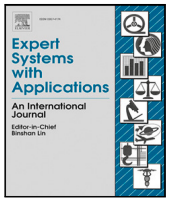
Expert Systems With Applications 242 (2024) 122754

Contents lists available at ScienceDirect

Expert Systems With Applications

journal homepage: www.elsevier.com/locate/eswa



A new tree-based approach to mine sequential patterns

Redwan Ahmed Rizvee [a], Chowdhury Farhan Ahmed [a,*], Md. Fahim Arefin [a], Carson K. Leung [b]

[a] Department of Computer Science and Engineering, University of Dhaka, Bangladesh
[b] Department of Computer Science, University of Manitoba, Canada

ARTICLE INFO

Keywords:
Sequential pattern
Tree-based mining
Incremental mining
Breadth-first based pruning
Pattern storage

ABSTRACT

Generic sequential pattern mining problem aims to mine the set of sequential patterns from a sequential database that satisfies a minimum support or occurrence threshold constraint. The main challenges that affect the efficiency of a solution lie in reducing the pattern search space, early detecting the infrequent patterns, representing the database in an efficient format, etc. Also, additional challenges get included when the problem environment transitions from static to incremental database leading to not to re-mine but efficiently tracking the effect of the incremental portion over the complete updated database. In this article, we introduce a new tree-based solution to the sequential pattern mining problem, including two sets of novel solutions for static and incremental sequential databases. We propose two new structures, *SP-Tree* and *IncSP-Tree*, and design two efficient algorithms, *Tree-Miner* and *IncTree-Miner* to mine the complete set of sequential patterns from static and incremental databases respectively. The proposed novel structures provide an efficient manner to store the complete sequential database maintaining "build-once-mine-many" property and giving scope to perform interactive mining. Additionally, we also design a new breath-first based support counting technique to efficiently identify the infrequent patterns at early stages and a new heuristic pruning strategy to reduce pattern search space. We also design 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 from various real-life and synthetic datasets demonstrate the efficiency of our work in comparison with the related state-of-the-art approaches.

1. Introduction

The idea of the 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 the 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. Table 1 shows an example of a camera market sequential database where each entry denotes a customer's purchase history. Items within the same bracket denote 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 Srikant and Agrawal (1996) 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 numerous applications of SPM, a wide range of literature has addressed the SPM problem and provided solutions that 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 the pattern's gradual extension. Pattern growth approaches are significantly faster compared to the Apriori approach (Borgelt, 2005). The main limitation of Apriori-based approaches is the generation of a huge number of redundant infrequent patterns. PrefixSpan-based approaches (Pei et al., 2004)

The code (and data) in this article has been certified as Reproducible by Code Ocean: (https://codeocean.com/). More information on the Reproducibility Badge Initiative is available at https://www.elsevier.com/physical-sciences-and-engineering/computer-science/journals.

* Corresponding author.
E-mail addresses: rizvee@cse.du.ac.bd (R.A. Rizvee), farhan@du.ac.bd (C.F. Ahmed), fahim@cse.du.ac.bd (M.F. Arefin), Carson.Leung@UManitoba.ca (C.K. Leung).

https://doi.org/10.1016/j.eswa.2023.122754
Received 19 June 2022; Received in revised form 25 November 2023; Accepted 25 November 2023
Available online 28 November 2023

improve this aspect by a gradually shrunk projection of the database during pattern expansion, though the cost is comparatively quite lesser but not insignificant. SPAM-based solutions (Ayres et al., 2002) use bit-based representations to generate the patterns. It is very efficient, but the data structure's size can be heavily affected by the length of the transactions or the database. Some data representation-based literature, e.g., vertical-database (Zaki, 2001) have also been proposed, but the improvement was not quite significant. 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. It has been an ongoing research issue for a long time to improve the data structure to represent the sequential database so that the aforementioned challenges can be efficiently addressed. 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 (Borgelt, 2005; Leung et al., 2007) and incorporate new strategies. But, due to the problem's complexity, the tree alike structures of the itemset mining were not suitable for the SPM problem and a structural solution was yet to be proposed.

Based on this motivation, in this study, we propose a novel tree-based 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 that 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 second proposal of the incremental solution is the result of prior motivation.

Making a tree-based structure, we want to embed ancestor-descendant node relationships in sequential pattern mining where each node represents a particular item of a particular sequence. Our tree tries to share prefixes to make the database representation concise. Also, using our tree node attribute *next_link* we can traverse faster for a particular item in the underlying subtrees, aka in the projected database. The nodes projection pointers or references can be used to represent a pattern's occurrences in the database aka tree in a much compact and intuitive way.

Tree structures help to apply various strategies to improve the solution. In this study, we propose a support counting technique based on a breadth-first based strategy. This strategy mainly implements the idea of gradual level-by-level traversal in the tree using the next link attribute while making an approximation over the maximum possible support for a pattern. While being very flexible and generic this strategy helps to early detect the infrequency of a pattern resulting in early pruning before completing the full projection of a pattern in the tree aka database. During conducting the projections cost of searching gets added. Our second proposal (heuristic support counting strategy) tries to mitigate these costs and re-use the projection information across multiple pattern extensions.

Fig. 1 shows an abstract representation of the core ideas of SP-Tree where each node represents a particular item of a sequence. Patterns are formed using the node concatenations and the nodes work as projection pointers. Due to the prefix sharing property support gets reduced while moving deeper into the tree. The next links shown in dotted lines work as jump connections. So when we need to search a node for a particular item we do not need to scan each branch (aka sequence) character by character sequentially. These properties accumulate while applying a breadth-first based support counting strategy leading to the novelty of our proposal over existing methods.

Generic SPM problem focuses on mining frequent sequential patterns from the static sequential databases (Chen, 2009; Fournier-Viger et al., 2014; Guidotti et al., 2019; Okolica et al., 2020; Pei et al., 2004; Perera et al., 2008; Rizvee et al., 2020; Srikant & Agrawal, 1996; Zaki, 2001). However, in many real-life applications, most of the time, the database is found not to be static; rather gets increased time to time with more information (Mallick et al., 2013). 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 the updated database again from scratch creates performance and resource bottlenecks. From these motivations, the problem of Incremental Sequential Pattern Mining (ISPM) was introduced in Wang (1997) to tackle the challenge of re-mining over the complete database rather than focusing more on the efficient handling of the new incremental database or the increased part of the database (Fournier-Viger et al., 2017; Huang et al., 2008; Mallick et al., 2013; Slimani & Lazzez, 2013).

Table 1
Initial camera market dataset.

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 2
Additional update in database.

sid	New Sequence	Type
1	⟨(ND Filter, Reverse Ring)⟩	Append
2	ϕ	–
3	ϕ	–
4	ϕ	–
5	⟨(camera, kit lens)⟩	Insert

Table 2 shows an example of an incremental version of our prior static database presented in Table 1. Here, two sequences newly appeared. The first one is an appended sequence to the existing sequence with an 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 ($Insert$), the 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 (Cheng et al., 2004; Lin et al., 2015), 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 (Cheng et al., 2004; Lin et al., 2015; Lin & Lee, 2004; Liu et al., 2012; Saleti & Subramanyam, 2019) have addressed this problem.

Because of various crucial factors in 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 (Zheng et al., 2002), semi-buffer (Cheng et al., 2004), pre-large with upper and lower thresholds (Lin et al., 2015), etc. The main problem with 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

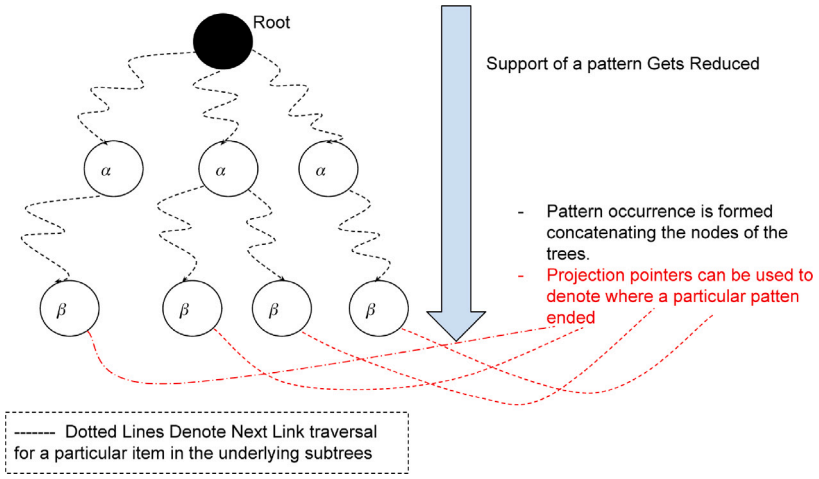


Fig. 1. An abstract representation of the core novelties of SP-Tree.

severely affected due to concept drift and create resource misuse and bottleneck. Another major concern comes from how to store the mined patterns so that during the database update the group of redundant patterns can be efficiently removed and the updated patterns can be easily maintained. A wide range of literature also focused on this and designed various tree-based solutions (Chen et al., 2007; Lin et al., 2015; Liu et al., 2012). An important issue also comes from supporting data structures (Fournier-Viger et al., 2014, 2017; Saleti & Subramanyam, 2019) that might be useful to store various information which guides to reduce search space. These structures face a challenge in an incremental environment. Because, when the size of the database has gotten updated its impact also needs to be propagated in these supporting structures while not imposing a grand computational cost.

Based on these aforementioned challenges and issues, in this study, we propose 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. To control the dependency over the threshold parameters and to reduce the issues of concept drift, our proposed mining algorithm IncTree-Miner mines the patterns based on a single support threshold parameter.

The usage of co-occurrence information can significantly reduce the search space which has been discussed in many literature (Fournier-Viger et al., 2014, 2017; Saleti & Subramanyam, 2019). Our solutions have also adopted this concept. This information stored in *CETable* adopted from Fournier-Viger et al. (2014) states the relationship among the two length items which guides during pattern extension. But, it becomes a challenge how to efficiently update this information for a gradually increasing database. To address this issue, we propose a novel structure *Sequence Summarizer* that helps calculate such information efficiently, especially in an incremental environment rather than complete re-calculation over the whole database or tree. This supporting structure based on our algorithm observes the modified portion of each sequence and based on that merges its impact with the existing items of that particular sequence in CETable. They also help to perform the *Append* operation over the existing sequences by directly reaching the desired node from which the update will incur. CETable (Rizvee et al., 2020) stores the support of all two-length ordered patterns found in the database so that it can tackle the dynamic distribution change of the items during database increment or modification.

In this study, to maintain the frequent mined patterns across different mining iterations, we design a new Bi-directional Projection Pointer Based Frequent Sequential Pattern Tree (*BPFSP-Tree*). This structure keeps the frequent sequential patterns, their support, and projection pointers using the node pointers 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 in a bottom-up manner.

Our proposed approach is a new take on the generic SPM problem. So, it might be 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 a prior database in-stream mining and runtime threshold parameter change. In summary, our main contributions are as follows:

1. We propose 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-based structure *IncSP-Tree* to solve the SPM problem for static and incremental databases, respectively.
2. Based on the tree properties, we designed two new pruning strategies, an efficient breadth-first based support counting technique and a heuristic pruning strategy to early detect the infrequency of a pattern during projection in the trees.
3. We also propose a supporting structure *Sequence Summarizer* to efficiently calculate and update the co-occurrence information and perform *Append* operation during database modification across different mining iterations.
4. We propose a new pattern storage structure *BPFSP-Tree* to store the frequent sequences along with projection pointers to reduce the number of tree scans and efficiently remove the infrequent patterns in a bottom-up manner.
5. We also present a Discussion regarding the efficiency and effectiveness of our proposed solutions and issues related to implementation.

The current article adds substantially new, extended, and different contributions beyond our PAKDD 2020 conference version (Rizvee et al., 2020), which include an incremental data structure and a novel incremental mining algorithm over prior proposed static mining and static tree-based structure. Here, we have also optimized and improved the nature of the prior static tree-based structure. Additional pattern pruning strategies, supporting new data structures to tackle the challenges of a dynamic database environment are also proposed. This study also includes a more comprehensive discussion of the solutions with enhanced motivation and extensive performance analysis in various real-life and synthetic datasets. The applicability of the proposals in various applications with many issues related to the implementation is also widely addressed.

The rest of the current article 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 conclude 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. First, we shall summarize the literature relevant to the static SPM problem and then we shall discuss the studies related to the incremental SPM problem in brief.

2.1. Static sequential pattern mining problem

Being a very important problem domain, a good amount of literature has addressed numerous issues related to SPM (Fournier-Viger et al., 2017; Gan et al., 2019) and provided a wide range of solutions, being first introduced in Srikant and Agrawal (1996). 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 the 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 (Fournier-Viger et al., 2014; Pei et al., 2004; Rizvee et al., 2020; Zaki, 2001). PrefixSpan (Pei et al., 2004) is one of the most popular and efficient techniques to solve the SPM problem which proposed the idea of database projection. SPAM (Ayres et al., 2002) used bit-based representation to calculate patterns' support incorporating mutual dependence-based search space pruning technique. FAST (Salvemini et al., 2011) improved the support counting technique of SPADE (Zaki, 2001) using sparse id-lists. LAPIN (Yang et al., 2007) showed the importance of the last event's items during pattern extension. CM-SPADE and CM-SPAM (Fournier-Viger et al., 2014) 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 has been briefly discussed. Besides these, there lie a set of literature that have addressed different variations and applications of SPM problems. He et al. (2019) proposed a new type of SPM problem named 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 corrections in this regard to reduce the number of false positive patterns. Tarus et al. (2018) proposed a hybrid recommendation system for e-learning purposes. In their proposal, they had to discover context-aware information in such regard that they used SPM techniques. In the aforementioned articles, both used GSP to generate the sequential patterns and over that, they added their contribution. Hosseininasab et al. (2019) 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 constraints. 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.

Gan et al. (2019) gave a summary to present the progress in parallel sequential pattern mining. Wu, Luo, et al. (2021) proposed the idea of non-overlapping sequential pattern mining based on three types of interest value over the items embedding the gap constraint. Wu, Wang, et al. (2021) addressed the problem of top-k self-adaptive contrast sequential pattern mining. This particular problem focuses on sequence classification mainly targeting contrasting patterns from the positive and negative sequences. Dong et al. (2018) proposed a solution to find repeating negative sequential patterns. Song et al. (2021) approached the maximal high average utility itemset mining problem. This study focuses more on itemset mining rather than sequential pattern mining. Wu, Hu, et al. (2022) proposed an algorithm to embed time constraints to preserve the ordering among the items during pattern mining. Wu, Lei, et al. (2021) merged the concept of both utility and one-off sequential pattern mining. One-off sequential pattern mining is another variation of SPM that addresses repetitive patterns with gap constraints. Li et al. (2022) proposed an algorithm to mine non-overlapping maximal sequence pattern mining. Huang et al. (2022) addressed the problem of query-focused pattern mining named as targeted SPM problem. Lin et al. (2020) proposed a chain-based structural solution for addressing the high utility pattern mining. Wu, Yuan, et al. (2022) designed a solution that combined a non-overlapping problem with weak gap constraints in sequential pattern mining.

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.

2.2. Incremental sequential pattern mining problem

The problem of mining sequential patterns in the incremental database has also attracted researchers because of its wide variety of challenges and applications (Mallick et al., 2013; Slimani & Lazzez, 2013). Different literature raised different factors and provided solutions based on them. To solve the ISPM problem, one of the earliest solutions was given by Wang (1997), who 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 on the database size and the sensitivity of the position where the update occurs. In ISM, Parthasarathy et al. (1999) adopted a vertical mining approach along with maintaining 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 algorithm (Masseglia et al., 2003) 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 Zhang et al. (2002) to provide a solution for the incremental database. The goal of the MFS algorithms is different from ours because they targeted to mine maximal frequent sequences.

Lin and Lee (2004) proposed the IncSP algorithm and provided two completely novel ideas for efficient counting and implicit merging and were found to be very efficient but had the problem of level-wise candidate generation and test paradigm because of adopting the Apriori approach. Cheng et al. (2004) designed IncSpan as one of the most popular approaches for solving the ISPM problem. It introduced the idea of a 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 additional 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 on the parameters. This approach also induced

some additional pre-computation. In PBIncSpan (Chen et al., 2007), 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 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 in projecting the database. Liu et al. (2012) proposed a modified version of PBIncSPan called ISPBS. 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 a tree is not practical and the result is far from optimal in databases of medium to large size.

Being motivated by the FP-Tree (Grahne & Zhu, 2005) 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 Hong et al. (2001, 2011) and Lin et al. (2015). This concept added two new parameters, named 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 buffers 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 on 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 compactly holds all the information, it is not affected due to the information's absence in stream databases. Researchers also introduced some hybrid and environment-based solutions. For example, Lin et al. (2007) provided an Apriori and maximal pattern-based approach, Chang et al. (2007) gave a solution to mine closed patterns from the incremental database, Saleti and Subramanyam (2019) 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 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.

Apart from the core and relevant literature to the addressed problem of ISPM, some recent studies can also be highlighted. These studies have tried to address different variations of incremental sequential pattern mining problems. Roy et al. (2021) proposed a solution to the SPM problem for uncertain databases. They also extended their solution for an incremental environment. Their solution was based on hierarchical indexing. Ishita et al. (2022) proposed a method to mine regular high-utility sequential pattern mining for incremental databases. Andrzejewski and Boinski (2019) developed an algorithm to mine co-location or spatial patterns for incremental databases using the computation power of parallel processing. Nam et al. (2020) designed an algorithm to combine weighted constraint over erasable sequential pattern mining. To solve the problem they used list-based structuring.

Table 3
Static database.

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

Table 4
Incremental database.

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

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 $I = \{i_1, i_2, i_3, \ldots, i_n\}$. An event or itemset e is a set of items such that $e \subseteq I$. A sequence $s = \langle e_1, e_2, e_3, \ldots, e_n\rangle$ 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 3.1 (*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} 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 Definition 3.2.

Definition 3.2 (*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 $S_P \geq min\_sup \times |D'|$.

Table 3 shows a static database, which will be used as an example for the remaining discussion of static mining. Table 4 shows an incremental database with two iterations, 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. For example, in $\langle(abc)(ac)\rangle$, 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. For example, in sequence $\langle(abc)(ac)\rangle$, 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. For example, in sequence $\langle(ab)(abc)\rangle$, 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 1th and 2nd 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.

In Section 4.1.4, we will provide an in-depth discussion on how the nodes of SP-Tree and IncSP-Tree represent the sequential databases shown in Tables 3 and 4 respectively with their corresponding attributes' values.

4.1.2. Sequence summarizer (seq_sum)

Co-occurrence information (Fournier-Viger et al., 2014, 2017) 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. 5(b). It illustrates key 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. Fig. 2 illustrates the idea of updating $CETable_s$

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

Table 5
Definition of sequence summarizer.

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 $\alpha \in s$ $\alpha : (bitset_\alpha^{previous}, bitset_\alpha^{current})$

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, bold capital *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.

Fig. 2(a) shows how we update the $CETable_s$ column for the newly observed occurrence of $\langle(\alpha)(\beta)\rangle$ that has not been found prior for a particular sequence. When $S_\alpha^{st} < S_\beta^{en_c urrent}$ occurs in a sequence for two items α and β, we observed an occurrence of $<(\alpha)(\beta)$ in that sequence. Now, we mainly need to test two cases,

1. α has occurred for the first time in the second iteration ($S_\alpha^{st} \geq S_B^{en\_previous}$): For example, let us consider in the first iteration we had a sequence $\langle(b)(c)\rangle$. Then, in the second iteration, we append another sequence $\langle(a)(b)\rangle$ at the end of it. So here $\alpha = a$ has occurred for the first time in the second pass. Here, after update, a's value will be $[2, -1, 2]$ and $\beta = b$'s value will be $[0, 0, 3]$. The order denotes S^{st}, $S^{en\_previous}$ and $S^{en\_current}$ respectively for each. 0 based indexing is followed.
2. β has occurred for the first time in the second iteration ($S_\alpha^{st} < S_B^{en\_previous}$ and $S_B^{st} >$ previous max event no): For example, let us consider in the first iteration we had a sequence $\langle(a)(c)\rangle$. Then in the second iteration, we append another sequence $\langle(b)(a)\rangle$ at the end of it. So $\beta = b$ has occurred for the first time in the second pass. Here, after update, $\alpha = a$'s value will be $[0, 0, 3]$, b's value will be $[2, -1, 2]$ and previous max event no was 1. The order denotes S^{st}, $S^{en\_previous}$ and $S^{en\_current}$ respectively for each. 0 based indexing is followed.

If any of these satisfies we increase by 1 for the value $CETable_s[\alpha][\beta]$.

Similarly, Fig. 2(b) shows how we calculate $CETable_i$ column to hold the number of $\langle(\alpha\beta)\rangle$ occurrences. For each item α we store the items β that have occurred with it in the same events where $\beta <_{order} \alpha$ in bitset format. Then, performing bitwise *XOR* operation between $bitset_\alpha^{previous}$ and $bitset_\alpha^{current}$ we extract the occurrences of β with α in the same event in the last iteration for a particular sequence. Then, performing the bitwise *AND* operation between the earlier resultant value and $bitwise_\alpha^{current}$ we extract the new pairs $(\beta\alpha)$ that have not been observed prior for this sequence. Traversing over the bitset of the final resultant value we get the βs to increase by 1 of $CETable_i[\beta][\alpha]$. For example, let us consider a sequence in the first iteration we had, $\langle(ac)\rangle$. Then, in the second iteration, we append $\langle(bc)\rangle$ at the end. So, here $bitset_c^{previous}$ and $bitset_c^{current}$ will be $(1)_2$ and $(11)_2$ respectively. Then, $unique_{bits}$ will be $(10)_2$ and $same\_event$ will be $(10)_2$. So, we just need to increment the value for $CETable_i[b][c]$-second set bit denotes item b.

Fig. 5(c) also shows some other examples for $sid = 1$. As always, only the final update is kept in the table.

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 Algorithm 1. In *Append* operation, a prefix of the corresponding sequence (s_{sid}) already exists and ev_p is used to denote the previous maximum event no. In *Insert* operation, it is considered as -1. In line 9, the actual item (I) and event no (ev') is calculated.

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** Return N ▷ All events been inserted
7: **if** $(I_{no} > \text{len}(s[ev]))$ **then** ▷ Add next event
8: Return InsIncSPTree(N,$ev+1$,0,0) ▷ 0 Based Indexing
9: $I \leftarrow s[ev][I_{no}]$, $ev' = ev + ev_p$
10: **if** $(N.ch[\{I, ev'\}] = \{\})$ **then** ▷ No child found
11: ▷ create a child node n and initialize the attributes
12: $n.I \leftarrow I$, $n.ev \leftarrow ev'$, $n.cr \leftarrow pa$, $n.C \leftarrow 0$, $n.C' \leftarrow 0$, $n.p_{inf} \leftarrow bit$
13: $n.md \leftarrow 0$ ▷ md will be set as pa in Algorithm 2
14: $N.ch[\{I, ev'\}] \leftarrow n$ ▷ Setting n as child node
15: $n \leftarrow N.ch[\{I, ev'\}]$
16: **if** $seq\_sum[s_{sid}][I] = \{\}$ **then** ▷ Positional Info update
17: $seq\_sum[s_{sid}][I]_{st} \leftarrow ev'$, $seq\_sum[s_{sid}][I]_{en\_previous} \leftarrow ev'$
18: $seq\_sum[s_{sid}][I]_{en\_current} \leftarrow ev'$
19: **else** $seq\_sum[s_{sid}][I]_{en\_current} \leftarrow ev'$ ▷ Ending position update
20: $bit' \leftarrow seq\_sum[s_{sid}][I]_{bitset_{current}}$ **OR** bit ▷ set bit/items are in same event
21: $seq\_sum[s_{sid}][I]_{bitset_{current}} \leftarrow bit'$ ▷ Same Event Info update)
22: **Return** InsIncSPTree(n,ev',$I_{no}+1$, bit' **OR** bit) ▷ bit' **OR** bit: Setting bit for I for next call

Algorithm 2 Updating attributes of IncSP-Tree

1: **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 $L_{S^{MAL}}$.
2: **Output:** Update of the concerned branch's attributes.
3: **procedure** Update Path(N)
4: **if** (N is **None**) **then** Return ▷ All the node's attributes are updated
5: $md' \leftarrow N.md$
6: **if** $N.md < pa$ **then** $N.md \leftarrow pa$ ▷ newly modified node
7: $N.C' \leftarrow N.C$, $N.nl_m \leftarrow \{\}$ ▷ Support Tracking, Runtime memory clear
8: **for** item $i \in L_{nl_m}$ **do**
9: $N.nl_m[i] \leftarrow N.nl_m[i] \cup L_{nl_m}[i]$ ▷ Tracking underlying modified nodes
10: **for** item $i \in L_{S^{MAL}}$ **do**
11: $N.S^{MAL}[i] \leftarrow N.S^{MAL}[i] + 1$ ▷ to reduce loop
12: **if** $md' \neq N.md$ **then** ▷ N was not tracked
13: $L_{nl_m}[N.I] \leftarrow N$ ▷ Need to track N from ancestors for I
14: **else** Delete $L_{nl_m}[N.I]$ ▷ Already Been Tracked
15: **if** $t = Insert$ **then**
16: $N.c \leftarrow N.c + 1, L_{S^{MAL}} \leftarrow L_{S^{MAL}} \cup \{N.I\}$
17: **else** ▷ Append Operation
18: **if** N is newly considered node for s **then**
19: $N.c \leftarrow N.c + 1, L_{S^{MAL}} \leftarrow L_{S^{MAL}} \cup \{N.I\}$
20: **else** $L_{S^{MAL}} \leftarrow L_{S^{MAL}} - \{N.I\}$ ▷ Old nodes, Already Tracked
21: **for** item $i \in L_{nl}$ **do** $N.nl[i] \leftarrow N.nl[i] \cup L_{nl}[i]$ ▷ Underlying nl update
22: **if** $N.cr = pa$ **then** ▷ Tracking new nodes for next link
23: **if** $md' = N.md$ **then** Delete $L_{nl}[n.I]$ ▷ Already Tracked for next link
24: **else** $L_{nl}[N.I] \leftarrow N$ ▷ Tracking node for nl
25: **else** Delete$L_{nl}[N.I]$ ▷ Already Tracked by ancestors
26: UpdatePath($N.parent$)

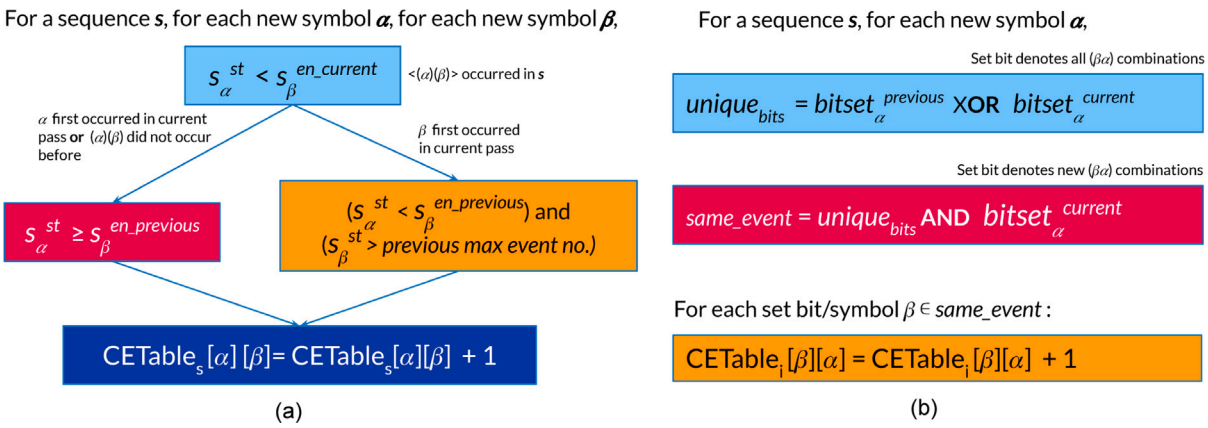


Fig. 2. (a) $CETable_s$ calculation, (b) $CETable_i$ calculation.

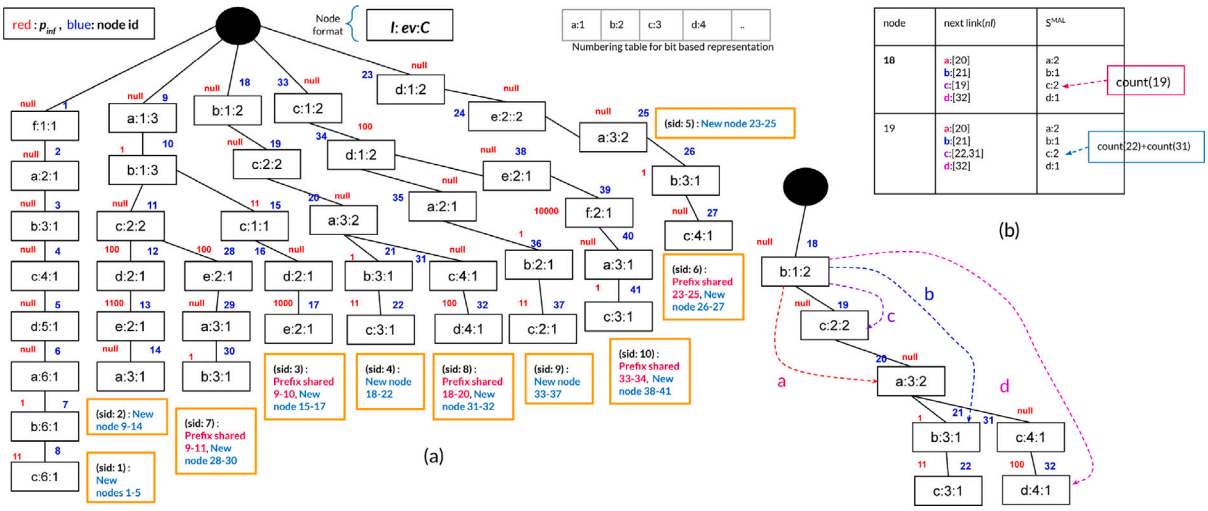


Fig. 3. (a) Complete SP-Tree, (b) Next links for nodes 18 and 19.

When a desired child node is not found it is created (lines 12–14) and set (line 15). *seq_sum* is updated in lines 16-21. Lines 16–19 update the information that is required to update $CETable_s$ column and lines 20–21 update the information that is required to update $CETable_i$ column. Bit-based operations are performed to update the Same Event Info of *seq_sum* in lines 20-21. Finally, in a recursive manner, the remaining items are inserted in the tree in line 22.

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, IncSP-Tree needs to run it after each sequence insertion, whereas SP-Tree can run only once after the complete insertion of the database. Using a bottom-up recursive traversal with the usage of the last node reference of *seq_sum* for each sequence s the nodes' attributes are calculated (line 26). For each N, its nl, nl_m and S^{MAL} attributes are updated using the information calculated from the underlying nodes in lines 21, (8–9) and (10–11) respectively. Also, for N the corresponding global lists L_{nl}, L_{nl_m} and $L_{S^{MAL}}$ are updated to send its information to its ancestor nodes in lines (22–25), (12–14), (18–20) respectively. In *Insert* operation, all the nodes' concerning sequence s count attributes are incremented (lines 18–19) whereas in *Append* operation only the newly considered nodes' for sequence s count attributes get incremented (lines 17–20). Algorithm 2 assumes 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

Fig. 3(a) shows the complete SP-Tree 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. Fig. 3 gives a complete commentary for understanding how each sequence is inserted, new nodes are created or existing nodes get prefixes shared, etc. For example, 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 section 4.1.1. To understand the usage of node attributes properly, we take a node. For example, let us consider node number 10. Here, as I, ev, and C attributes b, 1, and 3 are stored representing the item, the corresponding event no, and the number of times this node was overlapped or count attribute respectively. As P_{inf} (written in red color) we have stored 1 setting the first bit for a for node number 9 as both have the same event no. Fig. 3(b) also shows the next links for nodes 18 and 19. We have shown from each node for a particular item where we can jump into the corresponding underlying subtrees. Like here from node 19, using the next links for a, we can directly jump to node 20. Similarly from the same node using the next links for c we can jump to nodes 22 and 31 lying in different disjoint subtrees.

Fig. 4 shows the proposed IncSP-Tree for the exampled incremental database for the two iterations. Similar to static mining, sequences are

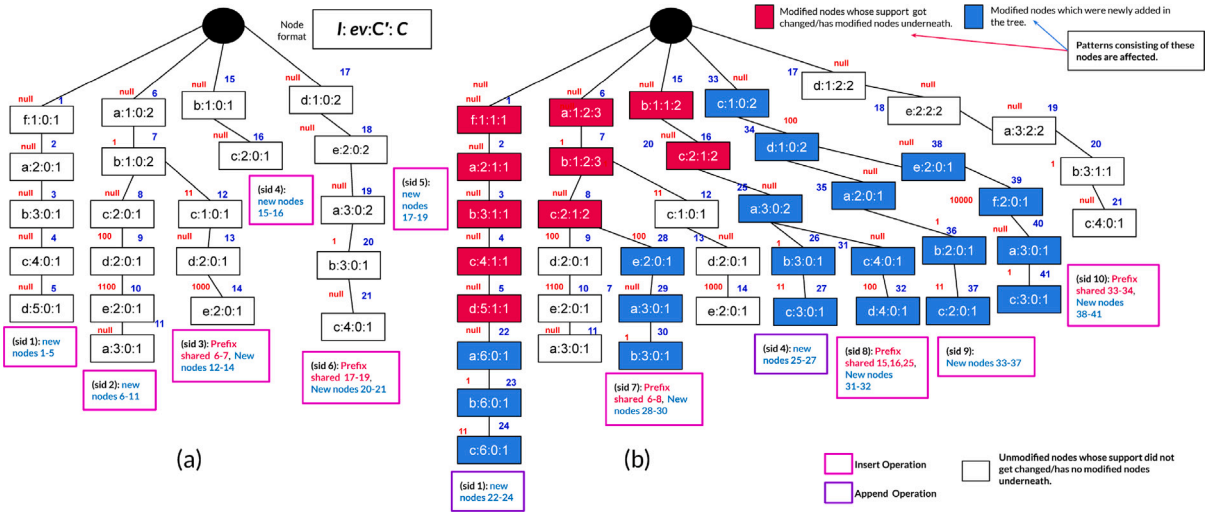


Fig. 4. (a) IncSP-Tree after iteration 1, (b) IncSP-Tree after iteration 2.

inserted according to the increasing order of sids 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 an update wherein $Insert$ operation every node's count increases. But, in the $Append$ 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 $sid = 1$. If we observe a particular node, we shall be able to understand the node attributes and their changes properly. For example, let us consider node number 7. After the first iteration, we shall have the values of I, ev, C' and C as b, 1, 0 and 2 respectively. After the second iteration, the count attributes of this node update to 2 and 3 for C' and C respectively. Because due to some new items' insertion, the number of times this node overlapped has increased. The present count and previous count denote the value of overlapping of this node up to the present and previous iterations respectively. Similarly, we have a modified next link attribute here to traverse only in the underlying modified subtrees.

Fig. 8(b) shows the next links (nl) and modified next links (nl_m) for nodes 6 and 7 where $nl_m \subseteq nl$. We observe that the complete set of modified next links is a subset of corresponding next links. Like, here from node 6 using the next links for a we can reach nodes {11, 29} whereas using the modified next links we can reach node 29. 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 two types of modified nodes, those nodes that are newly created (filled in blue color) and those old nodes (filled in red color) that have modified nodes underneath. Uncolored nodes are unmodified nodes. Patterns consisting of the modified nodes are the affected patterns due to database increment. Fig. 5(c) shows 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 4.1 (*Pattern Formation*). Based on Fig. 5(f), pattern formation concepts can be pointed out 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.
  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 have completed searching up to that node. During the following iterations, we will search only in their underlying subtrees.

Fig. 6 shows 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. For example, if we observe the pattern $\langle(ab)\rangle$, this pattern is then formed concatenating nodes $(6,7), (9,10), (20,21), (35,36), (25,26)$. Each concatenation represents an occurrence in a subtree and all lie in different disjoint subtrees. So, to represent the pattern's occurrences or projection nodes $N_{\langle(ab)\rangle}$ we can only consider those nodes where they have ended, meaning $N_{\langle(ab)\rangle} = \{7, 10, 21, 36, 26\}$. Summing these nodes count attribute's value, we get the support of $\langle(ab)\rangle$. While extending $\langle(ab)\rangle$, we have completed projecting up to these nodes and we shall only search in the underlying subtrees.

Suppose, we have a pattern $P = \langle(\alpha\beta)\rangle$ ending at nodes $N_P = \{V_i, V_j, V_k, \ldots, V_n\}$. We want to find nodes $N_{P\gamma}$ which will extend P for γ. For each $V \in N_P$, we follow a similar extension procedure (SE/IE) each having two cases. Based on Fig. 5(d), we discuss the extensions as follows:

Definition 4.2 (*SE in SP-Tree, $P \rightarrow P\{\gamma\}$*). Two cases are:

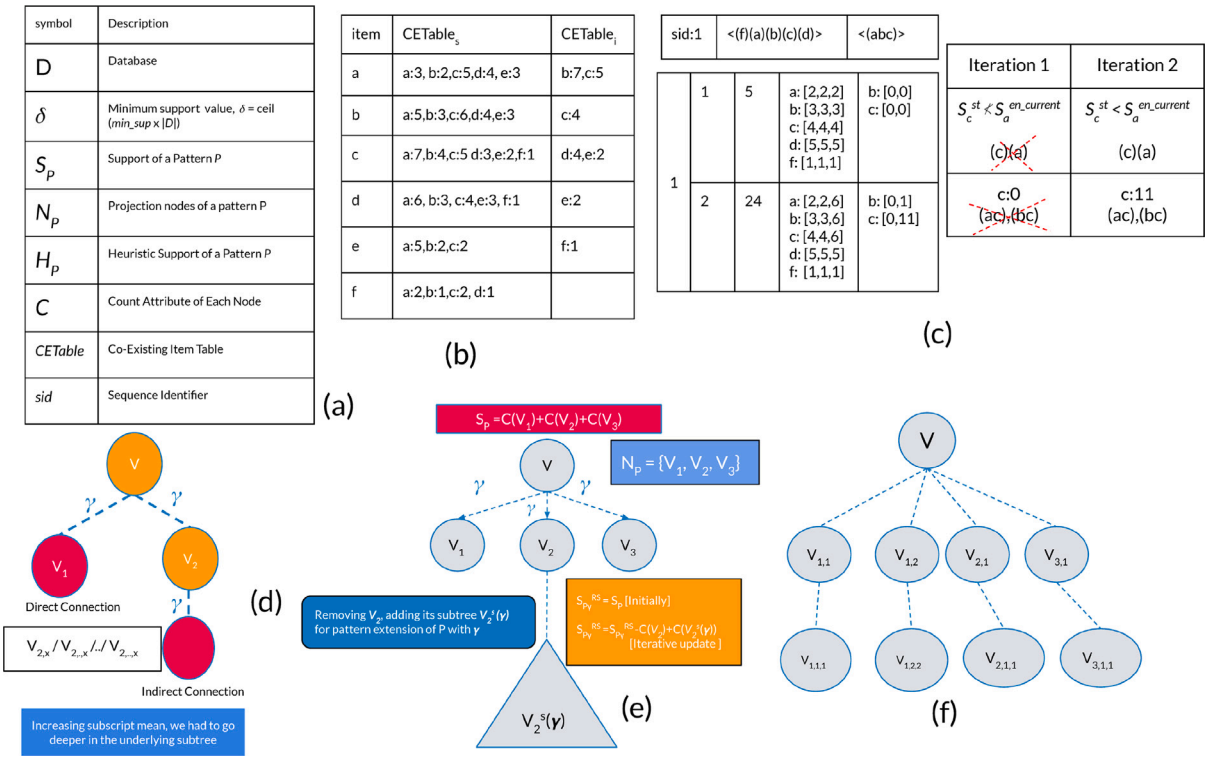


Fig. 5. (a) Terminologies for static mining (b) Co-existing item table, (c) Example of sequence summarizer structure for $sid = 1$, (d) Pattern generation example, (e) Breadth-first based support counting technique, (f) Pattern formation concepts.

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. For example, based on Fig. 6), pattern $\langle(a)\rangle$'s one projection node is 2. Now to construct $\langle(a)(b)\rangle$ from this node we perform a next link traversal for b and reach node 3. As their event no is not same, 3 performs a SE over node 2 for b.
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 cannot 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. For example, based on Fig. 6, $\langle(a)\rangle$'s one projection node can be found in node 9. Now to conduct a SE for b from this node we perform a next link traversal and reach node 10. As nodes 9 and 10 both have the same event no, 10 cannot perform SE for b over node 9. So, we perform another next link traversal for b from node 10 to go deeper in the subtrees and reach node 30 which supports the constraint of different event no, and perform the extension $\langle(a)\rangle \rightarrow \langle(a)(b)\rangle$.

Definition 4.3 (*IE in SP-Tree, $P \rightarrow \{P\gamma\}$*)**.** Two cases are:

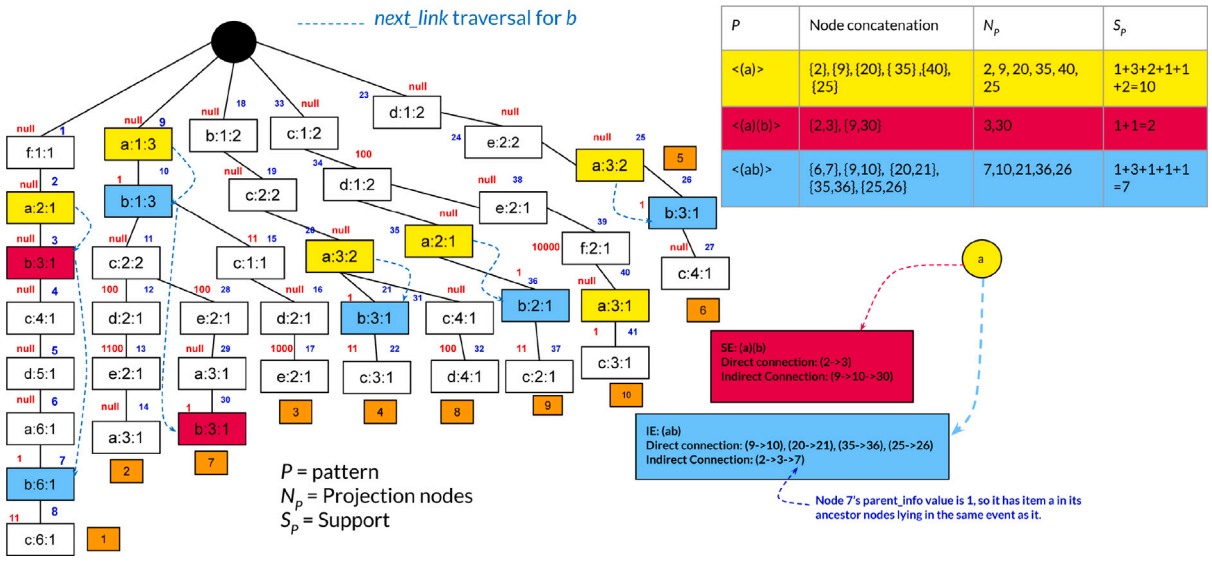
1. *Direct Connection*: For a node $V \in N_P$ suppose through next link for γ, we reach node V_1. If $V.ev = V_1.ev$, then V_1 can perform IE over V. For example, from Fig. 6, we oberve that pattern $\langle(a)\rangle$'s one projection node can be found in node 9. To conduct IE for b to construct $\langle(ab)\rangle$ from $\langle(a)\rangle$ we perform a next link traversal for b and reach node 10 that satisfies the event no matching constraint with node 9. So, 10 performs an IE over node 9 for b.
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 cannot perform IE over V. Then, we need to search in the 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 that are in the same event as it and can be calculated through bitwise AND operation. For example, in Fig. 6, we consider the projection node 2 which is an occurrence of $\langle(a)\rangle$. Now, to conduct IE for b from this node, we perform the next link traversal and reach node 3. As nodes 2 and 3 do not have the same event no, 3 cannot perform IE over 2. So, we go deeper using the next links and reach node 7. Node 7 has an ancestor node 6 that has the same event no as it and has item a. This is tested by observing the parent info of node 7 where the first bit is set for the presence of item a in one of the ancestor nodes (node 6) that have a similar event no as it. Finally, node 7 performs IE for b over node 6 and forms an occurrence of $\langle(ab)\rangle$.

To search in a node's underlying subtree we take all the nodes reached through the next link and simulate each connection case. Fig. 3(b) shows a small visualization of traversing through the next links. Our node extension procedure works in level-by-level manner through our proposed breadth-first based support counting mechanism.

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 $P = \langle(\alpha\beta)\rangle$ with support S_P ending at nodes $N_P = \{n_1, n_2, n_3, \ldots, n_k\}$ 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. 5(a).

Lemma 4.1 (*Support Downward Closure*)**.** *From a node $n_i \in N_P$, suppose we reach a set of nodes $M = \{m_{1,i}, m_{2,i}, \ldots, m_{x,i}\}$ through next link for any item ϵ, where each $m \in M$ are in disjoint subtrees and $m_{j,i}$ denotes the jth branch in n_i's subtree. Then, $C(n_i) \geq \sum_{j=1}^{x} C(m_{j,i})$.*



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. 6. Pattern extension examples.

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

Now, we 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 4.2 (*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 Fournier-Viger et al. (2014). □

For example, suppose, $\delta = 3$, now pattern $\langle(a)(c)(b)\rangle$ cannot be frequent, because $\langle(a)(b)\rangle$ is not frequent (in $CETable_s$ column of Fig. 5(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 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 Lemma 4.1. 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 4.3 using Fig. 5(e).

Lemma 4.3 (*Breadth-First Based Counting Strategy*). *Suppose, from a pattern P we have to check extension 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 4.1], so $S^{RS}_{P\gamma} \leq S_P$. □

Algorithm 3 Breadth-First Based Support Counting

```
1: Globals: δ = min_sup × |D|
2: procedure BreadthFirstSupportCounting(P, N_P, γ, S_P)
3:     N_Pγ ← {}, S^RS_Pγ ← S_P, Q ← N_P                ▷ Extended Nodes, Actual Support, Queue
4:     for each node n_i ∈ Q do                          ▷ Extending P for γ(Pγ) : (αβ)(γ) or (αβγ)
5:         if (S^RS_Pγ − C(n_i) + n_i.S^MAL[γ] < δ) then Return Infrequent
6:         Q ← Q − {n_i} , S^RS_Pγ ← S^RS_Pγ − C(n_i)    ▷ Removing Node, Will check in subtree
7:         for each node m_j ∈ n_i.nl[γ] do
8:             S^RS_Pγ ← S^RS_Pγ + C(m_j)                ▷ Maximum possible support can be found in S^RS_Pγ
9:             if (pattern extension constraint satisfied) then
10:                N_Pγ ← N_Pγ ∪ m_j
11:            else Q ← Q ∪ {m_j}                        ▷ Need to check further in underlying subtrees
12:        if (S^RS_Pγ < δ) then Return Infrequent
13:    Return N_Pγ, S^RS_Pγ                              ▷ Frequent: Return extension nodes with actual support
```

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. 7, which shows how we calculate the projection nodes for the pattern $\langle(a)(b)\rangle$. We discover the nodes in level-by-level manner and may stop projecting at any time based on the maximum possible support. We start with the projection nodes, {2, 9, 20, 35, 40, 25}, putting them in a queue Q and setting the maximum possible runtime support as $S^{RS}_{(a)(b)} = S_{(a)} = 10$. Then, we iterate over the queue's nodes gradually. For each node, we perform a next link traversal for b to find the appropriate node in the underlying subtrees to perform the extension and update $S^{RS}_{(a)(b)}$. Here, from nodes 2, 9, 20, 35, 40, and 25, we reach nodes 3, 10, 21, 36, None, and 26 respectively. At each step, we check the current updated $S^{RS}_{(a)(b)}$ against δ. If at any step we see that $S^{RS}_{(a)(b)} < \delta$, we stop projecting because the extended pattern will not be frequent. The nodes which have found their suitable appropriate nodes are removed from Q, here such node is $2 \rightarrow 3$. For the remaining nodes, we keep them in Q and continue searching in their subtrees, here such nodes are 10, 21, 36, 26. The searching stops when we cannot go much deeper or the maximum possible support has fallen below δ.

4.3.3. Heuristic iList pruning

This is our second proposed pruning strategy and stated in Lemma 4.4.

Lemma 4.4 (*Heuristic iList Pruning*). *Suppose we have an item $\epsilon(\epsilon \in sList \cap iList)$. During support calculation as SE for ϵ, we collectively*

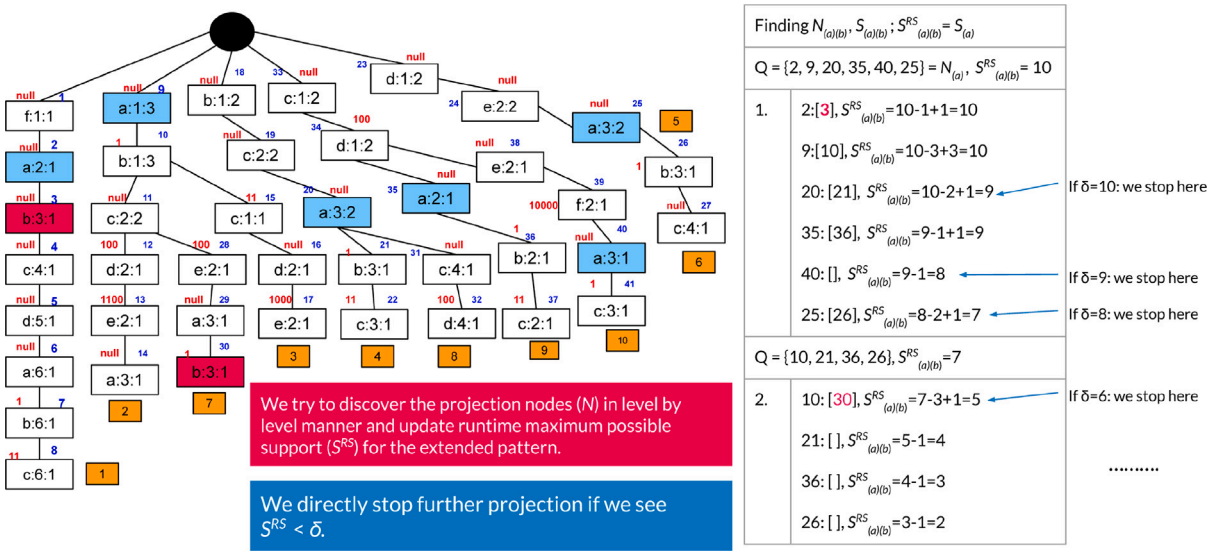


Fig. 7. Simulation of breadth-first based support counting technique.

reach nodes $M = \{m_{1,1}, m_{2,1}, \ldots, m_{1,2}, m_{2,2}, \ldots, m_{1,x}, \ldots, m_{2,y}, \ldots, m_{k,z}\}$ *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 *cannot perform IE over* P *and can be removed from* $iList$.

Proof. Intuition lies behind Lemma 4.1. 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. 3. Here, $N_{(a)} = \{2, 9, 20, 35, 40, 25\}$. For b, $\langle(a)\rangle$ 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 4.5, which is a very widely addressed pruning strategy which has been adopted in our solution and it comes from the apriori property that if $\langle(\alpha)(\gamma)\rangle$ is not frequent, then $\langle(\alpha)(\beta)(\gamma)\rangle$ will never be frequent.

Lemma 4.5 (*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 4.6 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 is no need to perform more projection.

Lemma 4.6 (*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 the 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 (Chen et al., 2007; Liu et al., 2012). 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 is 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 some of its 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, a bottom-up strategy is followed which leads to traversing lesser nodes. To traverse from one leaf node to another in BPFSP-Tree, a link is maintained named *end_link*.

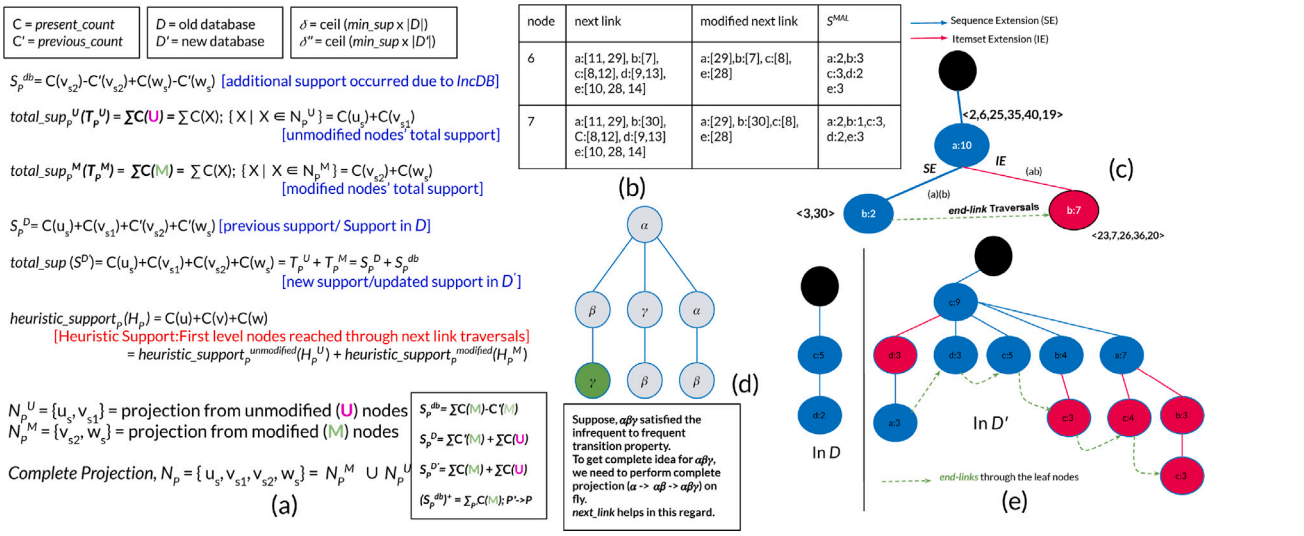


Fig. 8. (a) Pattern generation approach for IncTree-Miner, (b) Example of next links and modified next links, (c) Example of BPFSP-Tree, (d) Memory resilient IncTree-Miner, (e) BPFSP-Tree for patterns with prefix $\langle c \rangle$ after iterations 1 and 2 for $\delta = 2, \delta' = 3$.

A small example of BPFSP-Tree is shown in Fig. 8(c) with *end_links* for traversing through the leaf nodes using SP-Tree node references of Fig. 4(b). Here, we have shown the patterns that start with the prefix c. In the complete BPFSP-Tree, we shall have all the patterns that are observed as frequent up to current mining iterations. Traversing from root to a particular node we extract a frequent pattern. Each BPFSP-Tree node holds the pattern's projection nodes and their support. By numbering the nodes or providing ids, we can use bitset format to store a set of projection nodes where we set the bits of the corresponding node ids. Like, for $N_{\langle c \rangle} = \{4, 8, 12, 16, 33, 21\}$, we set the corresponding bits to save the projection information. This is possible because node references can be re-used across many patterns. This strategy will achieve significant memory reduction with some additional runtime cost executed during unpacking. Also, to remove the obsolete infrequent patterns we start from the leaf nodes of each path, from the nodes making patterns, $\langle (cd)(a) \rangle, \langle (c)(d) \rangle, \langle (c)(c) \rangle, \langle (c)(bc) \rangle, \langle (c)(ac) \rangle, \langle (c)(abc) \rangle$ It is intuitive because of the apriori property that these might get infrequent earlier. To traverse the leaf nodes we follow the end-links that are only present in the leaf nodes, shown in green color in Fig. 8(e). This helps not to traverse the complete tree to reach the leaf nodes but directly reach there.

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 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 an idea regarding the items that 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

Table 6
Example of projection in modified and unmodified subtrees.

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

H_P denote a pattern P's actual and heuristic support respectively. In Line 4, we apply CETable based or co-occurrence information based pruning over the lists. Then, we perform projection in the subtrees to discover the frequent extended patterns, for SE in lines 6–10 and for IE in line 14. Then, we recursively progress for the extended patterns in lines 15–16 and 17–18 for SE and IE respectively with the updated corresponding $sList$ and $iList$ for each. Fig. 9 shows how patterns are recursively discovered in Tree-Miner. To visualize, we have shown the generated patterns with the prefix $\langle (a) \rangle$ and considered $\delta = 2$. For each pattern we have shown its support, its initial $sList$, $iList$, shrunk $sList'$, $iList'$, and corresponding projection nodes. Node numbers can be mapped to the exampled SP-Tree shown in Fig. 7.

Algorithm 4 Tree-Miner

1: **Globals:** $\delta = \lceil min\_sup \times |D| \rceil$
2: **Output:** Complete set (F) of frequent sequential patterns.
3: **procedure** TreeMiner($P, N_P, sList, iList$) ▷ CETable based pruning
4: Reduce $sList$ and $iList$ based on CETable.
5: $N_{SE} \leftarrow \{\}, N_{IE} \leftarrow \{\}$ ▷ Nodes which will perform extension
6: **for** each item $\gamma \in sList$ **do** ▷ SE checking
7: $\forall n \in N_P$ perform SE for $\gamma(P\{\gamma\})$
8: **if** $(S_{P\{\gamma\}} < \delta)$ **then** $sList \leftarrow sList - \{\gamma\}$ ▷ Infrequent Pattern
9: **if** $(H_{P\{\gamma\}} < \delta$ and $\gamma \in iList)$ **then** $iList \leftarrow iList - \{\gamma\}$ ▷ 4.3.3
10: **else** $N_{SE}[\gamma] \leftarrow N_{P\{\gamma\}}$ ▷ Frequent Pattern, tracking extension nodes
11: **for** each item $\gamma \in iList$ **do** ▷ IE checking
12: $\forall n \in N_P$ perform IE for $\gamma(\{P\gamma\})$
13: **if** $(S_{\{P\gamma\}} < \delta)$ **then** $iList \leftarrow iList - \{\gamma\}$ ▷ Infrequent Pattern
14: **else** $N_{IE}[\gamma] \leftarrow N_{\{P\gamma\}}$ ▷ Frequent Pattern, tracking extension nodes
15: **for** (each item $\gamma \in sList$) **do** ▷ Recursive SE
16: TreeMiner($P\{\gamma\}, N_{SE}[\gamma], sList, \{\alpha | \alpha \in sList \cap \alpha > \gamma\}$)
17: **for** (each item $\gamma \in iList$) **do** ▷ Recursive IE
18: TreeMiner($\{P\gamma\}, N_{IE}[\gamma], sList, \{\alpha | \alpha \in iList \cap \alpha > \gamma\}$)

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: (i) some infrequent patterns become frequent (patterns with completely new prefix or new suffix with the existing ones), (ii) some frequent patterns become infrequent and (iii) some frequent

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. 9. Simulation of Tree-Miner for the patterns with prefix $\langle(a)\rangle$.

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. 10. Simulation of IncTree-Miner for the patterns with prefix $\langle(c)\rangle$.

Algorithm 5 Pattern Formation Block of IncTree-Miner

1: **procedure** PatternFormation($P\gamma$, N_P^M, B_P)
2: $\forall n \in N_P^M$ perform implicit projection in modified subtrees for γ
3: **if** $P\gamma \in B_P.child$ **then** ▷ previously frequent, $B_{P\gamma}$ node exists
4: **if** $S_{P\gamma}^{D'} \geq \delta'$ **then** update support and projection in $B_{P\gamma}$.
5: **else** ▷ Got Infrequent
6: Remove $B_{P\gamma}$ and its subtree, adjust end-links.
7: **if** (complete projection done) **then** ▷ Save support in NIB
8: $B_P.NIB[\gamma] \leftarrow S_{P\gamma}^{D'}$
9: **else if** $\gamma \in B_P.NIB$ **then** ▷ Previous complete support in NIB
10: **if** $S_{P\gamma}^{D'} \geq \delta'$ **then**
11: Perform projection in unmodified subtrees for γ.
12: Create node $B_P.child[\gamma]$, adjust end-links, remove $B_P.NIB[\gamma]$
13: **else** ▷ Still Infrequent
14: **if** (complete projection done) **then** $B_P.NIB[\gamma] \leftarrow S_{P\gamma}^{D'}$
15: **else** Delete $B_P.NIB[\gamma]$ ▷ Not full projection calculated
16: **else if** $P\gamma \notin B_P.child \cap \gamma \notin B_P.NIB$ **then**
17: **if** $S_{P\gamma}^{db} \geq \delta' - \delta + 1$ **then** ▷ Infrequent to Frequent Transition
18: Perform projection in unmodified subtrees for γ.
19: **if** $S_{P\gamma}^{db} + S_{P\gamma}^{D} = S_{P\gamma}^{D'} \geq \delta'$ **then** Create $B_P.child[\gamma]$,adjust links
20: **else** ▷ Infrequent Pattern
21: **if** (complete projection done) **then** $B_P.NIB[\gamma] \leftarrow S_{P\gamma}^{D'}$
22: **if** $H_{P\gamma}^{M} + H_{P\gamma}^{U} < \delta'$ **then** ▷ Heuristic Pruning during SE
23: Remove γ from $iList$ of pattern P
24: **if** $P\gamma$ is frequent $\cap$ $N_{P\gamma}^{M} \neq \{\}$ **then**
25: Return $N_{P\gamma}^{M}$ ▷ $P\gamma$ has update in subtree, need further checking
26: **else** Return *null* ▷ infrequent/no update in underlying subtree for $P\gamma$

Algorithm 6 IncTree-Miner

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

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 4.4 (*Incremental Support*)**.** Suppose, we have a pattern P ending at nodes $N_P = \{n_1, n_2, n_3, n_4, n_5\}$ 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 4.5 (*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. Fig. 8(a) shows 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). Table 6 shows two examples of how, IncTree-Miner performs 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. 4 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, lines 3-8) or its previous complete support is in NIB (Algorithm 5, lines 9–15) or it was previously infrequent (Algorithm 5, lines 16–23). We simulate each case separately. We always first project in modified subtrees (Algorithm 5, line 2) and based on the results we take the decision to perform projection in unmodified subtrees. We start with the items that were found in IncDB to make the initial $sList^M$ and $iList^M$ (Algorithm 6, line 3), gradually shrink them and track the patterns that were affected or had modifications in underlying subtrees in lines 5–7 of Algorithm 6 similar to the Tree-Miner algorithm discussed prior. We clear the NIB for the items for which no projection was done in the current iteration to maintain consistency (Algorithm 5 lines 6 and 15). Finally, we remove the unmodified previously frequent but currently infrequent patterns from BPFSP-Tree in a bottom-up manner (Algorithm 6, lines 18-19). In Fig. 10, a small visualization of IncTree-Miner has been shown where it can be seen, how IncTree-Miner recursively discovers the patterns with the prefix $\langle(c)\rangle$. 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. 8(e), the status of the BPFSP-Tree for the patterns with prefix $\langle(c)\rangle$ in the old and new database is shown which also reflects how pattern distribution can significantly vary due to database modification. From only two patterns $\langle(c)\rangle, \langle(c)(d)\rangle$ in D, a group of a new set of frequent patterns with prefix c arrives with the modification of the database to D'. To refer the nodes, the IncSP-Tree shown in Fig. 4 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 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. 8(d). It illustrates that, when we need to know the frequency status of $\langle(\alpha)(\beta)(\gamma)\rangle$, 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 also embed some over computing buffer based approaches here.

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 (Pei et al., 2004), CM-SPAM and CM-SPADE (Fournier-Viger et al., 2014) and to compare IncTree-Miner we considered IncSP (Lin & Lee, 2004) and PBIncSpan (Chen et al., 2007). 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 @ 3.90 GHz × 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. Here, some short terms have been used for page constraints, such as, sequence count (SC), unique items (UI), average sequence length (ASL), etc. 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 incremental database and represent the common phenomena, we followed the approach mentioned in the relevant literature (Lin & Lee, 2004). 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 construct an example using Tables 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 Lin and Lee (2004).

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 dense (with high ratio of d^*) and comparatively much lesser dense.

5.2. Tree-miner

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

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

Table 7
Dataset summarized information.

Name	Description	SC	UI	ASL	C	T	d^*
Bible	Conversion of Bible	36,369	13,905	21.64	10.97	15.49	0.16
MSNBC	Click Stream	31 790	17	13.23	7.83	10.81	77.82
BMSWebView1	Click Stream	59 601	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	990 000	41 270	8.1	3.7	6.23	0.02
L20k-C10 -T5-N1K	Synthetic	20 000	1000	44.21	10	5	4.42
L50k-C10 -T2.5-N10K	Synthetic	50 000	10 000	21.79	10	2.5	0.22

Table 8
Variables' description.

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

5.2.1. Runtime comparison with core literature

In this section, we shall experiment with our proposed mining algorithm SP-Tree based Tree-Miner with the core algorithms that approach the generic support-based SPM problem.

Fig. 11(a)–(g) presents the runtime performance of Tree-Miner for Sign, Bible, BmsWebView1, MSNBC, Kosarak, L20k-C10-T5-N1K and L50k-C10-T2.5-N10K datasets respectively. From the corresponding figures, it is obvious that our proposed Tree-Miner based on SP-Tree performs comparatively better than other experimented static mining algorithms. SP-Tree is a compact representation of the database that provides a huge structural control over it. Through the next links, we perform efficient traversals in the database which ultimately helps to generate the patterns faster. In the figures, in lower thresholds, our performance improvement is quite visible because we need to encounter a a good 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.

Now, we present a brief discussion on the important factors that control the proposed solutions' performance and how the compared algorithms comply with them to understand the underlying reasonings that have played roles behind our solution's improved runtime.

1. Effect of breadth-first based support counting strategy: During extension, we apply a runtime maximum possible support estimation compared to the item-by-item calculation over the projected database in CM-SPADE and calculating the set bit value over all the blocks equal to the number of transactions

during each extension to grasp the support in CM-SPAM. Here, the early estimation might have helped to prune many infrequent patterns before performing a complete projection that is asymptotically bounded by each pattern's support value.

2. Applied Pruning strategies: The CETable based pruning strategy is also used in CM-SPADE and CM-SPAM. The $sList$ and $iList$ based pruning strategy is also present in CM-SPAM. The heuristic pruning strategy does not add any additional runtime cost but comes as a by-product during extension. S^{MAL} attribute helps to make a decision that do we need to perform any type of projection in the underlying subtree or not. PrefixSpan lacks behind these strategies. So, overall, we adopt all the existing pruning strategies in our solution along with some new ones that do not add any major runtime bottleneck.
3. Nature of traversing: PrefixSpan and CM-SPADE traverse over each projected database to calculate the support over all the possible pattern extensions. CM-SPAM and Tree-Miner try to find the extensions for the items that are tracked in the corresponding lists, $sList$, and $iList$. So, when the average length of the sequences is larger, the performance of PrefixSpan and CM-SPADE starts to get reduced. Additionally, to calculate the support, CM-SPAM performs block-based bitwise operation between the data structures of two patterns to check the extendability over the support threshold. Here, the number of blocks depends on the number of transactions. So with the increased database size, this might become costly. Tree-Miner performs node projections by traversing through the next links in reduced subtrees during each extension bounding over the maximum possible support. So, in each extension, it is not heavily biased over the number of transactions. All of these factors help to achieve a good runtime improvement for Tree-Miner.
4. Representation Characteristics: SP-Tree tries to compact the tree based on prefix sharing. So, in the denser database, the tree will be compacter. Based on compactness, the number of projected nodes of a pattern will differ. Also, the ancestor-descendant relationship among the nodes helps to create the next links to jump among the nodes during pattern extension. This feature is very helpful to find the projected nodes for a particular pattern in the database (aka tree) compared to the linear item-by-item traversal and block-by-block bitwise operation over the number of transactions. In query-based problems, this feature can be very useful.

5.2.2. Runtime comparison with applied literature

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. In this section, we shall compare our proposal with some very recent algorithms that have addressed various aspects related to SPM problem. They do not propose completely core solution to the SPM problem but rather extends the traditional approaches with some new concepts and strategies.

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 two recent literatures (He et al., 2019; Tarus et al., 2018). He et al. (2019) 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. Tarus et al. (2018) 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. 12. For the comparisons, all the base datasets have been used here, multiple single itemed sequential events.

Fig. 12(a) and (b) shows the results of Tree-Miner embedded performance improvement where Disc denotes the applied solution in literature (He et al., 2019) 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 Tarus et al. (2018). A recommendation system was 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 proposal. Fig. 12(c) and (d) shows 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 Hosseininasab et al. (2019), 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 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 $\upsilon \in N_\alpha$. 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 get $S_{\alpha\beta} = 9 - 3 + 1 < \delta$. 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. 13.

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.

We have also compared our solution with Islam et al. (2022), a study that addresses weighted SPM problem. To make a fairground comparison we considered all the items equally weighted as 1. In the current article, they consider a metric where they multiply a pattern's weighted average with its support in the database and then divide it with tmw. tmw is a variable that sums up the highest weighted item's weight found in each sequence. So in our setup, as all the items' weights are 1, each pattern's average weight becomes 1 and tmw becomes the size of the database. Now, the problem becomes finding such patterns that satisfy a minimum support threshold, e.g. $\frac{support}{|D|}$. With this setup, we can convert the weighted SPM problem to the generic SPM problem.

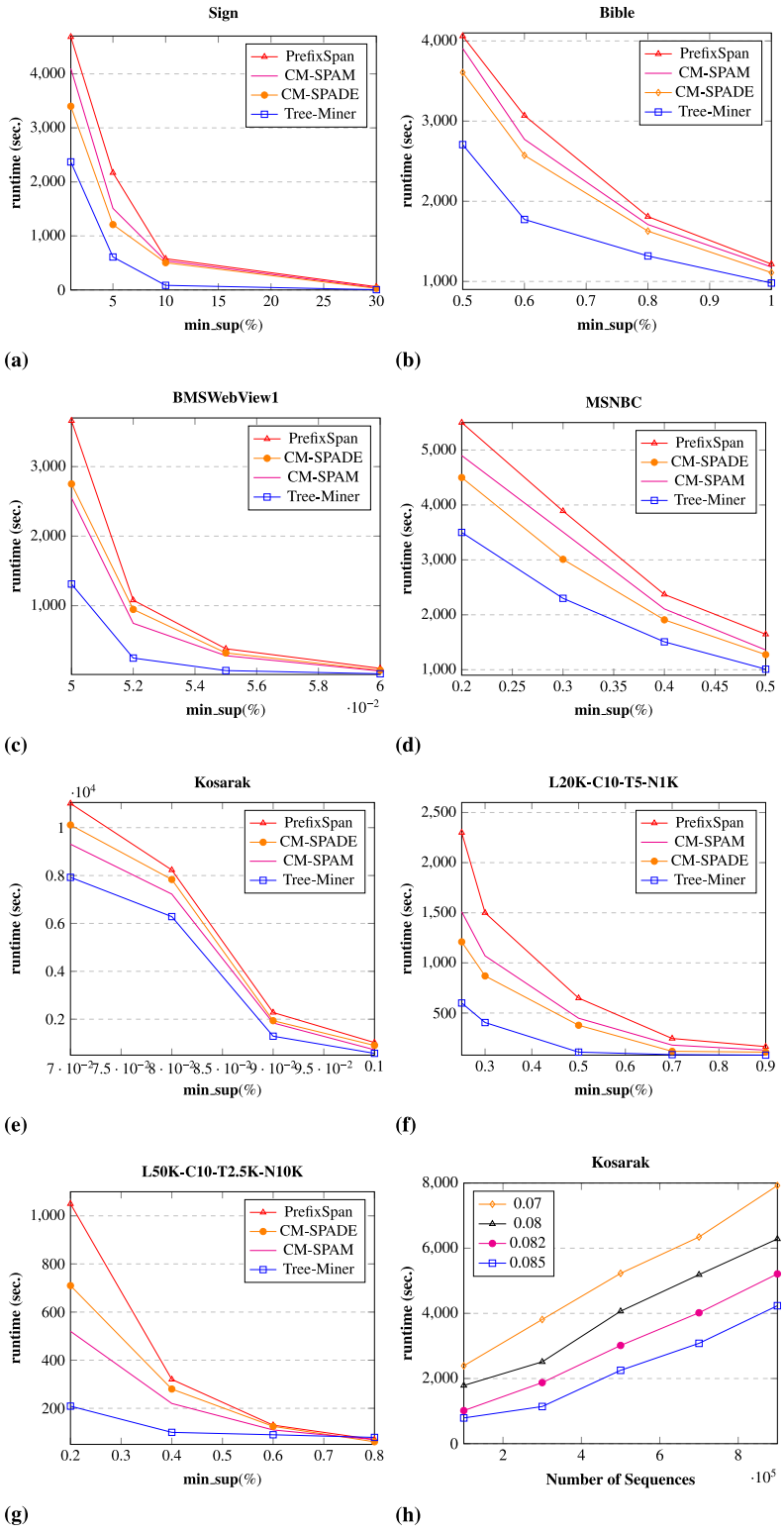


Fig. 11. (a)–(g): Runtime comparisons, (h) Scalability comparison for Tree-Miner.

Fig. 14 shows the results observed between FWSPM (Islam et al., 2022) and Tree-Miner (Islam et al., 2022) in three datasets. From our observation, it was found that Tree-Miner provides a quite improved runtime compared to the FWSPM algorithm. The underlying cause is, the FWSPM algorithm applies PrefixSpan alike projected database format with a pruning strategy denoted in the current article as $MaxPWS$. $MaxPWS$ mainly tries to assume the maximum weighted average possible for a pattern considering different length

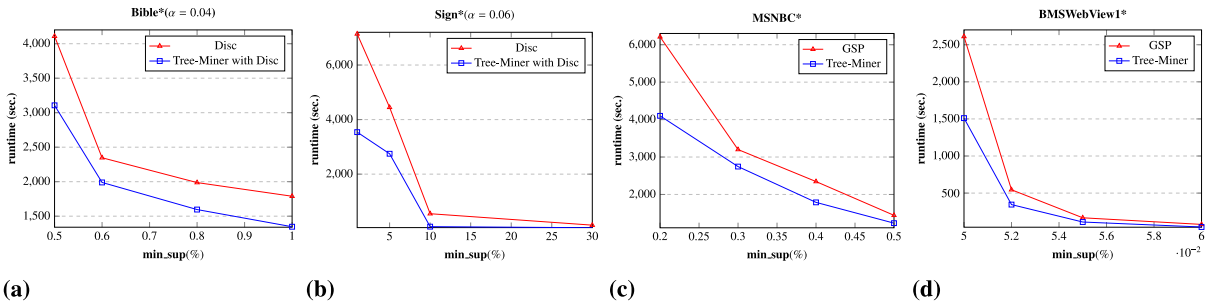


Fig. 12. Runtime improvement using Tree-Miner over GSP based literature.

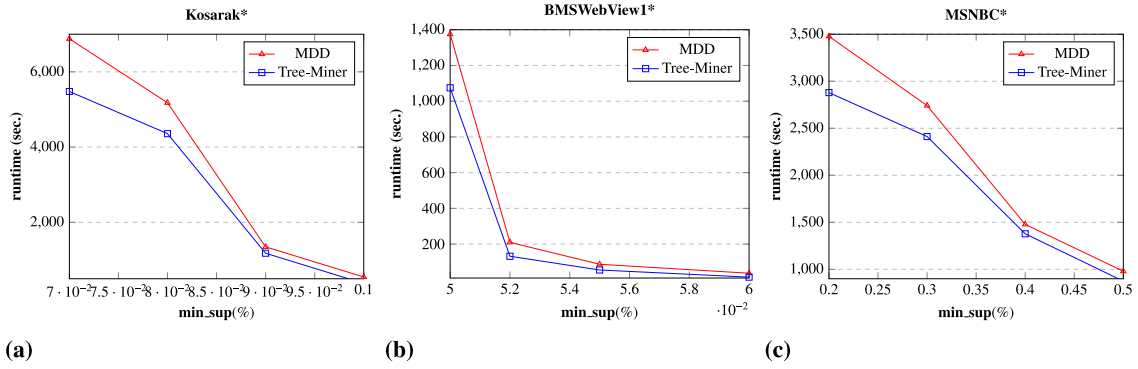


Fig. 13. Runtime comparisons between Tree-Miner and MDD.

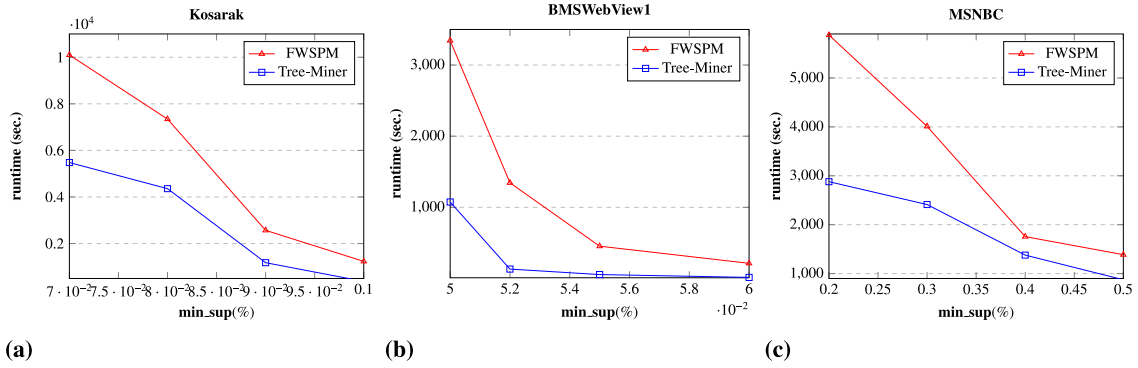


Fig. 14. Runtime comparisons between Tree-Miner and FWSPM.

extensions. As they have to calculate the metric considering different possible lengths, it is a costly operation along with they need to traverse the projected database completely in this regard. Also, in our setup, items are equally weighted, so such pruning strategy did not help much.

5.2.3. *Memory usage*

Due to the structured representation of the information, our solution needs comparatively more memory. Now, we shall highlight the factors that control the memory usage of our static solution,

1. SP-Tree: SP-Tree represents the complete database in a tree-based format. So the raw database is no more required. Each node holds some attributes to establish the ancestor-descendant relationships. Here, the most notable attribute is next link alike attributes that help to perform jumping in the tree for each unique items in the subtrees and implemented using hash maps for the compactness. The tree also tries to conduct prefix sharing, so in the denser databases the SP-Trees will be compacter.
2. CETable: CETable holds the co-occurrence information among two length ordered items (Fournier-Viger et al., 2014). For static mining, CETable can hold only the pairs information that satisfy the minimum support threshold. Using the SP-Tree's next links the necessary nodes can easily be traversed and the CETable can be updated similar to how the mining actually proceeds. Also, the sequence summarizer can easily be used in this regard based on the idea stated in Fig. 2. But, for static mining, such resource requirement is not necessary.
3. During Mining: During mining, the projected nodes are first extracted for the extended pattern and if it satisfies the support threshold then at each step another one length pattern progression occurs. An important feature of SP-Tree based solution is the ability to compress the projection nodes in bit-based formation. If we can number the node ids, then setting only the corresponding bits of a value we can represent a set of projection nodes in a very compressed format. At any time extracting the bits that are set of the value, we can get the complete set of projection nodes. For very resource-limited systems, we can perform the optimization. Though in our setup of our experimentation, we did not require it.

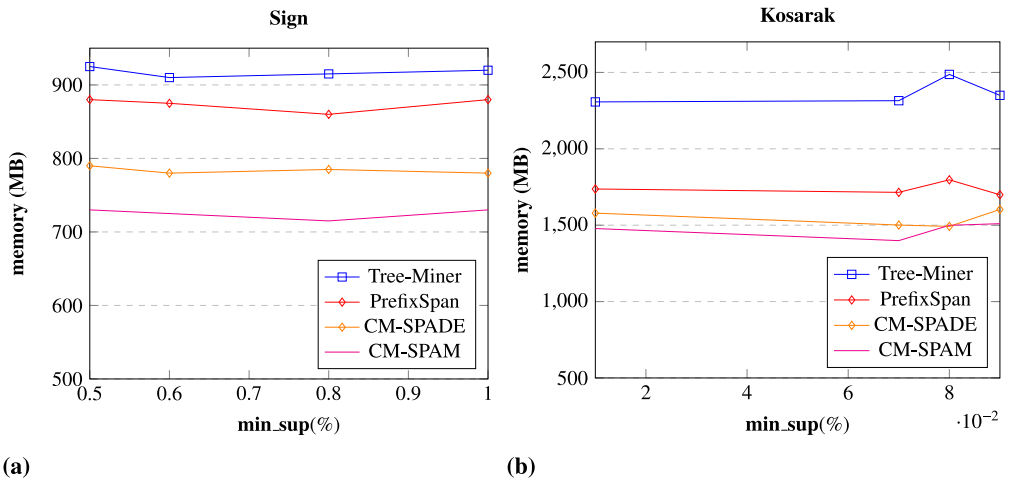


Fig. 15. Memory comparisons for Tree-Miner.

PrefixSpan uses the database projection technique where with each extended pattern it considers a portion of the database considered as the projected database. CM-SPADE uses lattice alike structures in vertical format and CM-SPAM uses bit based array structures to perform projection and calculate the support of the patterns. CM-SPAM creates blocks considering the number of transactions and the maximum number of event size. Fig. 15 shows the results for two datasets. From the figure, it can be observed that, Tree-Miner takes slightly more memory compared to the remaining algorithms. We have already discussed in the aforementioned paragraphs regarding the underlying reasoning of such. But, the notable point is that, though being a structural approach it takes some additional memory but alongside it provides a good runtime improvement in overall mining the complete set of frequent sequential patterns. We have also pointed how the memory usage can be reduced considering various strategies. These strategies will lead to memory reduction with some additional runtime costs.

5.2.4. Scalability

We have also conducted a scalability analysis of Tree-Miner and presented the result in Fig. 11(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 observe 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 comparison with core literature

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

Table 9
Construction time vs mining time.

Dataset	Construction Time (s) (C)	Mining Time (s) (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

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.

Fig. 16(a)–(g) shows 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 projection 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.

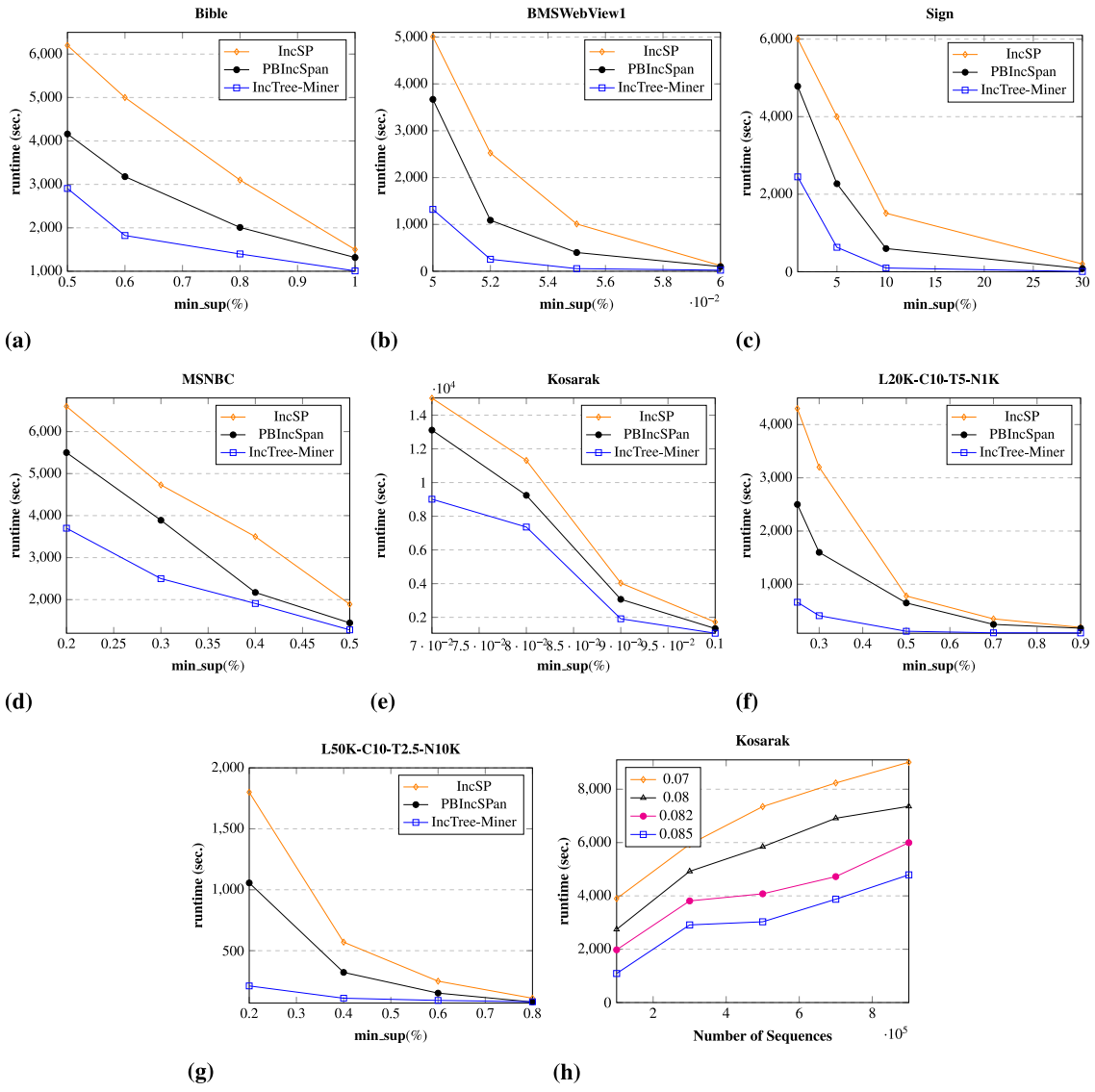


Fig. 16. (a)-(g): Runtime comparison, (h) Scalability comparisons for IncTree-Miner.

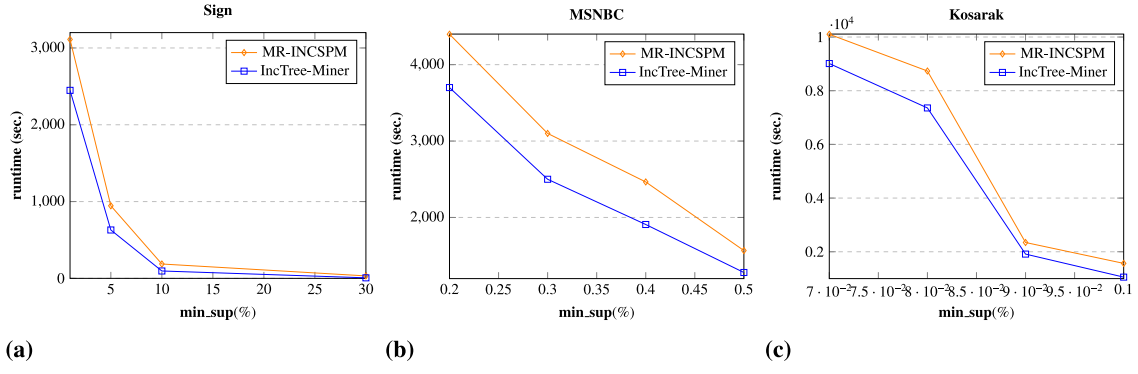


Fig. 17. Runtime comparison with MR-INCSPM.

5.3.2. Runtime comparison with applied literature

Up to 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 (Saleti & Subramanyam, 2019) 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.

Fig. 17 shows 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.

Vo et al. (2021) proposed two methods to mine clickstream sequential patterns from incremental databases. Clickstream database is comparatively simpler than a generic sequential database — in the prior one each event will only consist of a single item. The problem addressed in the study considered only the newly inserted sequences but not the appended sequences during database modification where we consider both. Also, they assumed in their work that, the size of the newly inserted sequence will be much smaller compared to the existing old database. Based on that, they tried to apply the pre-large concept (Lin et al., 2015). They used SPPC-Tree (Vo et al., 2021) as the data structure which cannot handle multiple itemed events. Another important limitation is that they had to maintain both the raw complete updated database and the corresponding compressed SPPC-Tree because during creation they omitted some items based on the used thresholding strategies. Apart from these, the study did not mention any specific strategy to store the frequent patterns across successive mining iterations.

In the pre-large concept, two thresholds, upper (s_u) and lower (s_l) support thresholds are used to decide to re-scan the complete database or the SPPC-Tree or not. The equation that embeds both of these thresholds and calculates a safety threshold is given in Eq. (1). There can be two types of cases,

1. $s_u \approx s_l$: $f \approx 0$, leads to the data structure SPPC-Tree being very compact, it will try to re-scan the updated database completely at each iteration to update the tree, the number of mined pre-large patterns will be very negligible. Almost all the mined patterns will belong to the set of frequent patterns.
2. $s_u \gg s_l$: $f \approx \frac{s_u}{1-s_u} \times |D|$, leads to comparatively a larger SPPC-Tree holding almost the complete updated database each time, the number of database re-scans will be smaller. The set of pre-large patterns will be very high which is not frequent but needs to be kept updated to be tracked.

$$f = \frac{s_u - s_l}{1 - s_u} \times |D| \tag{1}$$

s_u is always set to δ. It is quite evident that setting both s_u and s_l is very challenging and can control the performance of the algorithms to a great extent. These parameters are also very experimental as shown in Vo et al. (2021). As in this study, we want to focus completely on the frequent pattern set rather than depending on the pre-large buffer, the first strategy of $s_u \approx s_l$ aligns with our motivation more. For each experimented dataset, s_u was set to δ and s_l was set to $s_u - 0.001$. Also, for the comparison, for each dataset, each event was considered as single itemed. To fit Eq. (1), the thresholds were scaled between $(0, 1]$. For the experiments we considered $|D| = 90\% \times |D_{raw}|$, the remaining 10% were kept in D_{insert}.

Vo et al. (2021) proposed two algorithms namely inCMUB and Eff-inCMUB, to mine frequent clickstream patterns from the SPPC-Tree. The main difference between the two algorithms is, CMUB tries to update the existing SPPC-Tree and based on that generates the complete set of frequent patterns whereas Eff-inCMUB creates another small SPPC-Tree and using this new tree and pre-large patterns mines the complete set of frequent patterns. As we have not used the dependency over the pre-large buffer, the Eff-inCMUB algorithm could not provide us with much improvement. Thus We have compared our IncTree-Miner with inCMUB and shown results in two datasets in Fig. 18.

Also, as we have set s_l to a very close value to s_u, we had to re-scan the complete database to reconstruct the SPPC-Tree in the second iteration. Vo et al. (2021) used the SPPC-Tree node-based B-List structure to order the nodes that are used during mining. Joining over these nodes new patterns are constructed and their support values are updated. From our observation, joining over the B-List nodes is quite expensive. For each entry in the B-List of pattern A you search for another entry in the B-List of pattern B that matches the extension constraint of being in the same subtree. Among all of such computed nodes, a group of them contributes to the actual support and others are kept for next level pattern extensions. In contrast with our approach, we complete projecting up to a set of ancestor nodes not each possible concatenated node in each subtree. Fig. 18 matches our intuition. We have found that IncTree-Miner provides a much improved runtime compared to the inCMUB algorithm. With lower thresholds, the number of the mined pattern increases leading to a greater required time for mining.

5.3.3. Memory usage

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. 19(a)–(b) over Bible and Kosarak dataset. In the following points, we highlight the major factor that control the memory usage of IncTree-Miner in brief.

1. Tree-Based Inheritance: IncTree-Miner uses IncSP-Tree which inherits all the properties of the static SP-Tree with some newly added attributes, such as modified next links and count attributes. As stated in Section 5.2.3, IncTree-Miner also bears similar type of factors. Modified next links which are a subset of next links help to faster traverse or jump only in the modified subtrees.
2. Sequence Summarizer and CETable: We have already stated the procedure to update CETable in Section 5.2.3. But, in incremental environment, sequence summarizer can be quite useful in this regard. We can use it to update the CETable only for the corresponding updated sequence — by tracking only the new items with the older existing ones. We can also use only the modified next links and by traversing in the modified subtrees we can get a hold of two length modified patterns, but it will add some traversing costs. In a memory constrained environment, we can adopt this approach. In our setup, the prior one did not pose any problem.
3. BPFSP-Tree as pattern storage: As the pattern storage structure, we used BPFSP Tree to store the frequent sequential patterns and used it across successive mining iterations. As, already stated, using a pattern storage structure is very common in ISPM problems (Lin et al., 2015; Vo et al., 2021). But, important factor to note that, sometimes the projection information is also stored — to merge the projected databases between the old and newly

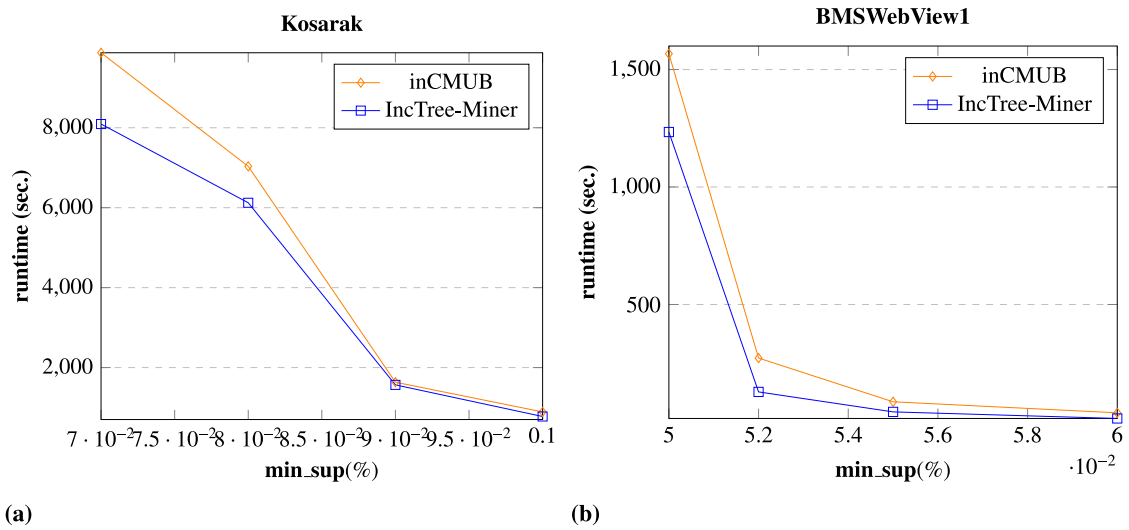


Fig. 18. Runtime Comparison with inCMUB based on SPPC-Tree.

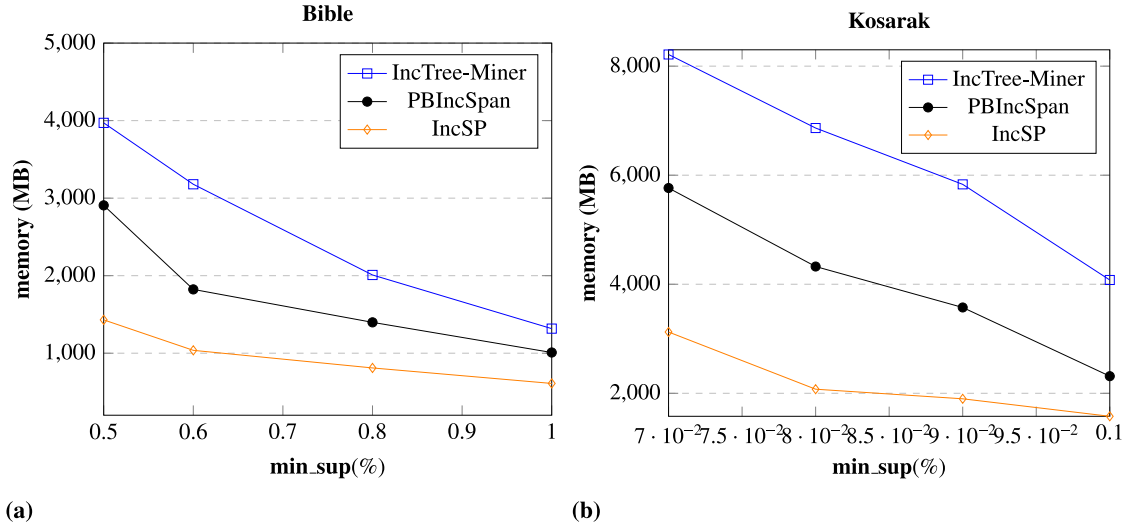


Fig. 19. (a)-(b) Memory evaluation.

added portions. We store the IncSP-Tree nodes in this regard to denote, for a particular pattern, up to which node traverse has already been done. As stated in Section 5.2.3, bit-based strategy over numbering the nodes can be quite useful in this regard to reduce the memory cost. For example, let node no 1, 3 and 7 are required to store the projection for a particular pattern, then storing the nodes references, only the value $(10001010)_2 = 2 + 8 + 128 = 138$ will be enough. As prior discussed, unpacking of these values during mining adds some runtime cost. For our experiments as shown in Figs. 16 and 19, we did not use this bit based idea, as we did not face memory issues. Also, note that, we may not even need to store the projection information, rather we can traverse in the IncSP-Tree for each concerned pattern whenever we need using next link like attributes — such attributes help to jump in the tree for any item from any node in its subtrees to get the required extended node which comes with some additional runtime cost. We consider this strategy as the memory resilient IncTree-Miner.

4. During Mining: The mining progresses recursively with gradual extension over a pattern with its projection list of IncSP-Tree nodes. From such patterns, we discover one-length extended frequent super patterns and progress similarly.

IncSP stores the support of the prior frequent patterns. So, its memory usage is dependent on the number of frequent patterns. PBIncSpan keeps the previous frequent patterns' support along with their projection information where PBIncSpan uses pseudo projection. We have already stated how IncTree-Miner takes strategies to address this measure in the aforementioned points. We have also mentioned in our study that in denser datasets, the constructed SP-Trees will be compacter. The compactness of the SP-Tree also controls the memory and runtime factors. In Fig. 19, we observe that our solution takes some additional memory. We have already discussed the underlying causes and the possibility to improve such. Our main goal was to improve runtime. In the earlier section, we have shown how our approach was able to provide a significant runtime improvement. We can adopt the prior mentioned strategies, but it might cause a trade off between runtime and memory.

5.3.4. *Scalability*

Similar to Tree-Miner, we also conducted a scalability test for IncTree-Miner and presented the result over the *Kosarak* dataset in Fig. 16(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,

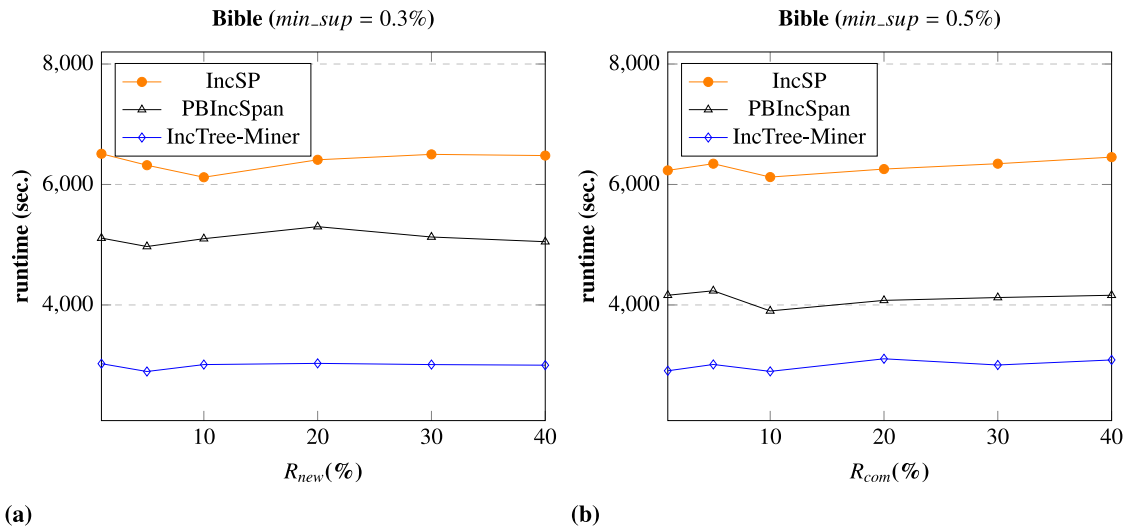


Fig. 20. (a) Performance over R_{new} (%) and (b) R_{com} (%) for IncTree-Miner.

we chose $R_{new} = 10\%$, $R_{com} = 50\%$ and $R_{prev} = 80\%$ over the considered number of transactions. Section 5.2 shows 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.

5.3.5. 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. First, we extract $R_{new}\%$ from D_{raw} to put into D_{insert}. Then, over the remaining portions, we keep $R_{com}\%$ sids in both D and D_{append} and over the common sids we keep $R_{prev}\%$ items in D and the remaining ones in D^{append}. Here, to conduct experiments, we have kept, $R_{com} = 50\%$, $R_{prev} = 80\%$ and $min\_sup = 0.3\%$. We have shown the result in Fig. 20(a). We started with 1% and gradually increased it to 40% 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. Observed from Fig. 20(a), 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. 20(a) supports our intuition.

Similar to R_{new}, we also evaluated our solution by changing the percentage of R_{com}. For the experiment, we had set $R_{new} = 10\%$, $R_{prev} = 80\%$ and varied R_{com} with $min\_sup = 0.5\%$ over the Bible dataset and shown the results in Fig. 20(b). 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 the support calculation of the extensions, we discover some patterns' infrequency. The algorithm's runtime improves depending on how quickly it can detect the infrequency rather than performing complete projection. In Fig. 21, we present an analysis to understand how quickly our proposed pruning technique detects

Table 10
Interactive mining: performance in descending *min_sup*.

Dataset		Descending *min_sup*			
Bible	*min_sup* (%)	10	1	0.8	0.5
	T (s)	228.2	579.2	675	935.5
BMSWebView1	*min_sup* (%)	0.06	0.055	0.052	0.05
	T (s)	19.8	71	226.5	865.2
Sign	*min_sup* (%)	30	10	5	1
	T (s)	19.8	71	226.5	865.2
MSNBC	*min_sup* (%)	0.085	0.08	0.075	0.07
	T (s)	1037.4	492.7	829	742.2
Kosarak	*min_sup* (%)	0.085	0.08	0.075	0.07
	T (s)	3037.3	2512	1231.4	1021.3
L20-C10-T5-N1K	*min_sup* (%)	0.04	0.035	0.03	0.025
	T (s)	313.7	221.5	325.3	710
L50-C10-T2.5-N10K	*min_sup* (%)	0.031	0.03	0.025	0.02
	T (s)	313.7	221.5	325.3	710

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 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 tested, 45% of them are detected performing only 50% projection of their corresponding complete projected database. In a numeric example, let us say we tried to perform projection for 200 infrequent patterns in the SP-Tree. Each pattern has a support value. We either discover all the projected nodes for an infrequent pattern and understand that it will not be frequent or during the gradual discovery of the projection nodes at one point before completing the whole projection we detect its infrequency. Here, for $200 * 45\% = 90$ infrequent patterns, we were able to detect their infrequency early. Also, let us say the support of these patterns was $min\_sup = 100$, so after processing around $100 * 50\% = 50$ projection nodes we found that support has reduced drastically. Another connected pruning strategy comes from the S^{MAL} attribute. S^{MAL} mainly works as a caching mechanism to stop projecting in the underlying subtrees by prior summing the count attributes' values-sort of works as a pre-processing step. Mainly, this attribute reflects or boosts up a knowledge that do we even need to perform projecting underneath or not. While generating Fig. 21, we omitted using S^{MAL} to properly capture the effect of breadth-first based support counting strategy.

From Fig. 21, it is clear that our proposed support counting technique can detect most of the infrequent patterns without performing a

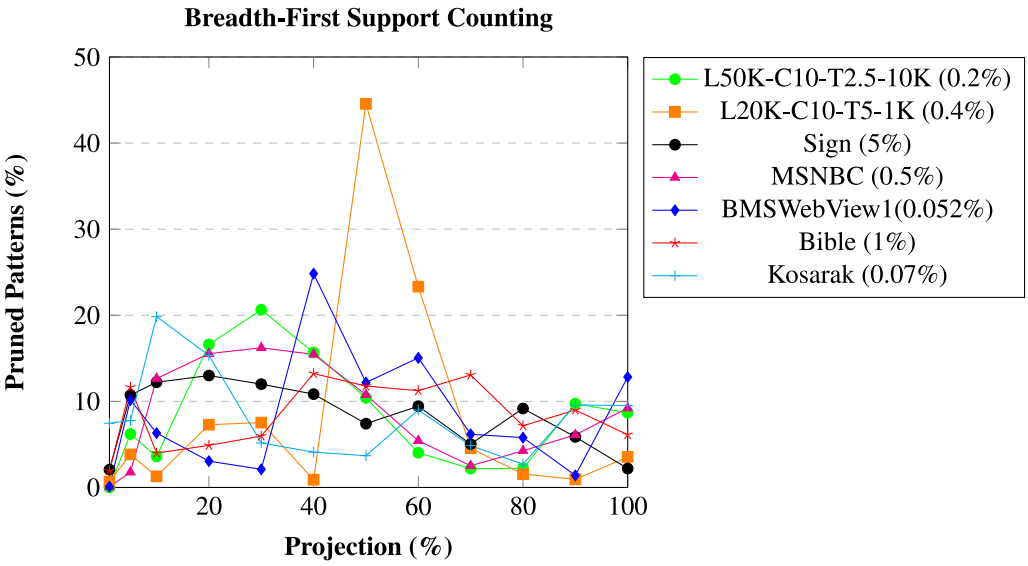


Fig. 21. Evaluation of breadth-first based support counting technique.

complete projection of the underlying sub-database. We have shown the corresponding *min_sup* values beside each legend in Fig. 21. The figure is generated over the data gathered during the execution of Tree-Miner.

5.5. CETable update in incremental environment

In this section, we want to show how using our idea presented in Fig. 2 we can efficiently update CETable with the help of sequence summarizer or *seq_sum*. We mainly want to demonstrate the time factor improvement here. In Fournier-Viger et al. (2014), no specific approach was proposed to calculate the co-occurrence table information CMAP there. So, we have assumed that the considered approach was traversing the complete database representation and during that calculating CMAP. The brute-force equivalent approach for CETable will be traversing the complete SP-Tree or IncSP-Tree and during that updating the information.

Here, we shall present the results in three datasets for an incremental environment. In each dataset, we have calculated the CETable in two ways. Here, *Re-calculate* means traversing the complete IncSP-Tree and re-computing the complete CETable from scratch, and *Using seq_sum* denotes updating the CETable with the help of sequence summarizer based on the idea presented in Fig. 2.

we observe from Fig. 22 that the second approach *Using seq_sum* reduces the time factor that is required to a significant amount. The underlying reasoning is pretty straightforward. In the second approach, we only focus on the newly added items in the incremental database with the previous items that were observed for each sequence. In this regard, we take the help of seq_sum to use the summarized information of the prior appeared items to find the modified two-length combinations. If we compare the first approach of re-calculating we need to traverse the complete tree again. It is intuitive that, during traversing the complete tree we need to reach many more nodes or items leading to the additional cost. So, the main concern of this experiment was to observe not only the time reduction but also the corresponding improvement factor. In the experiments, we achieved around 70% time reduction. The larger databases and sparser graphs increase the factor more. To compress memory and achieve constant time addressing complexity, our CETable and seq_sum implementation were done using hashing over the items and sids respectively.

Table 11
Interactive mining: performance in ascending *min_sup*.

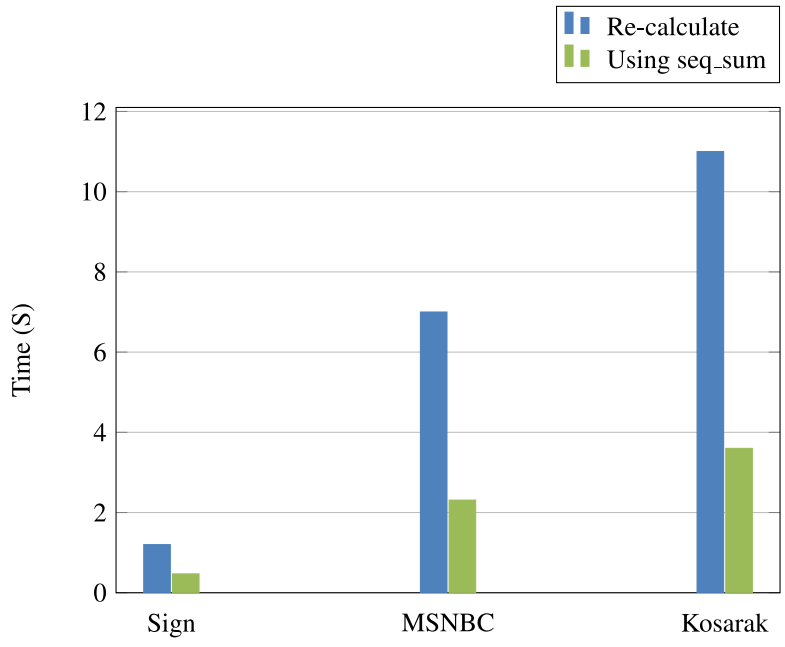
Dataset		Ascending *min_sup*			
Bible	*min_sup* (%)	0.5	0.8	1	5
	T (s)	2907	0.0008	0.0007	0.0003
BMSWebView1	*min_sup* (%)	0.05	0.052	0.055	0.06
	T (s)	1321	5.7	1.1	0.7
Sign	*min_sup* (%)	1	5	10	30
	T (s)	1	5	10	30
MSNBC	*min_sup* (%)	0.2	0.3	0.35	0.4
	T (s)	3700	0.4	0.3	0.2
Kosarak	*min_sup* (%)	0.07	0.075	0.08	0.085
	T (s)	7925	5.1	3.05	0.98
L20-C10-T5-N1K	*min_sup* (%)	0.025	0.03	0.035	0.04
	T (s)	1012	1.14	0.08	0.07
L50-C10-T2.5-N10K	*min_sup* (%)	0.02	0.025	0.03	0.031
	T (s)	2010	0.2	0.11	0.05

5.6. 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% → 10% → 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 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. Table 10 shows 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% → 10% → 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



(a)

Fig. 22. Time consumption variation of CETable while using seq_sum.

infrequent patterns, we use bottom-up traversal strategy of BPFSP-Tree, which will help traverse lesser patterns in this regard. Table 11 shows some results related to this scenario.

5.7. Analytical novelty of IncTree-Miner with IncSP-Tree

Our proposed SP-Tree and IncSP-Tree provides an efficient structural manner to store 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 (Cheng et al., 2004), multiple threshold based concepts (Lin et al., 2015) 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 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. For example, 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 article, 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 a structural advantage that ultimately helps to improve mining performance and handle manipulation over the database. We have also presented a new breadth-first based support counting technique that 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 can 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 various sorts of extendability, and many strategies to address different constraints. We have also explored various aspects related to the proposed solutions' implementations which ultimately lights up the solutions' flexibility. We have also discussed different challenges related to the incremental mining problem, e.g., the usage of sequence summarizer to incrementally update the co-occurrence information of the complete database, concept drift issues, etc. In Section 5, we have provided an in-depth analysis for both of our solutions and showed their efficiency in improving mining runtime. We have also highlighted the major factors that control different performance issues of our proposals along with the core differences with the existing relevant algorithms.

As ongoing and future work, we plan to experiment with our strategies in a wider range of datasets and different scenarios. As our solution addresses a new dimensional angle to approach the SPM problem, it creates opportunities for a wider range of new research for a complete tree-based sequential pattern mining. We aim 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 plan to modify our tree-based solutions to approach the specialized sequential pattern discovery problems (e.g., maximal patterns, closed patterns, top-k patterns, etc.) by using our novel tree structures' properties and utilities.

CRediT authorship contribution statement

Redwan Ahmed Rizvee: Conceptualization, Methodology, Software, Validation, Formal analysis, Investigation, Data curation, Writing – original draft. **Chowdhury Farhan Ahmed:** Supervision, Investigation, Conceptualization, Resources, Writing – review & editing. **Md. Fahim Arefin:** Conceptualization, Validation, Data curation, Resources, Writing – review & editing. **Carson K. Leung:** Conceptualization, Writing – review & editing, Funding acquisition.

Declaration of competing interest

The authors declare that they have no known competing financial interests or personal relationships that could have appeared to influence the work reported in this paper.

Data availability

Data will be made available on request.

Acknowledgments

We would like to express our deep gratitude to the anonymous reviewers of the current article. Their useful comments have played a significant role in improving the quality of this work. This work is partially funded by the Ministry of Posts Telecommunications and Information Technology, Information and Communication Technology Division, Government of the People's Republic of Bangladesh; NSERC (Canada); and University of Manitoba, Canada.

References

Andrzejewski, W., & Boinski, P. (2019). Parallel approach to incremental co-location pattern mining. *Information Sciences*, *496*, 485–505.

Ayres, J., Flannick, J., Gehrke, J., & Yiu, T. (2002). Sequential pattern mining using a bitmap representation. In *KDD* (pp. 429–435). ACM.

Borgelt, C. (2005). An implementation of the fp-growth algorithm. In *Proceedings of the 1st international workshop on open source data mining: frequent pattern mining implementations* (pp. 1–5).

Chang, L., Yang, D., Wang, T., & Tang, S. (2007). Imcs: Incremental mining of closed sequential patterns. In *Advances in data and web management* (pp. 50–61). Springer.

Chen, J. (2009). An updown directed acyclic graph approach for sequential pattern mining. *IEEE TKDE*, *22*, 913–928.

Chen, Y., Guo, J., Wang, Y., Xiong, Y., & Zhu, Y. (2007). Incremental mining of sequential patterns using prefix tree. In *PAKDD* (pp. 433–440). Springer.

Cheng, H., Yan, X., & Han, J. (2004). Incspan: incremental mining of sequential patterns in large database. In *KDD* (pp. 527–532).

Dong, X., Gong, Y., & Cao, L. (2018). e-RNSP: An efficient method for mining repetition negative sequential patterns. *IEEE Transactions on Cybernetics*, *50*, 2084–2096.

Fournier-Viger, P., Gomariz, A., Campos, M., & Thomas, R. (2014). Fast vertical mining of sequential patterns using co-occurrence information. In *PAKDD* (pp. 40–52). Springer.

Fournier-Viger, P., Lin, J. C.-W., Kiran, R. U., Koh, Y. S., & Thomas, R. (2017). A survey of sequential pattern mining. *Data Science and Pattern Recognition*, *1*, 54–77.

Gan, W., Lin, J. C.-W., Fournier-Viger, P., Chao, H.-C., & Yu, P. S. (2019). A survey of parallel sequential pattern mining. *TKDD*, *13*, 1–34.

Grahne, G., & Zhu, J. (2005). Fast algorithms for frequent itemset mining using fp-trees. *IEEE TKDE*, *17*, 1347–1362.

Guidotti, R., Rossetti, G., Pappalardo, L., Giannotti, F., & Pedreschi, D. (2019). Personalized market basket prediction with temporal annotated recurring sequences. *IEEE TKDE*, *31*, 2151–2163.

He, Z., Zhang, S., & Wu, J. (2019). Significance-based discriminative sequential pattern mining. *Expert Systems with Applications*, *122*, 54–64.

Hong, T.-P., Wang, C.-Y., & Tao, Y.-H. (2001). A new incremental data mining algorithm using pre-large itemsets. *Intelligent Data Analysis*, *5*, 111–129.

Hong, T.-P., Wang, C.-Y., & Tseng, S.-S. (2011). An incremental mining algorithm for maintaining sequential patterns using pre-large sequences. *Expert Systems with Applications*, *38*, 7051–7058.

Hosseininasab, A., van Hoeve, W.-J., & Cire, A. A. (2019). Constraint-based sequential pattern mining with decision diagrams. In *Proceedings of the AAAI conference on artificial intelligence, vol. 33* (pp. 1495–1502).

Huang, G., Gan, W., & Yu, P. S. (2022). TaSPM: Targeted sequential pattern mining. arXiv preprint arXiv:2202.13202.

Huang, J.-W., Tseng, C.-Y., Ou, J.-C., & Chen, M.-S. (2008). A general model for sequential pattern mining with a progressive database. *IEEE TKDE*, *20*, 1153–1167.

Ishita, S. Z., Ahmed, C. F., & Leung, C. K. (2022). New approaches for mining regular high utility sequential patterns. *Applied Intelligence*, *52*, 3781–3806.

Islam, M. A., Rafi, M. R., Azad, & Ovi, J. A. (2022). Weighted frequent sequential pattern mining. *Applied Intelligence*, *52*, 254–281.

Leung, C. K.-S., Khan, Q. I., Li, Z., & Hoque, T. (2007). Cantree: A canonical-order tree for incremental frequent-pattern mining. *Knowledge and Information Systems*, *11*, 287–311.

Li, Y., Zhang, S., Guo, L., Liu, J., Wu, Y., & Wu, X. (2022). NetNMSP: Nonoverlapping maximal sequential pattern mining. *Applied Intelligence*, 1–24.

Lin, N. P., Hao, W.-H., Chen, H.-J., Hao, E., & Chang, C. (2007). Discover sequential patterns in incremental database. *International Journal of Computers*, *1*, 196–201.

Lin, J. C.-W., Hong, T.-P., Gan, W., Chen, H.-Y., & Li, S.-T. (2015). Incrementally updating the discovered sequential patterns based on pre-large concept. *IDA*, *19*, 1071–1089.

Lin, M.-Y., & Lee, S.-Y. (2004). Incremental update on sequential patterns in large databases by implicit merging and efficient counting. *Information Systems*, *29*, 385–404.

Lin, J. C.-W., Li, Y., Fournier-Viger, P., Djenouri, Y., & Zhang, J. (2020). Efficient chain structure for high-utility sequential pattern mining. *IEEE Access*, *8*, 40714–40722.

Liu, J., Yan, S., Wang, Y., & Ren, J. (2012). Incremental mining algorithm of sequential patterns based on sequence tree. In *Advances in intelligent systems* (pp. 61–67). Springer.

Mallick, B., Garg, D., & Grover, P. (2013). Incremental mining of sequential patterns: Progress and challenges. *Intelligent Data Analysis*, *17*, 507–530.

Masseglia, F., Poncelet, P., & Teisseire, M. (2003). Incremental mining of sequential patterns in large databases. *Data & Knowledge Engineering*, *46*, 97–121.

Nam, H., Yun, U., Yoon, E., & Lin, J. C.-W. (2020). Efficient approach for incremental weighted erasable pattern mining with list structure. *Expert Systems with Applications*, *143*, Article 113087.

Okolica, J. S., Peterson, G. L., Mills, R. F., & Grimaila, M. R. (2020). Sequence pattern mining with variables. *IEEE TKDE*, *32*, 177–187.

Parthasarathy, S., Zaki, M. J., Ogihara, M., & Dwarkadas, S. (1999). Incremental and interactive sequence mining. In *Proc. of eighth CIKM* (pp. 251–258).

Pei, J., Han, J., Mortazavi-Asl, B., Wang, J., Pinto, H., Chen, Q., Dayal, U., & Hsu, M.-C. (2004). Mining sequential patterns by pattern-growth: The prefixspan approach. *IEEE TKDE*, *16*, 1424–1440.

Perera, D., Kay, J., Koprinska, I., Yacef, K., & Zaïane, O. R. (2008). Clustering and sequential pattern mining of online collaborative learning data. *IEEE TKDE*, *21*, 759–772.

Rizvee, R. A., Arefin, M. F., & Ahmed, C. F. (2020). Tree-miner: Mining sequential patterns from sp-tree. In *PAKDD* (pp. 44–56). Springer.

Roy, K. K., Moon, M. H. H., Rahman, M. M., Ahmed, C. F., & Leung, C. K. (2021). Mining sequential patterns in uncertain databases using hierarchical index structure. In *Advances in knowledge discovery and data mining: 25th pacific-Asia conference, PAKDD 2021, virtual event, May 11–14, 2021, proceedings, Part II* (pp. 29–41). Springer.

Saleti, S., & Subramanyam, R. (2019). A mapreduce solution for incremental mining of sequential patterns from big data. *Expert Systems with Applications*, *133*, 109–125.

Salvemini, E., Fumarola, F., Malerba, D., & Han, J. (2011). Fast sequence mining based on sparse id-lists. In *ISMIS* (pp. 316–325). Springer.

Slimani, T., & Lazzez, A. (2013). Sequential mining: patterns and algorithms analysis. arXiv preprint arXiv:1311.0350.

Song, W., Liu, L., & Huang, C. (2021). Generalized maximal utility for mining high average-utility itemsets. *Knowledge and Information Systems*, *63*, 2947–2967.

Srikant, R., & Agrawal, R. (1996). Mining sequential patterns: Generalizations and performance improvements. In *International conference on extending database technology* (pp. 1–17). Springer.

Tarus, J. K., Niu, Z., & Kalui, D. (2018). A hybrid recommender system for e-learning based on context awareness and sequential pattern mining. *Soft Computing*, *22*, 2449–2461.

Vo, B., Nguyen, H.-C., Huynh, B., & Le, T. (2021). Efficient methods for clickstream pattern mining on incremental databases. *IEEE Access*, *9*, 161305–161317.

Wang, K. (1997). Discovering patterns from large and dynamic sequential data. *JIIS*, *9*, 33–56.

Wu, Y., Hu, Q., Li, Y., Guo, L., Zhu, X., & Wu, X. (2022). OPP-miner: Order-preserving sequential pattern mining for time series. *IEEE Transactions on Cybernetics*.

Wu, Y., Lei, R., Li, Y., Guo, L., & Wu, X. (2021). HAOP-Miner: Self-adaptive high-average utility one-off sequential pattern mining. *Expert Systems with Applications*, *184*, Article 115449.

Wu, Y., Luo, L., Li, Y., Guo, L., Fournier-Viger, P., Zhu, X., & Wu, X. (2021). NTP-miner: Nonoverlapping three-way sequential pattern mining. *ACM Transactions on Knowledge Discovery from Data (TKDD)*, *16*, 1–21.

Wu, Y., Wang, Y., Li, Y., Zhu, X., & Wu, X. (2021). Top-k self-adaptive contrast sequential pattern mining. *IEEE Transactions on Cybernetics*, *52*, 11819–11833.

Wu, Y., Yuan, Z., Li, Y., Guo, L., Fournier-Viger, P., & Wu, X. (2022). NWP-Miner: Nonoverlapping weak-gap sequential pattern mining. *Information Sciences*, *588*, 124–141.

Yang, Z., Wang, Y., & Kitsuregawa, M. (2007). Lapin: Effective sequential pattern mining algorithms by last position induction for dense databases. In *DASFAA* (pp. 1020–1023). Springer.

Zaki, M. J. (2001). Spade: An efficient algorithm for mining frequent sequences. *Machine Learning*, *42*, 31–60.

Zhang, M., Kao, B., Cheung, D., & Yip, C.-L. (2002). Efficient algorithms for incremental update of frequent sequences. In *PAKDD* (pp. 186–197). Springer.

Zheng, Q., Xu, K., Ma, S., & Lv, W. (2002). The algorithms of updating sequential patterns. arXiv preprint cs/0203027.