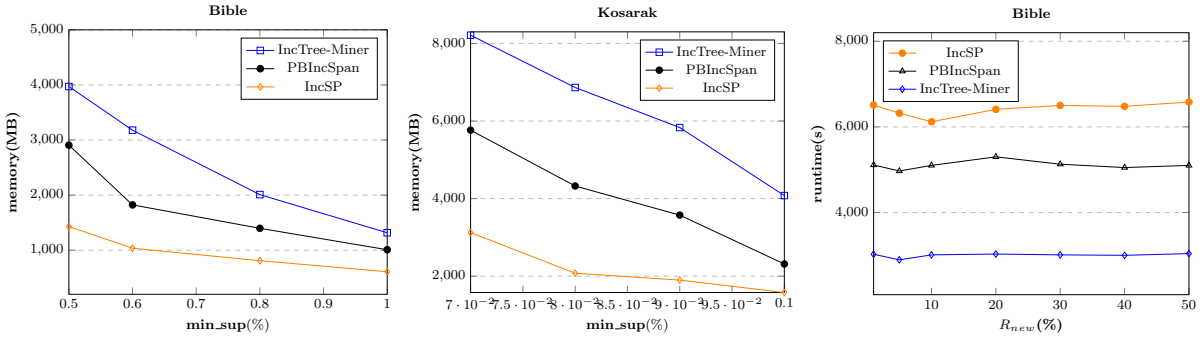
ture through using its next links and modified next links. These over-computation based approaches suffer greatly when concept drift appears along with the critical selection of the empirically set thresholds.

ISPM algorithms generally need more memory compared to static mining algorithms because they store the frequent patterns' information which is used in the successive iterations. We show a memory usage analysis in Fig. 16 (a)-(b) over Bible and Kosarak dataset. IncSP stores the support of the prior frequent patterns. So, its memory usage is dependent on the number of frequent patterns. PBIncSpan and IncTree-Miner both keep the previous frequent patterns' support along with their projection information where PBIncSpan uses pseudo projection and IncTree-Miner uses the compact IncSP-Tree node references. But IncTree-Miner stores the sequential database in tree format along with maintaining some data structures leading to comparatively more memory usage than PBIncSpan. As memory usage is directly related to the compactness of the tree, so in dense datasets, this metric's performance is comparatively closer to PBIncSpan. Our incremental solution is the extension of prior proposed static solution and both maintain almost similar types of data structures. In the earlier section, we have discussed how this additional usage gives us significant improvement in mining which also applies here. Being very flexible solution, we have our own memory resilient version. In the memory resilient version, we do not store pattern's projection in-



(a) (b) (c)

Fig. 15: Runtime Comparison with MR-INCSPM



(a) (b) (c)

Fig. 16: (a)-(b) Memory Evaluation, (c) Performance over R_{new}, for IncTree-Miner

formation which reduces the memory to a great extent with additional time cost during mining.

Similar to Tree-Miner, we also conducted a scalability test for IncTree-Miner and presented the result over the *Kosarak* dataset in Fig. 14 (h). We started with few transactions, gradually increased it and recorded the results by varying *min_sup*. The corresponding figure shows the linear scalability of the solution. To conduct the experiment, we chose $R_{new} = 10\%, R_{com} = 50\%$ and $R_{prev} = 80\%$ over the considered number of transactions. In the Tree-Miner section, we have shown that the structural solution does not create a bottleneck to construct them rather provides improvement in mining time. As IncSP-Tree is an incremental version of SP-Tree it also maintains similar characteristics.

5.3.2 Effect on Different R_{new} and R_{com}

To evaluate the performance over the ratio of the completely new sequences (new sids) we conducted experiment by varying the percentage of R_{new} over *Bible* dataset by keeping $R_{com} = 50\%$, $R_{prev} = 80\%$ and $min\_sup = 0.3\%$. We have shown the result in Fig. 16 (c). We started with 1% and gradually increased it to 30% and we always had two iterations to output the final set of patterns. We wanted to observe if the increased ratio of R_{new} affects the solution's performance or not. From Fig. 16 (c) it is clear that IncTree-Miner's performance does not degrade due to the R_{new}'s increasing ratio compared to PBIncSpan and IncSP. With R_{new}'s increment the number of newer patterns' and updated patterns' get increased in the second pass and the number of frequent patterns' get decreased in the first pass. So, in our described scenario, the ISPM algorithm's performance should be almost linear and Fig. 16 (c) supports our intuition.

Similar to R_{new}, we also evaluated our solution by changing the percentage of R_{com}. For experiment, we had set $R_{new} = 10\%, R_{prev} = 80\%$ and varied R_{com} with $min\_sup = 0.5\%$. Like the previous discussion, we got almost similar types of results which matched our intuition that IncTree-Miner is not affected due to the increment of the R_{com} and performs comparatively better than PBIncSpan and IncSP.

5.4 Evaluation of Breadth-First Based Support Counting Technique

In this section, we will evaluate the performance of the proposed breadth-first based support counting technique to understand how fast it can detect an infrequent pattern and stop support counting. To guide our mining process we use the co-occurrence information of the items stored in *sList* and *iList*. With the suffix extension of the pattern these two lists shrink. During support calculation of the extensions, we discover some patterns' infrequency. The algorithm's runtime improves depending on how quickly it is able to detect the infrequency rather than performing complete projection. In Fig. 17, we present an analysis to understand how quickly our proposed pruning technique detects infrequent patterns. In the x-axis, we have provided the percentage value of the projection size and in the y-axis, we have provided the percentage of infrequent patterns been detected. For example, in the *L20-C10-T2.5-10K* dataset, at 50% value in the x-axis, we get a 45% value in the corresponding y-axis, which denotes that, of all the infrequent patterns been tested, 45% of them are detected performing only 50% projection of their corresponding complete projected database. From, Fig. 17, it is clear that our proposed support counting technique is able to detect most of the infrequent patterns without performing a complete projection of the underlying sub-database. We have shown the corresponding *min_sup* values beside each legend in Fig. 17. The figure is generated by executing Tree-Miner.

5.5 Effectiveness in Interactive Mining

The proposed SP-Tree structures hold "build-once-mine-many" property which states that the structures are capable of performing multiple mining iterations based on users' requests without bringing any change to the existing structures for different *min_sup* values.

The most difficult scenario in interactive mining is, the gradual decrease in *min_sup* (20% $\rightarrow$ 10% $\rightarrow$ 5%, etc) where with this decrease a new set of previously infrequent patterns may become frequent. So, to discover those patterns, we need to perform a mining iteration. Here, using BPFSP-Tree's patterns' projection information we can efficiently discover the newly frequent patterns which will be the super patterns (apriori property) of existing frequent patterns. Through projection information, we can directly reach a pattern's corresponding nodes and start expanding from there using the next links. In Table 10(second group of column) we have shown some results related to this scenario. If, we had to start mining from scratch then the time usage would have increased significantly. Here the mining time gets reduced because already a group of patterns have been calculated alongside can use projection information and faster pattern search procedure.

The best case scenario, in interactive mining comes from the gradual increase in *min_sup* (5% $\rightarrow$ 10% $\rightarrow$ 20%, etc) where with this increase no new patterns become frequent rather a group of previously frequent patterns may become infrequent. To efficiently remove these newly infrequent patterns, we can use Bottom-up traversal strategy of BPFSP-Tree which will help traverse lesser patterns in this regard. In Table 10(third group of column), we have shown some results related to this scenario.

5.6 Analytical Novelty of IncTree-Miner with IncSP-Tree

Our proposed SP-Tree and IncSP-Tree provides an efficient structural manner to store the the sequential database leading to an overall improved runtime during mining. Our proposed Tree-Miner is an efficient algorithm to mine sequential patterns from static database and our proposed IncTree-Miner is an efficient algorithm to solve the incremental mining problem. IncTree-Miner's performance depends on single support threshold parameter where as some literature have adopted buffering concepts [6], multiple threshold based concepts [9] to solve the ISPM problem. Main problems of these approaches are, their performance solely depends on these empirically set thresholds' values. As it is difficult to guess the database characteristics prior, it is very difficult to set these additional parameters appropriately which leads to additional complexity and wastage in both memory and runtime as they might need to pre-compute and store huge amount of infrequent (or can be regarded as semi-frequent) patterns' information which might never get frequent. Also, these approaches are severely affected due to seasonal concept drift. Concept drift basically indicates the sudden shifting in frequent patterns' distribution where a huge number of previously infrequent patterns suddenly become frequent and most of the previously over-computed semi-frequent patterns also do not bear that transition characteristics. In this case, the existing additional parameter based solutions have to re-mine the complete raw database to discover such patterns.

But as our designed solutions store the databases in a structured format, here pattern searching is comparatively faster. Also by storing the projection information of the frequent patterns, we get an advantage to efficiently track the newer frequent patterns which

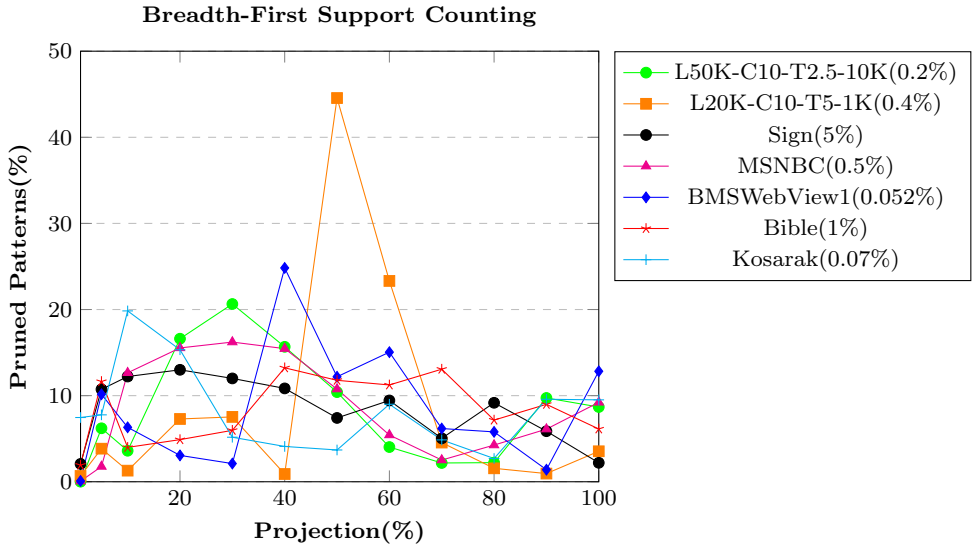


Fig. 17: Evaluation of Breadth-First Based Support Counting Technique

Dataset	Descending *min_sup*					Ascending *min_sup*			
Bible	*min_sup*(%)	10	1	0.8	0.5	0.5	0.8	1	5
	T(sec.)	228.2	579.2	675	935.5	2907	0.0008	0.0007	0.0003
BMSWebView1	*min_sup*(%)	0.06	0.055	0.052	0.05	0.05	0.052	0.055	0.06
	T(sec.)	19.8	71	226.5	865.2	1321	5.7	1.1	0.7
Sign	*min_sup*(%)	30	10	5	1	1	5	10	30
	T(sec.)	19.8	71	226.5	865.2	1	5	10	30
MSNBC	*min_sup*(%)	0.085	0.08	0.075	0.07	0.2	0.3	0.35	0.4
	T(sec.)	1037.4	492.7	829	742.2	3700	0.4	0.3	0.2
Kosarak	*min_sup*(%)	0.085	0.08	0.075	0.07	0.07	0.075	0.08	0.085
	T(sec.)	3037.3	2512	1231.4	1021.3	7925	5.1	3.05	0.98
L20-C10-T5-N1K	*min_sup*(%)	0.04	0.035	0.03	0.025	0.025	0.03	0.035	0.04
	T(sec.)	313.7	221.5	325.3	710	1012	1.14	0.08	0.07
L50-C10-T2.5-N10K	*min_sup*(%)	0.031	0.03	0.025	0.02	0.02	0.025	0.03	0.031
	T(sec.)	313.7	221.5	325.3	710	2010	0.2	0.11	0.05

Table 10: Interactive Mining: Performance in Descending *min_sup* and Ascending *min_sup*.

are super patterns of the existing ones. As, we did not perform the over computations to calculate the information of the semi-frequent patterns which could not help much here, that cost also does not add in our solution. Moreover, our proposed structure stores the complete database in a compact format. So, it is able to handle the absence of prior database in stream mining and runtime threshold parameter change. Also, our solution is able to mine patterns based on user's requests at anytime having no dependency of mining after each iteration. Our proposed tree-based technique is a new approach to solve sequential mining problem. So, our solution can also be fitted to other extra parameter based approaches, e.g., when those approaches would need to re-mine the database, they can use our proposed SP-Tree structures to faster traverse in the database along with generating the patterns.

6 Conclusions

In this study, we have proposed two novel tree-based solutions, Tree-Miner based on SP-Tree and IncTree-Miner based on IncSP-Tree to solve the SPM problem for static and incremental databases respectively. The tree-based structure provides structural advantage which ultimately helps to improve mining performance and handle manipulation over the database. We have also presented a new breadth-first based support counting technique which helps detect the infrequent patterns early and a heuristic pruning strategy to reduce redundant search space. We have also discussed the newly proposed pattern storage structure BPFSP-Tree for the ISPM problem based on IncSP-Tree and its efficient bottom-up pruning strategy. Our proposed solutions are designed based on a single support threshold parameter and able to mine the complete set of

frequent sequential patterns having "build-once-mine-many" property leading to also being suitable for interactive mining.

Moreover, we have discussed the extendability of our approach to other solutions, such as, memory resilient version. We have explored various aspects related to the proposed solutions' implementations which ultimately lights up on the solutions' flexibilities. We have also discussed different challenges related to the incremental mining problem, e.g., usage of sequence summarizer to incrementally update the co-occurrence information of the complete database, concept drift issues, etc. In the performance evaluation section, we have provided analysis for both of our solutions and showed their efficiency in improving mining runtime. As an ongoing and future work, we have planned to extend our solution to solve the SPM problem for data streams, parallel and distributed environments and specialized attribute based databases, such as, weighted and uncertain databases. We also have planned to modify our tree-based solutions to approach the specialized sequential pattern discovery problems, e.g, maximal patterns, closed patterns, top-k patterns, etc. using our novel tree structures' properties and utilities.

Acknowledgements This work is partially funded by ICT Division, Government of People's Republic of Bangladesh.

References

1. J. Ayres, J. Flannick, J. Gehrke, and T. Yiu. Sequential pattern mining using a bitmap representation. In *KDD*, pages 429–435. ACM, 2002.
2. C. Borgelt. An implementation of the fp-growth algorithm. In *Proceedings of the 1st international workshop on open source data mining: frequent pattern mining implementations*, pages 1–5, 2005.
3. L. Chang, D. Yang, T. Wang, and S. Tang. Imcs: Incremental mining of closed sequential patterns. In *Advances in Data and Web Management*, pages 50–61. Springer, 2007.
4. J. Chen. An updown directed acyclic graph approach for sequential pattern mining. *IEEE TKDE*, 22(7):913–928, 2009.
5. Y. Chen, J. Guo, Y. Wang, Y. Xiong, and Y. Zhu. Incremental mining of sequential patterns using prefix tree. In *PAKDD*, pages 433–440. Springer, 2007.
6. H. Cheng, X. Yan, and J. Han. Incspan: incremental mining of sequential patterns in large database. In *KDD*, pages 527–532, 2004.
7. Z. He, S. Zhang, and J. Wu. Significance-based discriminative sequential pattern mining. *Expert Systems with Applications*, 122:54–64, 2019.
8. A. Hosseininasab, W.-J. van Hoeve, and A. A. Cire. Constraint-based sequential pattern mining with decision diagrams. In *Proceedings of the AAAI Conference on Artificial Intelligence*, volume 33, pages 1495–1502, 2019.
9. J. C.-W. Lin, T.-P. Hong, W. Gan, H.-Y. Chen, and S.-T. Li. Incrementally updating the discovered sequential patterns based on pre-large concept. *IDA*, 19(5):1071–1089, 2015.
10. S. Saleti and R. Subramanyam. A mapreduce solution for incremental mining of sequential patterns from big data. *Expert Systems With Applications*, 133:109–125, 2019.
11. J. K. Tarus, Z. Niu, and D. Kalui. A hybrid recommender system for e-learning based on context awareness and sequential pattern mining. *Soft Computing*, 22(8):2449–2461, 2018.
12. P. Fournier-Viger, A. Gomariz, M. Campos, and R. Thomas. Fast vertical mining of sequential patterns using co-occurrence information. In *PAKDD*, pages 40–52. Springer, 2014.
13. P. Fournier-Viger, J. C.-W. Lin, R. U. Kiran, Y. S. Koh, and R. Thomas. A survey of sequential pattern mining. *Data Science and Pattern Recognition*, 1(1):54–77, 2017.
14. W. Gan, J. C.-W. Lin, P. Fournier-Viger, H.-C. Chao, and P. S. Yu. A survey of parallel sequential pattern mining. *TKDD*, 13(3):1–34, 2019.
15. G. Grahne and J. Zhu. Fast algorithms for frequent itemset mining using fp-trees. *IEEE TKDE*, 17(10):1347–1362, 2005.
16. R. Guidotti, G. Rossetti, L. Pappalardo, F. Giannotti, and D. Pedreschi. Personalized market basket prediction with temporal annotated recurring sequences. *IEEE TKDE*, 31(11):2151–2163, 2019.
17. T.-P. Hong, C.-Y. Wang, and Y.-H. Tao. A new incremental data mining algorithm using pre-large itemsets. *Intelligent Data Analysis*, 5(2):111–129, 2001.
18. T.-P. Hong, C.-Y. Wang, and S.-S. Tseng. An incremental mining algorithm for maintaining sequential patterns using pre-large sequences. *Expert Syst Appl*, 38(6):7051–7058, 2011.
19. J.-W. Huang, C.-Y. Tseng, J.-C. Ou, and M.-S. Chen. A general model for sequential pattern mining with a progressive database. *IEEE TKDE*, 20(9):1153–1167, 2008.
20. C. K.-S. Leung, Q. I. Khan, Z. Li, and T. Hoque. Cantree: a canonical-order tree for incremental frequent-pattern mining. *Knowledge and Information Systems*, 11(3):287–311, 2007.
21. M.-Y. Lin and S.-Y. Lee. Incremental update on sequential patterns in large databases by implicit merging and efficient counting. *Information Systems*, 29(5):385–404, 2004.
22. N. P. Lin, W.-H. Hao, H.-J. Chen, E. Hao, and C. Chang. Discover sequential patterns in incremental database. *International Journal of Computers*, 1:196–201, 2007.
23. J. Liu, S. Yan, Y. Wang, and J. Ren. Incremental mining algorithm of sequential patterns based on sequence tree. In *Advances in Intelligent Systems*, pages 61–67. Springer, 2012.
24. B. Mallick, D. Garg, and P. Grover. Incremental mining of sequential patterns: Progress and challenges. *Intelligent Data Analysis*, 17(3):507–530, 2013.
25. F. Masseglia, P. Poncelet, and M. Teisseire. Incremental mining of sequential patterns in large databases. *Data & Knowledge Engineering*, 46(1):97–121, 2003.
26. J. S. Okolica, G. L. Peterson, R. F. Mills, and M. R. Grimaila. Sequence pattern mining with variables. *IEEE TKDE*, 32(1):177–187, 2020.
27. S. Parthasarathy, M. J. Zaki, M. Ogihara, and S. Dwarkadas. Incremental and interactive sequence mining. In *Proc. of eighth CIKM*, pages 251–258, 1999.
28. J. Pei, J. Han, B. Mortazavi-Asl, J. Wang, H. Pinto, Q. Chen, U. Dayal, and M.-C. Hsu. Mining sequential

patterns by pattern-growth: The prefixspan approach. *IEEE TKDE*, 16(11):1424–1440, 2004.
29. D. Perera, J. Kay, I. Koprinska, K. Yacef, and O. R. Zaïane. Clustering and sequential pattern mining of online collaborative learning data. *IEEE TKDE*, 21(6):759–772, 2008.
30. R. A. Rizvee, M. F. Arefin, and C. F. Ahmed. Tree-miner: Mining sequential patterns from sp-tree. In *PAKDD*, pages 44–56. Springer, 2020.
31. E. Salvemini, F. Fumarola, D. Malerba, and J. Han. Fast sequence mining based on sparse id-lists. In *ISMIS*, pages 316–325. Springer, 2011.
32. T. Slimani and A. Lazzez. Sequential mining: patterns and algorithms analysis. *arXiv preprint arXiv:1311.0350*, 2013.
33. R. Srikant and R. Agrawal. Mining sequential patterns: Generalizations and performance improvements. In *International Conference on Extending Database Technology*, pages 1–17. Springer, 1996.
34. K. Wang. Discovering patterns from large and dynamic sequential data. *JIIS*, 9(1):33–56, 1997.
35. Z. Yang, Y. Wang, and M. Kitsuregawa. Lapin: effective sequential pattern mining algorithms by last position induction for dense databases. In *DASFAA*, pages 1020–1023. Springer, 2007.
36. M. J. Zaki. Spade: An efficient algorithm for mining frequent sequences. *Machine learning*, 42(1-2):31–60, 2001.
37. M. Zhang, B. Kao, D. Cheung, and C.-L. Yip. Efficient algorithms for incremental update of frequent sequences. In *PAKDD*, pages 186–197. Springer, 2002.
38. Q. Zheng, K. Xu, S. Ma, and W. Lv. The algorithms of updating sequential patterns. *arXiv preprint cs/0203027*, 2002.