New Approaches for Tree-Based Sequential Pattern Mining

by

Redwan Ahmed Rizvee

Exam Roll: Curzon-2301

Registration No: 2014-716-612

Session: 2018-19

Supervised By

Dr. Chowdhury Farhan Ahmed

Professor

Department of Computer Science and Engineering

University of Dhaka

A thesis submitted in partial fulfilment of the requirements for the degree of Master of Science in Computer Science and Engineering

DEPARTMENT OF COMPUTER SCIENCE AND ENGINEERING

UNIVERSITY OF DHAKA

January 26, 2022

Declaration of Authorship

I, Redwan Ahmed Rizvee, declare that the thesis entitled *New Approaches for Tree-Based Sequential Pattern Mining* and the work presented in the thesis are both my own, and have been generated by me as the result of my own original research. This study is the outcome of the investigation performed by me under the supervision of Dr. Chowdhury Farhan Ahmed, Professor, Department of Computer Science and Engineering, University of Dhaka. We confirm that:

- this work was done wholly or mainly while in candidature for a research degree at this University;
- where any part of this thesis has previously been submitted for a degree or any other qualification at this University or any other institution, this has been clearly stated;
- where we have consulted the published work of others, this is always clearly attributed;
- where we have quoted from the work of others, the source is always given. With the exception of such quotations, this thesis is entirely our own work;
- we have acknowledged all main sources of help;
- this work is accomplished entirely by us and our contributions and extensions from other works are clearly stated.

- a part of this study has been published in a conference (Please see Section 1.3).

Countersigned	Signature
.....	.....
(Dr. Chowdhury Farhan Ahmed)	(Redwan Ahmed Rizvee)
Professor	Exam Roll: Curzon-2301
Dept. of CSE	Registration No: 2014-716-612
University of Dhaka	Session: 2018-19
Supervisor	Semester: Spring, 2019

"Everyone dies alone. But if you mean something to someone, if you help someone or love someone, and even a single person remembers you, then maybe you never die at all."

– Person of Interest

Abstract

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

Acknowledgements

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

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

I also want to express wholehearted thankfulness and gratitude towards my family, friends and well wishers for their support and suggestions. They define who we are, what we are and what we think. This work has been done at a very crucial time when the novel coronavirus was spreading its deadly claws over the complete world by affecting millions of people. This work could not be completed if I had not received the support from my family and friends. A special gratitude will also be conveyed to Md. Fahim Arefin, Shahadat Hossain Shahin, Riddho Ridwanul Haque and Nahian Ashraf for their guidance, valuable suggestions and comments throughout the work. Lastly, I want to thank Department of Computer Science and Engineering, University of Dhaka, its faculty, staff, and all other individuals related to the department for providing us with an opportunity to be a part of this family and facilitating us with a good educational and research environment.

Redwan Ahmed Rizvee

January 26, 2022

Contents

List of Figures

List of Tables

List of Algorithms

Chapter 1

Introduction

Data mining is a research domain which intends to mine interesting knowledge from a knowledge database. It is basically a process of extracting different types of knowledge from a large database. Due to wide range of complexity and variation in problems, the whole data mining domain has been divided to numerous sub domains, such as, pattern mining, classification, clustering, graph mining, etc.

Pattern Mining problem tries to discover different types of interesting patterns from a database. Pattern mining problem was originally defined based on *Itemset Mining* problem using market basket analogy to analyse the common customer purchasing behaviour. For example, suppose, we are in a departmental store, where day to day products are sold. Itemset mining problem may here try to discover the common itemsets (or here daily goods) which can be frequently found in the grocery list of the customers, such as, it can be observed that (egg, bread, butter) is a very common product combination (or itemset) which is often purchased together. Similarly other interesting frequently occurring product combination may also be found, e.g., (potato, onion, green peeper), (rice, pulses), (tea, sugar, lemon), etc. are some of many. Generally such a knowledge database is huge in length and have significant amount of underlying co-occurrence complexity among the elements [2]. So, the itemset mining problem is not very trivial and imposes a significant number of challenges and variation. Thus, a good number

of literature have addressed various aspects related to the problem and tried to devise different sets of strategies [12, 31, 33, 52, 60].

Though itemset mining problem was originally defined from the market basket analogy but eventually has found its usage in numerous applications [20]. Initially, the pattern interestingness was defined based on the idea of frequent occurrences [1, 17, 52] but slowly other types of interesting measures and variations were addressed, such as, weight based [61], utility based [3], uncertainty based [33], dynamicity based [31], constraint based [19], data structure improvement based [32, 53] and many more. So, the pattern mining problem has always found a significant amount of importance to the researchers. Though itemset mining has been found as a very important tool to discover different types of interesting information from a database, it bears a very significant limitation. Itemset mining do not consider the *ordered* co-occurrence relation among the elements of a database which can be quite important and interesting depending on the application specification. So, to solve this challenge a new separate research sub-domain was introduced from the itemset mining named as *Sequential Pattern Mining* (SPM) [51].

1.1 Motivation

The idea of pattern mining problem was introduced to discover interesting characteristics or behavior from the database. Due to the wide variation of database characteristics pattern mining problem has been divided into numerous sub-domains, among which Sequential Pattern Mining or *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 to extract the ordered co-occurrence relation among the elements of the database. Being first introduced in [51], the SPM problem has found its usage in numerous applications, e.g., web usage mining [13], customer behavior analysis [28], DNA sequence mining [29], online collaborative education [54], network traffic analysis [56] etc.

How, SPM problem addresses a different type of crucial challenge compared to itemset mining can be understood using an example based on market basket analogy. Suppose, we are in a camera gadget market which deals with camera related products. Now, if we analyze the customer purchase history, we will find that there are some items or products which are often found in the buying transactions, e.g., camera body, kit lens, 50 mm prime lens, tripod, camera bag, battery, ND filter, etc. But there lies some interesting observation among these elements which is the ordered buying relationship. For example, if a customer C_1 does not have a camera body, he will not vouch for buying lenses (*camera* $\rightarrow$ *lens*). Similarly a beginner level camera user intends to buy comparatively entry level and inexpensive camera lenses (e.g., *kit lens*) compared to an intermediate or experienced user who will opt for more advanced and expensive lenses (e.g., *prime lens*). But after sometime, when the prior beginner level user improves his photography skill he or she then opts for more expensive lenses (*kit lens* $\rightarrow$ *prime lens*). So, there can be some interesting ordered co-occurrence relation among the elements purchased thoughout different transactions.

In Table 1.1, we have shown a small snapshot of a camera market sequential dataset. Here, suppose, sequence s_i denotes the purchase history for customer C_i. Each product (or item) belonging to same open and close brackets denotes a transaction for the corresponding customer. Items belonging to the same transaction denote that they were bought together by the customer and are unordered. Also each transaction maintains order among them, e.g., in sequence $s_1(sid = 1)$ first transaction for the customer C_1 is (*camera*, *kit lens*), second transaction is (*50 mm prime lens*) and third transaction is (*tripod*). A pattern from this database can be $<$(*camera*, *kit lens*)(*tripod*)$>$ which denotes that some customer in i^{th} transaction bought (*camera*, *kit lens*) together and in j^{th} transaction bought a *tripod* where $j \geq (i + 1)$. The support of a pattern is the number of customers which had such transaction history, e.g., here support of the pattern $<$(*camera*, *kit lens*)(*tripod*)$>$ is 3. In the table, sid means sequence identifier ID

TABLE 1.1: Camera Market Sequential 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) >*

or sequence ID. Each sequence s_i ($sid = i$ or sid_i) is the transaction history of customer C_i.

Sequential mining problem tries to extract such information from a sequential or ordered database. We have discussed the basic goal of SPM problem using market basket analogy, but based on the established problem definition it can be extended to other applications as well. Discovering the click order pattern for a user during web browsing can help to improve resource controlling strategies and to suggest more person specific advertisements. Analyzing a network's bandwidth requirement information using time ordered parameter might help to identify possible DOS (Denial of Service) attack patterns and possible resource requirement in the near future. Genome sequence pattern analysis might help to identify new species along with help to perform structural separation among different species. In summary, SPM problem can be used to numerous applications to extract different types of ordered information. Due to having such significant amount of importance and applications, in our study, we have approached the sequential mining problem.

Similar to itemset mining, sequential pattern mining domain has also been a widely recognized research area and has attracted numerous researchers due to its imposed complexities and variations. Some literature have focused on designing new mining techniques to solve the generic frequent occurrence based sequential mining problem [5, 44, 51, 62], some has focused on developing new strategies to prune redundant search [14, 58], some has focused on structural representation to improve mining performance [21, 39], some focused on efficient tackling of dynamic database characteristics [10, 11, 26, 41], some focused on specialized attribute

based sequential problems [30, 59], some on environment specific solutions [16, 47] and many more. An interesting observation regarding sequential mining literature is that its branches have most often established themselves based on the basic solutions which have been designed to solve the generic frequency based sequential pattern mining, such as GSP [51], SPADE [62], PrefixSpan [44], SPAM [5], etc. So, when a new mining or solution approach to solve the generic problem is being proposed, it provides the opportunity to improve the existing solutions along with gives scope to address newer challenges and strategies as well. Thus, it has always been a burning research problem to design newer and more efficient solutions to address the generic mining problem.

Due to having numerous applications, a wide range of literature has addressed SPM problem and provided solutions which can be broadly categorized into two groups: apriori based and pattern growth based. Apriori based approaches follow candidate generation and testing paradigm and pattern growth approaches follow the projected database or database shrink concept with pattern's gradual extension. Pattern growth approaches are significantly faster compared to apriori approach [6]. In traditional itemset mining it has been shown that tree structure alike approaches provide more control over the database which ultimately help improve the mining runtime [6, 32] and incorporate new strategies. But due to the problem complexity and definition, the tree alike structures of the itemset mining were not suitable for SPM problem and a structural solution was yet to be proposed. In Table 1.2, we have shown the unordered simple itemset representation of prior sequential camera market dataset presented in Table 1.1. Such unordered itemset database is used during itemset mining which stores the co-occurrence information of the elements but looses the ordered relationship among them. Here items are stored based on their lexicographic name order. As far to our knowledge, a complete tree-based solution which will represent all the information of a sequential database in a tree structured format and provide an efficient mechanism to solve the generic sequential mining problem has not yet been fully

Table 1.2: Camera Market Itemset Dataset

sid	Itemset
1	{ *50 mm prime lens, camera, kit lens, tripod*}
2	{ *85 mm prime lens, camera, kit lens, tripod*}
3	{*camera, kit lens, tripod*}
4	{*50 mm prime lens, 85 mm prime lens, camera, kit lens*}

discovered. Another important factor between a structured and unstructured solution is that, the prior one provides some structural advantages to faster scan the databases while searching for the patterns which ultimately improves pattern discovery time. Also, due to the structural definition of trees, it helps to impose some additional ancestor-successor node relations which improve the searching performance in this regard. Additionally, a properly designed structural solution gives scope to perform interactive mining [4] which indicates the solution's ability to perform mining multiple times based on users' requests without bringing any modification to the existing base structure.

Generic SPM problem focuses on mining frequent sequential patterns from the static sequential databases [9, 14, 18, 22, 42, 45, 46, 51, 62]. But in real-life applications, most of the time, the database is found not to be static; rather gets increased time to time with more information [40]. With increased database size, patterns' distribution can vary significantly which necessitates the urge to mine again over the updated complete database. But, mining sequential patterns over the complete database is a very costly operation. So re-mining over the updated database again from scratch creates several performance and resource bottleneck. From these motivations the problem of Incremental Sequential Pattern Mining (ISPM) was introduced in [55] to tackle the challenge of re-mining over the complete database rather than focusing more over the efficient handling of the new incremental database or increased part of the database [15, 26, 40, 49].

In Table 1.3, we have shown an incremental version of prior camera market database presented in Table 1.1. In Table 1.3, we can see that only 2 sequences were additionally introduced. One completely new sequence in sid 5 as *Insert*

TABLE 1.3: Camera Market Incremental Sequential Dataset

sid	Iteration 1	Iteration 2
1	*<(camera, kit lens) (50 mm prime lens) (tripod) >*	*<(ND Filter, Reverse Ring)>*
2	*<(camera, kit lens) (85 mm prime lens) (tripod) >*	ϕ
3	*<(camera, kit lens) (tripod) >*	ϕ
4	*<(camera, kit lens) (50 mm prime lens) (85 mm prime lens) >*	ϕ
5	ϕ	*<(camera, kit lens) >*

operation and one appended sequence to the end of existing sid 1 as *Append* operation. Here we have shown two iterations of database modification and ϕ denotes that no sequence was added to this corresponding sid in current iteration. An important point to observe from here that, if the prior database's length is huge and a very small incremental or additional database is added, then it would become efficient if we can track only the affected patterns rather than perform re-mining from scratch.

There are several crucial challenges in ISPM problems due to the problem's nature. For example - handling the modification of existing sequences (*Append*), the addition of completely new appearing sequences ($Insert$), ratio of the incremental database vs existing database, updating the existing data structures, the change in the frequent patterns' distribution or concept drift, choosing appropriate values of the extra introduced parameters to control the candidate buffers [11, 35] etc. So, in summary, several issues exist which control the efficiency, applicability and complexity of the solution to approach the problem of ISPM and thus numerous literature have addressed this problem [11, 35, 36, 38, 47]. In summary, though incremental mining problem is a close advancement over the static mining problem, it holds a set of crucial factors which bear significant amount of importance during designing the incremental algorithm. Also, depending on the base solution sketches, the designed solutions for static mining can also vary in performance and design while addressing the incremental version [10, 11, 40]

and the direct usage of static algorithms will not bring any improvement rather focus on re-mining only. So, it is also an important research problem to assess the performance and modification of a static mining algorithm while approaching the incremental one.

As already mentioned, main goal of sequential mining problem is to discover complex ordered relationship among the elements of the database including ordered relationship among groups of items. So, the concerned problem becomes more challenging when the group size increases (e.g., *camera* $\rightarrow$ *kit lens*) $\rightarrow$ *tripod* $\rightarrow$ *prime lens* $\rightarrow$ *filters* ...) or some uncommon but interesting behaviour is observed (e.g., *camera* $\rightarrow$ *Reverse Ring*). Also, the size of the most of the real-life databases is quite huge. So, one of the core challenges of the whole pattern mining domain is to efficiently reduce the pattern search space and discover the patterns of interest. As a very important research domain under pattern mining, sequential domain also provides a great deal of interest in this regard to design compact and efficient pruning strategies [5, 14, 15, 47]. Importance of pruning strategies can be understood with an example. For this purpose, we will use the sequential database shown in Table 1.1. Suppose, a pattern can be interesting (or frequent) if it has occurred in at least 30% sequences ($4 \times 30\% = 1.2 \approx 2$) or 2 times in the complete database. Now, from the database, we can see that the pattern <*(85 mm prime lens) (tripod)*> has occurred only once (25%) which means that this pattern is not frequent based on the set threshold. Now, if we have to perform complete database scan for the pattern and detect its infrequency it will be a costly unnecessary operation. Pruning strategies are developed in this regard to reduce the number of such unnecessary operations.

Mining performance of a solution largely depends on the pruning strategies it develops and adopts. Because, based on these strategies it reduces its pattern search space which ultimately improves the pattern discovering runtime. So, it has always been an important research problem to design more and more compact pruning strategies in this regard. From this point of view, a good number of

literature have focused on designing some supporting structures and strategies [14, 47]. But, it also imposes a significant amount of challenge on how to efficiently update and control these structures and maintain the strategies in an incremental environment.

1.2 Objectives

Based on the motivations discussed in the prior section, we will highlight the objectives of our study in this section.

The main objective of our work is to design a novel tree-based structure to efficiently capture the complete sequential database in a compact format which will give us more structural advantage to control and handle the manipulation of the database along with provide us the scope to embed newer strategies to reduce the pattern search space. After designing such structure, the main challenge will be to design an efficient mining algorithm which will use the structure to discover the complete the set of interesting patterns. As, we have planned to design a new viewing angle to approach the sequential mining problem, first our goal will be to design a complete solution for the generic frequency based sequential mining problem for the static databases.

After designing a complete solution for the static databases, the main challenge will be to assess the extendability of the base solution to approach different variations of SPM problem. In this regard, we will focus on a very close but important variation over generic sequential mining problem for the static databases which is the incremental sequential mining problem. We will extend our base structure to efficiently capture the sequential database in an incremental environment. Then, we will focus on designing an incremental mining algorithm which will efficiently track those patterns which were affected due to the database modification rather than complete re-mining over the updated database. We will also try to investigate and develop other strategies that are important and closely related to improving the performance of incremental mining.

1.3 Contributions

In the light of the motivations and objectives presented in the earlier discussions, in this section, we will talk about our contributions and investigations.

In this study, we have proposed a novel tree structure *SP-Tree* to represent the sequential database in a structured format and an efficient mining algorithm *Tree-Miner* to mine sequential patterns using the node properties of *SP-Tree*. The advantage of the proposed tree structure is, it provides efficient structured control over the database and pattern space which ultimately leads to a faster generation of the patterns. We have already mentioned that, an important motivation behind designing such a structure was to have the scope to embed newer pruning strategies and to have more control over the manipulation of the database, our designed structure satisfies these goals. Our second proposal for the incremental environment is the direct result of this motivation. In this study, we have proposed a new viewing approach to address the sequential mining problem.

After designing a complete solution for the static sequential databases, we have extended our prior solution to solve the incremental mining problem. In this regard, we have extended our prior SP-Tree structure to *IncSP-Tree* and a new incremental mining algorithm *IncTree-Miner* to approach the ISPM problem which provides an elegant manner and structural advantage to implicitly track the updated database and the patterns which are affected due to the new addition. We have also discussed various flexibility of the proposed solution including a resource constraint version.

We have already shown that there lies numerous factors which hold significant importance to improve the performance of SPM problems. Some of them are, reducing the number of database (DB) scans, making the DB scans faster, reducing the search space by designing compact pruning strategies, early detection of the infrequent patterns during support calculation, etc. Each factor addresses a good number of research challenges. In our proposed structures we have introduced the idea of *next_link* which makes the DB scans significantly faster. Also utilizing

the tree properties we have developed two new pruning strategies: a *Breadth-First Based Support Counting Technique* which helps to detect the infrequent patterns early before calculating complete support and a *Heuristic Strategy* to reduce the search space. It has already been discussed that, mining runtime is significantly dependent on the applied pruning strategies. Alongside proposing some new pruning strategies, we have also adopted existing popular pruning strategies. One of the adopted strategies is, the usage of small grouped co-occurrence information to discover the co-occurrence relation among larger grouped patterns [14, 48]. This stored information states the relationship among the items which guides during pattern extensions and this is calculated over the complete database as a preprocessing step. But it becomes a challenge on how to efficiently update this information for a gradually increasing database. To solve this issue, in this literature, we have proposed a novel structure *Sequence Summarizer* which helps calculate such information efficiently specially in an incremental environment. This novel structure is also helpful to perform the *Append* operation over the existing sequences during incremental mining.

As, with database increment, the support of the patterns gets updated in each iteration, popular literature maintain a tree alike structure to keep the patterns' support [10, 35, 38]. In this study, we further investigate this approach and design a new Bi-directional Projection Pointer Based Frequent Sequential Pattern Tree (*BPFSP-Tree*) which keeps the frequent sequential patterns, their support, and projection pointers using the node references of *IncSP-Tree*. This newly designed pattern storage structure also helps reduce the number of database scans and provides an efficient mechanism to remove the non-frequent patterns which were previously frequent.

We have already shown that, incremental mining problem or ISPM problem bears a good number of crucial factors which are related to its performance. So, a significant amount of literature have addressed different factors related to ISPM problem and proposed different solutions. Among them, many introduced

additional parameters or concepts such as negative border [64], semi-buffer [11], pre-large with upper and lower threshold [35], etc. The main problem of additional parameters is that the solution's performance and complexity largely depends on the appropriate selection of these parameters and their mutual dependency and it is difficult to estimate database characteristics prior. Also, these approaches are severely affected due to concept drift and create resource misuse and bottleneck. Based on these observations, we wanted to reduce the parameters' dependency and stick to the single traditional support threshold parameter. Besides, our approach is a new take to solve the generic SPM problem. So, it is also flexible to other modules. Moreover, our proposed structure stores the complete database in an efficient format. So, it is also able to handle the absence of prior database in stream mining and runtime threshold parameter change and is able to report the complete set of frequent patterns at any time based on users' requests. Due to this characteristic, our proposed structures maintain "build-once-mine-many" property by providing scope to perform interactive mining which means that, we can mine multiple times for different thresholds provided by users without performing any modification to the existing structures.

So based on all the prior discussion, we can summarize the total contributions of this current study as follows -

1. We have proposed two new tree-based solutions, an efficient *Tree-Miner* algorithm based on a novel tree structure *SP-Tree* and an efficient *IncTree-Miner* algorithm based on a novel tree structure *IncSP-Tree* to solve the SPM problem for static and incremental databases respectively. Our proposed solutions are able to mine the complete set of frequent patterns. The designed structures have "build-once-mine-many" property and they are also suitable for interactive mining.

2. Based on the tree properties, we have designed two new pruning strategies: an efficient *Breadth-First Based Support Counting Technique* and a *Heuristic Pruning* Strategy.

3. A novel structure *Sequence Summarizer* is proposed to efficiently calculate and update the small grouped co-occurrence information and helps perform *Append* operation during database increment.

4. A new structure *BPFSP-Tree* is designed based on compact *IncSP-Tree* node references to store the frequent sequences along with projection pointers to reduce the number of tree scans during pattern generation and efficiently remove the infrequent patterns.

5. A Discussion regarding efficiency and effectiveness of our proposed solutions and various issues related to the implementation.

The preliminary version of the current work on static databases has been published in [46].

1.4 Thesis Outline

The rest of the study is organized as follows. Related works are discussed in Chapter 2. We formulate our addressed problem and discuss our proposals in Chapter 3. In Chapter 4, 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 Chapter 5.

Chapter 2

Related Works

In this section, we are going to discuss regarding the literature which are related to our contribution. This chapter will help to understand how existing literature have approached the problems which will guide to understand our contribution.

2.1 Frequent Pattern Mining

The goal of frequent pattern mining is to mine all the patterns or itemsets which satisfy a minimum support threshold in the transaction database. The support or frequency of a pattern is the number of transactions in the database which contain the pattern. The pruning strategy applied in frequent pattern mining is downward closure property which states that if a pattern is infrequent then all of its super pattern is also infrequent. The initial approach to mine the frequent patterns was the *Apriori Algorithm* [1, 2, 50] which was based on level wise candidate generation and test paradigm. It had problems of multiple times database scan and a pool of redundant infrequent candidate generation. Thus why a large number of literature addressed this problem and proposed elegant solutions. Among them *FP-Growth* [23] was the first proposal which designed an algorithm to mine all the frequent patterns without candidate generation. Their solution was based on FP-Tree and it needed only two database scans to mine all the patterns. Due to their contribution,

the idea shifted from level wise candidate generation and test paradigm to direct frequent pattern generation. FP-Array [17] technique was designed to reduce the number of traversals in the FP-Tree and improved the performance to a great extent specially in sparse datasets. To identify the strong support affinity of the frequent patterns a very interesting measure, h-confidence was proposed in [27, 57]. CP-Tree [53] was proposed to mine the frequent patterns in single pass. So, in summary due to huge application and importance a significant amount of literature have addressed this problem and proposed elegant approaches. But the main limitation in traditional frequent pattern mining is that they do not consider the ordered relationship among the items or elements of the database which has found its usage in numerous applications, e.g., web usage mining, market basket customer behaviour analysis, DNA genome pattern sequence, etc. So, from this observation, to discover the sequential information among the elements of the database, a new pattern mining domain was introduced named as *Sequential Pattern Mining* [51].

2.2 Sequential Pattern Mining

In this section, we will discuss the literature related to traditional sequential pattern mining problem. Sequential pattern mining (SPM) problem was first introduced in [51]. SPM problem mainly targets to discover those frequent sequences from a sequential database whose support satisfy a minimum support requirement. Being a very popular problem a good amount of literature have addressed this problem and provided huge variety of solutions.

The proposed solutions to approach SPM problem can broadly be categorized into two groups, *apriori based* and *pattern growth based*. First apriori based solution was GSP [51]. Apriori based solutions follow the candidate generation and testing paradigm. They generate all the patterns in level by level manner. They first discover all the K length frequent patterns and based on it, they generate the candidate set for $K+1$ length frequent patterns, perform frequency test and finally discover $K+1$ length frequent pattern set. Following this procedure,

recursively they generate the complete frequent pattern set. As, apriori solution follow candidate generation and testing paradigm they often generate huge number of unnecessary frequent patterns. They also suffer from the problem of multiple times database scans which is a very costly operation. Apriori based approaches are comparatively slower than pattern growth based approaches.

Pattern growth based approach are more efficient compared to apriori approaches because they use the concept of database's continuous space reduction with the pattern's gradual extension. They discover the patterns not in level by level manner but rather in depth first manner. PrefixSpan [22] is one of the first solutions based on pattern growth approach. It is a very efficient approach to solve the sequential mining problem based on database's continuous projection with pattern extension. Database projection concept provides significant runtime improvement compared to apriori solution. But still, database's actual projection was a costly operation. Because, it had to slice the projected database again and again with pattern's gradual increase. So, to solve this issue, pseudo projection concept was introduced which did not perform the actual database projection rather took a pointer based approach to indicate how much of the database has already been seen or projected. But, it had its own limitation. PrefixSpan and GSP both work on horizontal database format. Performance of PrefixSpan reduced with the increase of sequence length because of scanning the repeating subsequence again and again.

In [62], SPADE algorithm was proposed. It was also based on candidate generation and testing paradigm. But it used a lattice or table alike structure to store the information of sub databases to faster the database scan to calculate the actual support. But this solution have memory issue along with the problem of huge number of infrequent pattern generation in the candidate phase. In [5], SPAM algorithm was proposed. SPAM also used depth first approach to generate the patterns. They introduced the idea of bit based representation to denote pattern occurrence and calculate the support. This ultimately helped to improve

mining time. They also introduced the idea of additional two lists associated with each pattern to guide during their extension phase. SPAM is another very efficient and popular approach to solve the sequential mining problem.

In [48], FAST algorithm was proposed. FAST introduced the idea of sparse id lists as a new data structure to perform faster database scans and calculate the support of the pattern. FAST is another very efficient algorithm to solve sequential mining problem. But, have some memory requirement as it had to maintain the sparse data structures. FAST was designed to improve the idea of SPADE. In [58], LAPIN algorithm was proposed. LAPIN showed how the items of the last event can control the pattern search space. This idea ultimately helped to improve the performance of SPAM and other depth first based approaches similar to it. In [14], the idea of co-occurrence information was shown and how this information can guide during pattern extension was discussed. Based on the established CMAP data structure, they also improved two very popular algorithms, SPADE and SPAM to CM-SPADE and CM-SPAM.

In our current study, we have proposed a new tree-based structure SP-Tree to represent the sequential database along with a new mining algorithm Tree-Miner which uses this tree to mine the complete set of frequent sequential patterns. The main reasoning behind designing a tree structure was, it provides some additional structural advantage to control the database and helps to embed new pruning strategies which ultimately helps to improve the mining performance. In frequent itemset mining, it has already been shown [17, 32] that how tree based structure helps improve mining along with gives the advantage to extend the solution in different directions [7, 31, 34]. It was also another motivation behind designing a tree-based solution. Due to the problem nature, the tree-based solutions of itemset mining can not be used to solve the sequential mining problem. As, sequential mining problem considers both order and multiple itemed single events making the problem comparatively more difficult. In [13] PLWAP tree was proposed to mine the sequential patterns from web click data stream. But this solution could not

handle multiple itemed event occurrences. Our proposed solution is a complete tree-based approach to solve the traditional sequential pattern mining problem.

2.3 Incremental Sequential Pattern Mining

Basic sequential pattern mining problem was designed to mine the frequent sequential patterns from static databases. But, in most of the applications of sequential mining, it is found that the database is not static, rather it grows more and more with time providing new range of challenges. So, a new research problem was addressed as mining frequent sequential patterns from incremental databases [55] to efficiently approach the challenges of a gradually increasing database.

Mining frequent patterns from an incremental database arises good number of research challenges [40, 49]. So, many researchers have addressed numerous aspects and provided their solutions. To solve the incremental mining problem, one of the earliest solutions was given in [55]. They designed a suffix tree-based solution to approach the problem which maintained the substring w.r.t their address rather than positions. The critical performance issues of this approach were the suffix tree's dependency over the database size and the sensitivity of the position where the update occurs. In ISM [43], the authors adopted a vertical mining approach along with maintaining a negative border information to determine the part of the original database which needs to be scanned again. Negative border included those infrequent sequences whose subsequences were frequent. But the size of the negative border creates a severe memory bottleneck by keeping the information of a huge number of unnecessary patterns. ISE [41] algorithm adopted a level-wise apriori approach for mining and reused the information gathered from the previous passes and had the problem of multiple database scans. MFS+ and GSP+ were introduced in [63] to provide a solution for the incremental database. The goal of the MFS algorithms is different from us because they targeted to mine maximal frequent sequences whereas we intend to mine the complete set of frequent sequences with the actual support.

In, [36] IncSP algorithm was proposed. They provided two completely novel ideas for efficient counting and implicit merging and was found to be very efficient but had the problem of level-wise candidate generation and test paradigm because of adopting the apriori approach. In [11], one of the most popular approaches for solving ISPM problem was given, known as IncSpan. It introduced the idea of semi-buffer concept to reduce the database 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 number of complete database scan. We also wanted to reduce the dependency over the parameters. This approach also induced some additional pre-computation.

In PBIncSpan [10], it was shown that IncSpan is not complete because of some wrong conditioning in the algorithm. They corrected the solution and added two new pruning concepts named as width pruning and depth pruning and provided a prefix tree data structure to store the frequent patterns. It was very efficient but still had a good complexity of projecting the database. In ISPBS [38], they proposed a modified version of PBIncSPan. They suggested keeping all the patterns' (frequent and infrequent) information in the tree to reduce the database Scan. The number of patterns in a database can be combinatorially explosive and therefore, keeping all the patterns' information in tree is not practical and the result is far from optimal in the databases of medium to large size. Being motivated by the FP-Tree [17] 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 [24, 25, 35]. This concept added two

new parameters, named as lower threshold and upper threshold. These parameters acted as a buffer, and using them, the FUSP-Tree structure was updated. Pre-large sequences are those sequences that are nearly large but not truly large. These approaches also have a problem similar to buffer like concepts of dependency over multiple parameters, over computation, and concept drift. The algorithm's performance will largely depend on the appropriate selection of the parameters which is difficult to estimate prior and in case of concept drift, for the streaming databases it will cause a good amount of memory misuse and unnecessary pre-computations. Our proposed solution IncTree-miner based on IncSP-Tree enforces more importance over the frequent patterns and their frequency transition properties. As it is based on a single parameter, the solution does not perform unnecessary pre-computation and is not affected due to concept drift. Also, as it holds all the information in a compact manner, it is not affected due to the information absence in stream databases. Researchers also introduced some hybrid and environment based solutions. For example, in [37], the authors provided an apriori and maximal pattern-based approach, in [8], the authors gave a solution to mine closed patterns from the incremental database, in [47] the authors provided a distributed solution based on MapReduce framework, etc. But as our approach discovers all the frequent sequential patterns from the current updated database with their support from a single machine environment, these literature do not match with our key performance issues.

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

Chapter 3

Proposed Methods

In this literature, we have proposed a new tree-based solution to approach the Sequential Pattern Mining (SPM) problem. We have developed a tree-based structure to represent the sequential databases and a set of mining algorithms to mine the frequent patterns from the structure. In our work, we have approached two problems related to SPM. They are -

- Sequential Pattern Mining from Static Databases or *Traditional Sequential Pattern Mining Problem (SPM).*
- Sequential Pattern Mining Problem from Incremental Databases or *Incremental Sequential Pattern Mining Problem (ISPM).*

In the upcoming sections, we will provide the definitions of the approached problems along with their differences and challenges. In this study, initially, we have proposed a new tree-based structure, *SP-Tree* to represent sequential database in a structured format. Then, we have discussed the proposed mining algorithm *Tree-Miner* which mines the patterns from *SP-Tree*. Then, we have proposed an extended structure, *IncSP-Tree* and a new mining algorithm, *IncTree-Miner* to approach the second addressed problem. In our work, we have also developed various new pruning strategies based on the properties of our proposed SP-Tree alike structures. We have also addressed some specific challenges related

to mining in an incremental environment and devised our solutions to tackle them. All the related concepts will be discussed in great detail in the upcoming sections.

3.1 Problem Definition

In this section, we will provide the definitions of the approached problems along with their differences and challenges. But, first we will provide the core definitions related to Sequential Pattern Mining (SPM) problem.

Let, there be a set of items $I = \{i_1, i_2, i_3, .., i_n\}$. An event or itemset e is a set of items such that $e \subseteq I$. A sequence $s = <e_1, e_2, e_3, .., e_n>$ is a collection of ordered events. A sequence Database D consists of sequences where each sequence, s_i, has an unique identifier aka sid ($sid = i$ or sid_i). An event e is contained in another event e' ($e \subseteq e'$) iff all the items within e are found in e'. A sequence $s = <e_1, e_2, e_3, ..., e_m>$ is contained in another sequence $s' = <e'_1, e'_2, e'_3, ..., e'_n>$ if and only if $e_1 \subseteq e'_{i_1}$, $e_2 \subseteq e'_{i_2}$, $e_3 \subseteq e'_{i_3}$,...,$e_m \subseteq e'_{i_m}$ where $1 \leq i_i < i_2 < i_3 < i_4 ... < i_m \leq n$. 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, satisfy a minimum support threshold parameter min_sup set by users.

3.1.1 Sequential Pattern Mining from Static Databases

In Definition 3.1, we have stated the problem. This problem mainly focuses on mining frequent sequential patterns from a static database. Static database means, the database has no update with time. The algorithm will be given a complete database and it has to efficiently discover all the patterns from the complete database which satisfy the minimum support count condition. This definition also represents the traditional or generic sequential pattern mining problem upon which the sequential pattern mining research domain was created. In Table 3.1, we have shown an example of a sequential database. Main challenges of any algorithm related to SPM are, reducing the pattern search space, performing faster

TABLE 3.1: Static Sequential Database

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

database projection, reducing the number of database scans, detecting the infrequent patterns early, etc. Algorithm's efficiency and performance will depend on these factors.

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, S_P satisfy min_sup ($S_P \geq min\_sup \times |D|$).

3.1.2 Sequential Pattern Mining from Incremental Databases

In Definition 3.2, we have stated the problem. This problem mainly focuses on mining the frequent sequential patterns from an incremental sequential database. Incremental database means, the database will not be fixed and will have new addition of data with time. In each iteration, a new incremental database db will be added and our original or old database D will be 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 sids 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 required for a pattern to be frequent gets increased. In each pass, we need to mine those sequential patterns, P which have support, $S_P \geq min\_sup \times |D'|$. This problem is also known as *Incremental Sequential Pattern Mining (ISPM)* problem.

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 support of P, $S_P \geq min\_sup \times |D'|$.

In Table 3.2, we have shown an example of an incremental sequential database. Here we include two iterations. In each iteration, for a row, ϕ means, no sequence was given for this row's sid in current iteration. 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 be 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. The database of Table 3.2 is an incremental version of the database shown in Table 3.1. To understand the difference between static and incremental database, we can observe that, the complete database of Table 3.1 can be found from Table 3.2 after combining all the sequences against their corresponding sids.

TABLE 3.2: Incremental Sequential Database

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

3.2 SP-Tree: A Tree-Based Structure to Represent the Sequential Database

In this section, we will talk about our proposed structure SP-Tree (Sequential Pattern Tree) to represent the sequential database in a tree-based format. SP-Tree provides an efficient mechanism to structurally represent the database in a tree alike structure. In this section, we will discuss the node attributes of SP-Tree and give an example of complete SP-Tree for the sample sequential database shown in Table 3.1. We will also discuss the tree construction and attribute update procedure in the upcoming sections. We will also observe how basic SP-Tree can be extended to approach different kinds of sequential pattern mining problem.

Each row of the table 3.1 represents a sequence. Each sequence consists of event(s) and each event consists of item(s). In each sequence, items within curly braces form events and items within events are lexicographically sorted. Here, $I = \{\mathrm{a}, \mathrm{b}, \mathrm{c}, \mathrm{d}, \mathrm{e}, \mathrm{f}\}$ and I represents the itemset domain of the sequential database.

3.2.1 Node Structure

In this section, we will talk about the node structure of SP-Tree. We have already stated that, items within events must be lexicographically sorted. This sorting

TABLE 3.3: Items with embedded event number

sid	Sequence	Embedded with event number
1	$<(f)(a)(b)(c)(d)(abc)>$	$<(f:1)(a:2)(b:3)(c:4)(d:5)(a:6,b:6,c:6)>$
2	$<(ab)(cde)(a)>$	$<(a:1,b:1)(c:2,d:2,e:2)(a:3)>$
3	$<(abc)(de)>$	$<(a:1,b:1,c:1)(d:2,e:2)>$
4	$<(b)(c)(abc)>$	$<(b:1)(c:2)(a:3,b:3,c:3)>$
5	$<(d)(e)(a)>$	$<(d:1)(e:2)(a:3)>$
6	$<(d)(e)(ab)(c)>$	$<(d:1)(e:2)(a:3,b:3)(c:4)>$
7	$<(ab)(ce)(ab)>$	$<(a:1,b:1)(c:2,e:2)(a:3,b:3)>$
8	$<(b)(c)(a)(cd)>$	$<(b:1)(c:2)(a:3)(c:4,d:4)>$
9	$<(cd)(abc)>$	$<(c:1,d:1)(a:2,b:2,c:2)>$
10	$<(cd)(ef)(ac)>$	$<(c:1,d:1)(e:2,f:2)(a:3,c:3)>$

helps to a great extent in mining. Before diving into extensive discussion, We want to talk about a very important observation which we use in our structure. This observation is stated in Observation 3.2.1. Based on Observation 3.2.1, we can say that, in the first transaction ($sid = 1$) of sample database first a's event number is 2 because it lies in second event and second a's event number 6 because it lies in sixth event. Similarly, in the tenth transaction, first c's event number is 1 and second c's event number is 3. So, embedded with the event number the sequential database shown in table 3.1 can be represented according to the table 3.3.

Observation 3.2.1 (Sequence Observation). Each item of each sequence has a corresponding event number which denotes the number of event where this item belongs to that sequence.

Now, we will talk about the node attributes of the SP-Tree.

1. *item:* 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.

2. *event_no:* Each sequence consists of ordered events. So, in each sequence each item has a specific event number to represent it (Observation 3.2.1). E.g., in sequence $<(abc)(ac)>$, first a's event number is 1 and second a's

event number is 2. To represent this, each node will have an attribute named $event\_no$.

3. *count:* This attribute's value denotes the number of times this node has been traversed during construction of the tree from the raw database. This value also denotes how many times this path (root up to this node) or prefix has been shared among the sequences. This attribute's value directly contribute to the generated pattern's support.

4. *child_node:* Similar to normal tree structures each non leaf node of the tree will have child nodes for $\{item, event\_no\}$ combination.

5. *next_link:* Next links for an item α from a node V denotes the first occurrences for that item in different disjoint subtrees of V. Next link is helpful to reach the nodes faster to generate the patterns. We will have a detailed discussion regarding the advantage of $next\_link$ in the upcoming sections. $next\_link$ is an important part of our proposal.

6. *parent_info:* $parent\_info$ 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 the $parent\_info$ by numbering the items in the database which leads to memory compactness along with operational efficiency. E.g., in sequence $<(ab)(abc)>$, c's (node which represents second event's c) parent info will contain $\{a, b\}$ represented as 11 by numbering $a = 0$ and $b = 1$ and setting 1 in 0^{th} and 1^{st} position.

7. S^{MAL}*:* (*Maximum Achievable Local Support*) This attribute of a node V represents the maximum possible achievable local support for an item α from the underlying subtree. It helps detect the infrequent patterns early and reduces pattern searching cost.

   Now, we will provide a short discussion regarding how we can construct the SP-Tree from our sample sequential database shown in Table 3.1. At

the beginning, we will only have the root node. Then, we will start to put each sequence in the tree and the rest of the nodes will be created. For each sequence we put each item of the sequence of each event in the tree sequentially with their corresponding event number. We always start from the root and recursively put the items in the tree and traverse the tree. For each node, we check if we have child node from the current node for the item (with corresponding event number) we want to put. If we do not have that, then we create a node with the item and event with count 1. But if we already have a child node then we just increase that node's count attribute's value. After creating or increasing the count value of the child node we go there and perform recursive process to put the next item of the sequence into the tree similarly. Based on the above ideology, a pseudo code to insert a sequence into the SP-Tree has been shown in Algorithm 1. The InsertSPTree function is called with root, a sequence s, the first item in the first event ($item\_no = 0, event\_no = 0$) and last event's bitset representation ($event\_bitset$). Similar to earlier discussion, when we already have a node for the concerned $item, ev$ combination, we directly reach there (line 13). Otherwise we create a node for it (lines 9-12). Then we set the current node for $\{item, ev\}$ as the next level node, increase the count attribute (line 13). Finally, we perform further recursion call to insert each remaining item of each event in the SP-Tree.

In Figure 3.1, we have shown the steps after inserting sequences 1-8. Here we have shown when the new nodes are created and when the prefix is shared. In Figure 3.2, we have shown the complete SP-Tree of the sample raw database shown in Table 3.1. For discussion purpose, we have used node number shown in blue color. The $parent\_info$ attribute value of each node is shown in red color. To save the information as $parent\_info$, we used bit based representation by numbering the items and allocating the corresponding bit. In each node, we have shown the following order to show the attributes, $item, event\_no, count$. We

Algorithm 1 Insert sequence into SP-Tree

Input: node N, sequence s, $event\_no$, $item\_no$, $event\_bitset$.
▷ $event\_bitset$: Current Event's bitset represent.
Output: s is inserted into the tree.
procedure InsertSPTree
    if ($event\_no ==$ len(s)) **then Return** N ▷ All the events been inserted
    if ($item\_no >$ len($s[event\_no]$)) **then** ▷ Add next event
        InsertSPTree(N,s,$event\_no + 1$,1,0) ▷ 1 Based Indexing
    $item = s[event\_no][item\_no]$,$ev = event\_no$
    if ($N.child\_node[\{item, ev\}] == \{\}$) **then** ▷ No child found
        ▷ create a child node n and initialize the attributes
        n.item=item, n.event_no=ev,n.count=0,n.parent_info=event_bitset
        $N.child\_node[\{item, ev\}] = n$ ▷ Setting n as child node
    $n = N.child\_node[\{item, ev\}]$,$n.count = n.count + 1$ ▷ Increasing overlapped count
    InsertSPTree(n,s,$event\_no$,$item\_no + 1$, $event\_bitset$ **OR** $item$
        ▷ Set bit for $item$ in event_bitset for next call

have also used orange color value to denote the node position where a sequence ended. For simplicity and to understand the concepts we have assumed that the sequences arrive following the ascending order of sid(s). We have numbered the nodes in such a way so that it depicts the node creation order. To understand how $parent\_info$ attribute works we can see node 8. This node has $parent\_info$ value = $(11)_2$ because it has item a (node 6) and item b (node 7) in it lying in the same event (event 6) as it. Similarly node 13's (item e) $parent\_info$ is $(1100)_2$ because it has item c (node 11) and item d (node 12) with it in the same event. For $parent\_info$, we have used bit based representation but other representations will also work. A node just need to know the items which are in the same event with it found in its ancestor nodes.

In this section, we have discussed how SP-Tree alike structures can be constructed from the raw database. In the upcoming sections, we will discuss the procedure to calculate and update the node attributes along with investigate various important aspects of SP-Tree structures.

3.2.2 *next_link*

In this section, we will talk about $next\_link$ attribute. *next link* is a very important part of our proposal. It plays a very significant role during mining and applying pruning conditions. Formal idea of $next\_link$ is given in Definition 3.3 based

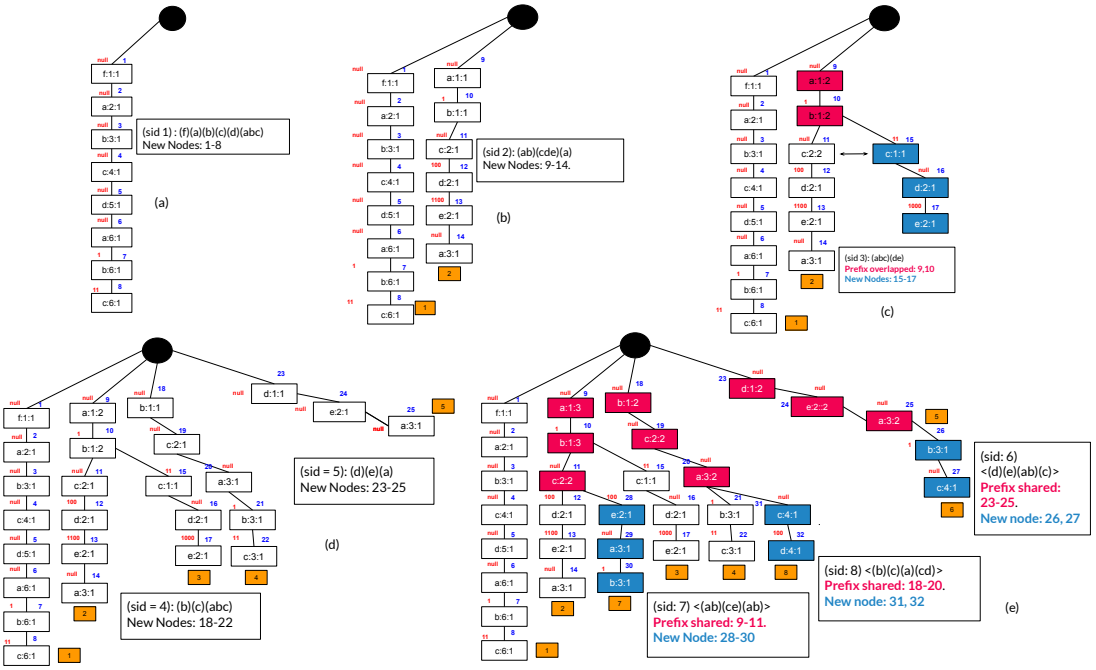


FIGURE 3.1: (a) After Inserting Sequence 1, (b) After Inserting Sequence 2, (c) After Inserting Sequence 3, (d) After Inserting Sequence 4 and 5, (e) After Inserting Sequence 6, 7 and 8

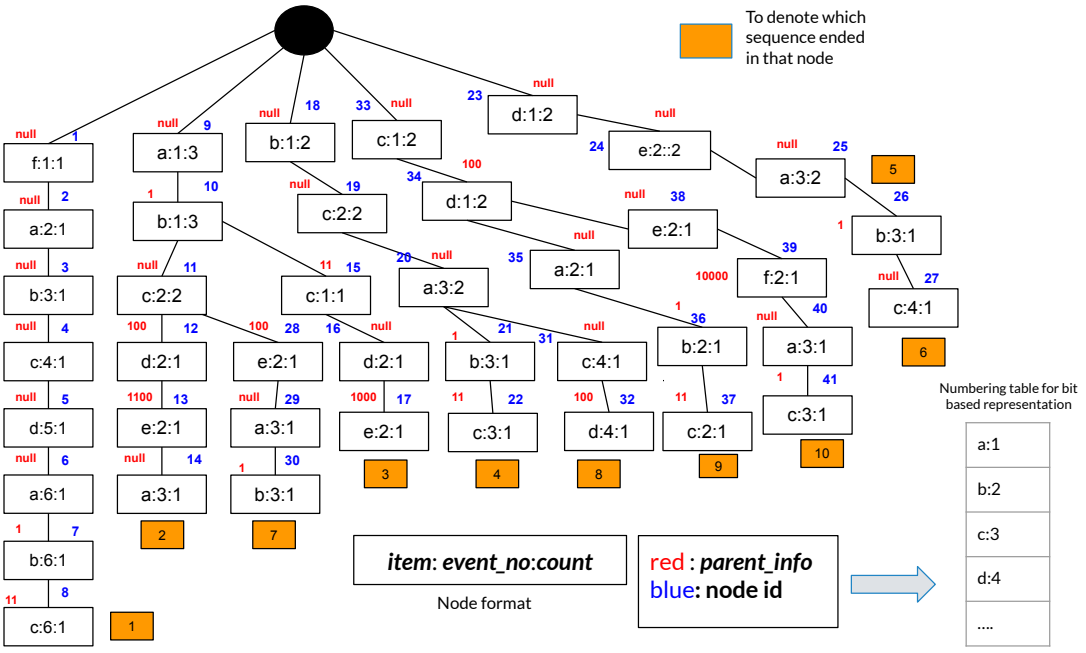


FIGURE 3.2: Complete SP-Tree of Sample Database of Table 3.1

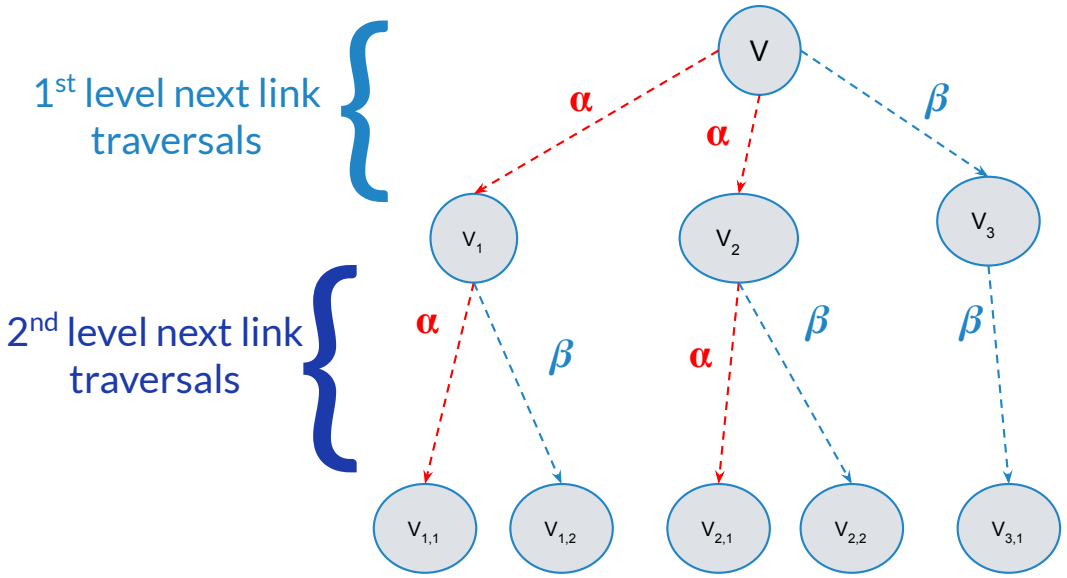


FIGURE 3.3: *next_link* definition

on the Figure 3.3. In the definition, we have used symbol α to explain the concept ($V.next\_link[\alpha] = \{V_1, V_2\}$). The same concept also applies for item β, where from V through *next_link* for β, we can reach node V_3 ($V.next\_link[\beta] = \{V_3\}$).

Definition 3.3 (*next_link*)**.** From a node V, through *next_link* for α we can reach a set of nodes, $v' = \{V_1, V_2\}$ in V's underlying subtrees as first occurrences for symbol α in different disjoint subtrees.

Patterns are generated through combination of nodes. Next link helps to access the nodes faster and efficiently for a symbol or item to generate the patterns. Next link helps to perform faster database traversals resulting in improvement of mining runtime. Nodes directly contribute to the patterns' frequency along with pattern generation.

We have shown an example of next links for some nodes of our SP-Tree of Figure 3.2 in Figure 3.4 to understand how through next links we can perform faster traversal in the database or underlying subtree of a node.

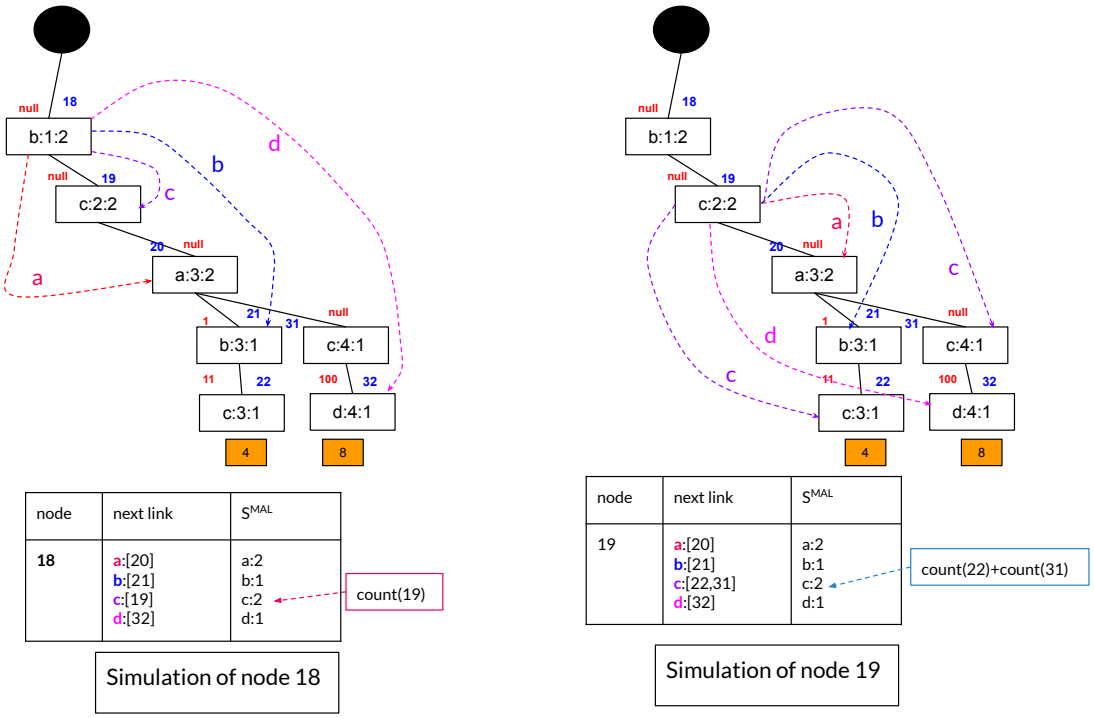


FIGURE 3.4: Example *next_link*s for some nodes

We calculate the next link of each node recursively in bottom up manner. We will discuss about this in the upcoming section.

3.3 Pattern Generation from SP-Tree Structures

In this section, we will discuss how patterns can be generated from SP-Tree structures. Our pattern mining approach follows the pattern growth technique with suffix extension. In the related literature of pattern extension, two types of extensions are maintained. *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 larger compared to the existing items of the concerned event. In summary, we will discussion regarding the following concepts in the upcoming sections -

- **Pattern Formation Rules:** How patterns are generated and their frequency is calculated using SP-Tree node structures.
- **Pattern Extensions:** How two types of aforementioned extensions (sequence and itemset extensions) are performed using our tree structure.

3.3.1 Pattern Formation Rules

The concept behind pattern formation using the node structure of SP-Tree is given formally in Definition 3.4.

Definition 3.4 (Pattern Formation). Based on the Figure 3.5, pattern formation concepts can be pointed as follows -

- Patterns are generated through SP-Tree node concatenations, i.e., $N'_P = \{ (V, V_{1,1}, V_{1,1,1}), (V, V_{1,2}, V_{1,2,2}), (V, V_{2,1}, V_{2,1,1}), (V, V_{3,1}, V_{3,1,1})\}$.
- Each node concatenation $n \in N'_P$ -
  1. Represents a pattern occurrence in a subtree.
  2. Ends in different disjoint subtrees compared to other $n_j \in N'_P$.
  3. Always the first occurrences in different subtrees are considered.

  So, a pattern P can be represented by the nodes where it ends, N_P=$\{ V_{1,1,1}, V_{1,2,2}, V_{2,1,1}, V_{3,1,1}\}$.
- Support of P is $S_P = \sum_{n \in N_P} C(n)$. C denotes the count attribute of each node.
- For each $n \in N_P$, we can say that, we have completed searching up to that node. During following iterations, we will search only in their subtrees.

To explain the idea we are going to use an example. In figure 3.6, we have shown how pattern $<(b)(c)>$ is generated in the tree. We have used yellow color

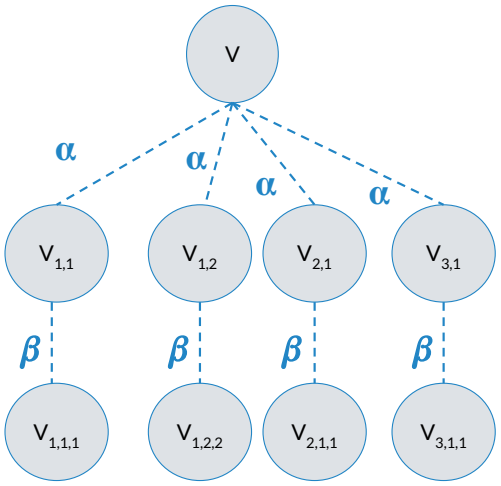


FIGURE 3.5: Pattern Formation

to ease understanding. Combining node 3 and 4 (e.g., (3)(4)) we generate pattern $<(b)(c)>$ in the leftmost branch. Similarly combining (10)(11), (18)(19) and (26)(27), we generate $<(b)(c)>$ in the other branches. As, items under same first bracket form an event in the database, to match this concept we have used similar notation. So, nodes within same first bracket form an event and nodes within different bracket form different events. This is just used for representation and discussion purpose. We will discuss in detail how using nodes' relations we can perform sequence (SE) and itemset extension (IE) in the upcoming sections. We always consider the first occurrence in each disjoint subtree and last node in each occurrence contributes to the pattern's total support (S_P). Here, pattern $<(b)(c)>$'s support ($S_{(b)(c)}$) is 6 (adding node 4, 11, 19 and 27's count value). As, up to the last node (in each contribution), we have completed searching and in the next iterations we will only look into the remaining subtree. We basically need to know just this information that, to construct a pattern, up to which node we have visited. In this example, to construct pattern $<(b)(c)>$, we have visited up to nodes $\{4, 11, 19, 27\}$. So, in the next iterations, when we will extend $<(b)(c)>$, we will look into only node $4, 11, 19$ and 27's underlying subtrees.

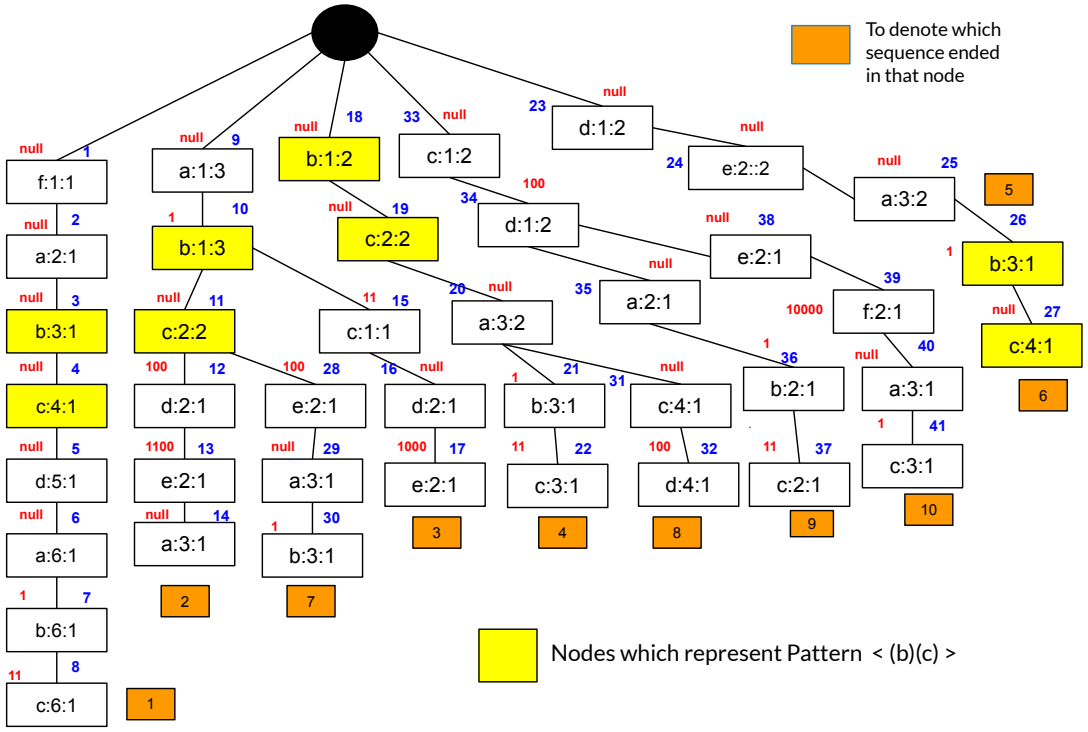


FIGURE 3.6: Pattern Formation of $<(b)(c)>$

3.3.2 Pattern Extension

In this section, we will talk about how using SP-Tree we can perform pattern extension. In the literature related to sequential pattern mining, there are basically two types of pattern extensions. They are -

- **Sequence Extension (*SE*):** Extending a pattern by adding a new event. For example, suppose we have a pattern $<(a)(ab)>$. Then we extend this pattern as sequence extension by adding a new event having item or symbol c. The resultant pattern is, $<(a)(ab)(c)>$.

- **Itemset Extension (*IE*):** Extending a pattern by adding a new item in the last event. For example, suppose, we have a pattern $<(a)(ab)>$. Now, we can extend this pattern as itemset extension by adding a new item c in the last event of the pattern. The resultant pattern will be $<(a)(abc)>$.

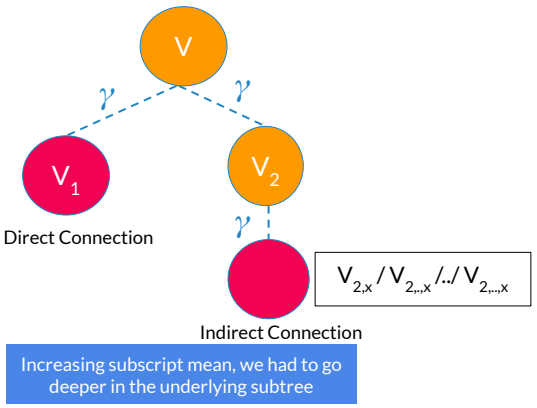


FIGURE 3.7: Pattern Extension Cases

Now, we will discuss in detail regarding the approaches to extend a pattern. First, we will provide some variables upon which we will discuss the pattern extension strategies.

Suppose, we have a pattern $P = <(\alpha\beta)>$ ending at nodes $N_P = \{V_i,V_j,V_k,..,V_n\}$. We want to find nodes $N_{P\gamma}$ which will extend P for γ. Let ev denotes $event\_no$ attribute. For each $V \in N_P$, we follow similar extension procedure (SE/IE) each having two cases. Based on Figure 3.7, we can discuss the extensions as follows -

Definition 3.5 (SE 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 $ev(V) \neq ev(V_1)$, then V_1 can perform SE over V. E.g., $(a) \rightarrow (a)(b)$ with nodes $\{(2) \rightarrow (3)\}$.

2. Indirect Connection: For a node $V \in N_P$ suppose through next link for γ, we reach node V_2. If $ev(V) = ev(V_2)$ then V_2 can not perform SE over V. Then, we need to search in underlying subtree of V_2 to find such node(s) $V_{2,x}$ where $ev(V) \neq ev(V_{2,x})$. In this case, we need to perform two level next link traversals for γ from V. E.g., $(a) \rightarrow (a)(b)$ with nodes $\{(9) \rightarrow (10) \rightarrow (30)\}$.

Definition 3.6 (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 $ev(V) = ev(V_1)$, then V_1 can perform IE over V. E.g., $(a) \rightarrow (ab)$ with nodes $\{(9) \rightarrow (10)\}$.

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

3.3.3 Sequence Extension Examples

In Definition 3.5 we have provided a formal explanation of how sequence extension is performed using our SP-Tree nodes based on two cases with some examples. In this section, we will investigate the concepts in detail.

The idea of sequence extension can be explained using an example. In figure 3.8, we have shown an example of sequence extension where we extend pattern a by adding a new event with item b. Pattern a ends at nodes $\{2, 9, 20, 35, 40, 25\}$. To add b as sequence extension, in node 2's subtree we look for nodes through next link for b and get node 3. As $event\_no(2) \neq event\_no(3)$, it (3) can be added to it (2) as sequence extension to make pattern $(a)(b)$. So $(a)(b)$ can be found by reaching node 3 ($(2)(3) \approx (3)$). This is an example of Direct Connection.

Now, we will provide an example of Indirect Connection. When we look for node in the subtree of node 9 to perform sequence extension for b, through next link first we reach node 10. But as node 9 and 10 are in the same event, they can not be concatenated to perform sequence extension to make pattern $(a)(b)$. So, we perform another move through next link of b from 10 and reach node 30. Node 9 and 30 ($(9)(30) \approx (30)$) can be concatenated to perform sequence extension as they belong to different event. For the Indirect Connection case, we need to

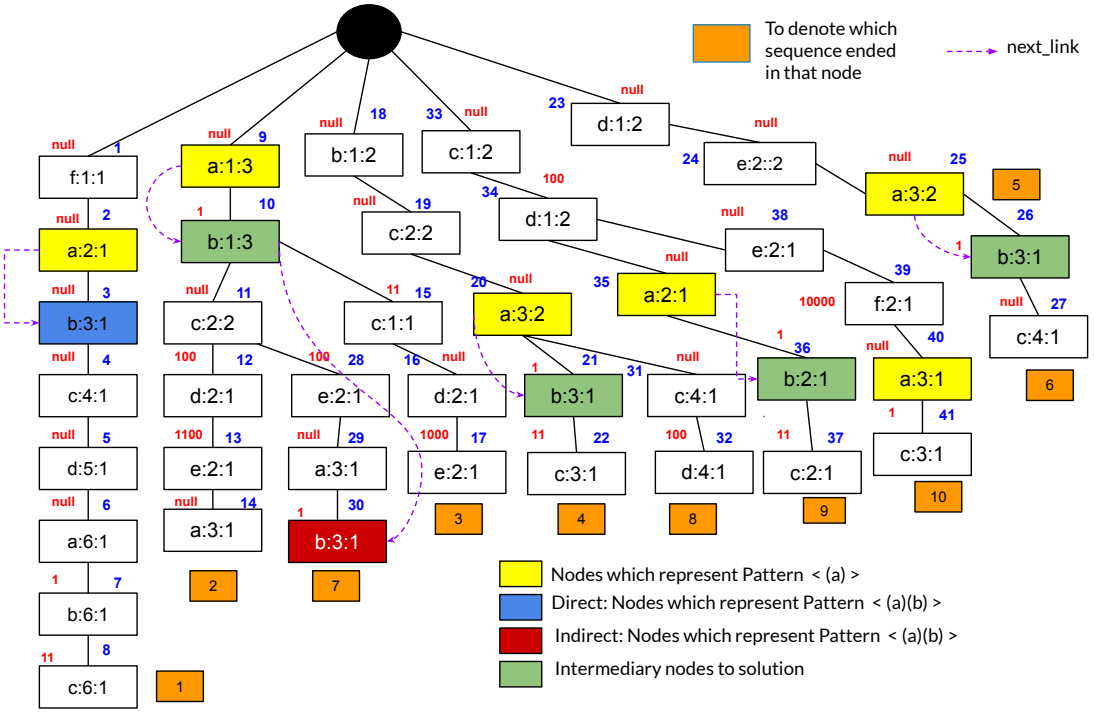


FIGURE 3.8: Sequence Extension ($<(a) \rightarrow (a)(b)>$)

perform two level traversals through next links. In the first traversal, we reach a node having same event number and in the second traversal we reach the solution node having different event number. In our Figure 3.8 , we have used color coding to understand the final extension nodes (direct and indirect cases), intermediary nodes and next link moves.

In the following section, we will discuss how we can perform itemset extension using our SP-Tree.

3.3.4 Itemset Extension Examples

We have provided formal statements to perform itemset extension in our SP-Tree in Definition 3.6. In this section, we will provide some detailed examples to understand the concepts.

In figure 3.9, we have shown an example of itemset extension. Here, we generate pattern (ab) from (a), by adding a new item in the last event. Pattern

(a) ends at nodes $\{2, 9, 20, 35, 40, 25\}$. Now, from node 2, using next link on the leftmost branch we reach node 3. As node 2 and node 3 are not in the same event they can not be concatenated to perform itemset extension here. So, we need to look into the subtree of node 3, to find such node which has label b and an ancestor node with label a in the same event. Through next link for b from node 3, we reach node 7 which has item or symbol a in ancestor node lying in same event as it (node 6). So, node 6 and 7 can be concatenated to perform itemset extension to construct pattern (ab). This is an example of Indirect Connection $((6, 7) \approx (7))$ and pattern (ab) can be found by reaching node 7. Using $parent\_info$ attribute of a node we can get which items or symbols are with it in the same event found in its ancestor nodes. For efficiency issue, $parent\_info$ is stored as bit based representation setting the corresponding bits related to the items and through bitwise AND operation we can check for the items of which we are expecting. Here we wanted to check bit for the 0^{th} position (a is ordered as 1, so for it first bit will be set).

Now, we will show the case of Direct Connection. While processing node 9, using next link we reach node 10. As node 9 and 10 both have same event number, they can be concatenated. So, pattern (ab) can be found by reaching node 10 $((9, 10) \approx (10))$ in this branch. Similar concepts hold for extension from node 20, 35 and 25. From node 40, we can not get any node which satisfy the extension constraints.

3.3.5 Recursive Pattern Generation

We have already discussed how different types of patterns are generated from the SP-Tree by performing sequence and itemset extension. To, generate a complete set of patterns, we perform recursive extension operation on the patterns.

Suppose, we have a pattern P. For each pattern P, we will have two special lists, $sList$ and $iList$. $sList$ contains those items or symbols which may extend pattern P as sequence extension. Similarly, $iList$ contains those symbols which may

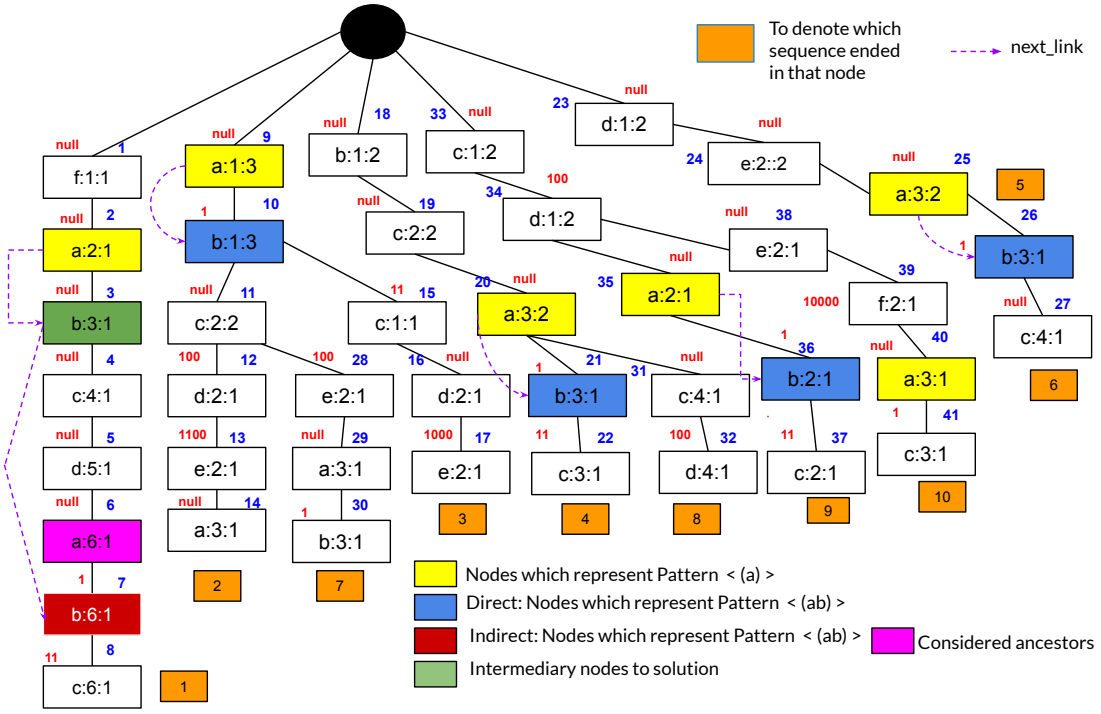


FIGURE 3.9: Itemset Extension ($<(a)> \rightarrow <(ab)>$)

extend P as itemset extension. With pattern extension ($P \rightarrow P' \rightarrow P'' \rightarrow ..$), these two lists will shrink ($sList \subseteq sList' \subseteq sList''..$ and $iList \subseteq iList' \subseteq iList''..$). This happens because larger patterns have lesser frequencies compared to smaller patterns generated from the same prefix. So, with pattern extension, the possible frequent pattern space gets pruned. These two lists actually guide during pattern extension and control the search space. The more compact these two lists are, the more compact the resulting search space will be. So, each pruning mechanism's goal is to reduce these two lists and make them as compact as possible.

In the following sections, we will discuss some pruning approaches which help to make these lists compact along with some mechanism to detect the infrequent patterns early. We have embedded a set of strategies in our proposal to improve the mining runtime. Among them, some are established and popular pruning mechanisms which we have adopted in our solution and some are our own proposed

strategies which we have designed based on our tree alike structure.

3.4 Pruning Mechanisms

In this section, we will discuss the applied pruning techniques which help to detect the redundant patterns early and reduce the search space. Suppose, we have a pattern $P = <(\alpha\beta)>$ with support S ending at nodes $N_P = \{n_1, n_2, n_3, ..., n_k\}$ and corresponding $sList = \{\alpha, \beta, \gamma, \delta\}$ and $iList = \{\gamma, \delta\}$. $sList$ and $iList$ give idea regarding which items may extend P as sequence and itemset extension respectively. Let $\delta = \lceil min\_sup \times |D| \rceil$ which denotes the minimum support count threshold. First, we will discuss the support downward closure.

Lemma 3.4.1 (Support Downward Closure). From any node $n_i \in N_P$, suppose we reach nodes $M = \{m_{1,i}, m_{2,i}, ..., m_{x,i}\}$ through next link for any item ϵ, where each $m \in M$ are in disjoint subtrees and $m_{j,i}$ denotes the j^{th} branch in n_i's subtree. Then $C(n_i) \geq \sum_{j=1}^{x} C(m_{j,i})$. Here C denotes the count attribute's value of each node.

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

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

3.4.1 Co-Existing Item Table Based Pruning

We have already discussed that a pattern can be extended in two ways. By adding a new item in the last event (itemset extension) or by adding a new item as a new event (sequence extension) to the end of the pattern. Co-Existing Item Table (CETable) helps to give an idea regarding which items may extend a pattern as sequence extension ($<(\alpha)> \rightarrow <(\alpha)(\beta)>$) and which items may extend it as itemset extension ($<(\alpha)> \rightarrow <(\alpha\beta)>$). This information ultimately guides during the construction of $sList$ and $iList$ by providing them suggestions to consider only the possible items or symbols rather than considering all of them. This helps

to reduce pattern search space by giving idea regarding two items combinations' ($<(\alpha)(\beta)>$ or $<(\alpha\beta)>$) support value or co-occurrence information over the complete database.

The idea of co-existing item table was adopted from [14]. Here, they pointed out, the relation among the co-occurrences of the items. For example, suppose, we have a pattern $<(\alpha\beta)>$. Now, we want to perform sequence extension on it by adding a new item γ resulting in pattern $<(\alpha\beta)(\gamma)>$. Now, γ will have a chance to be added as sequence extension if and only if γ co-occur with both α and β as sequence extended form and satisfy minimum support threshold. If it does, then we might initiate checking (there also exists other pruning techniques) the actual support count. This information is kept in the co-existing item table.

Similar kind of idea applies for itemset extension also. For example, suppose , we have a pattern $<(\alpha\beta)>$. We want to add γ to this as itemset extension resulting in pattern $<(\alpha\beta\gamma)>$. Now, does γ co-occur in itemset extended form with both α and β or not and satisfy the minimum support threshold or not, this information will be found through co-existing item table. So, in summary, the co-existing item table gives an idea regarding pattern extension by suggesting the possible symbols/items which may satisfy the minimum support threshold resulting in compact generation of both $sList$ and $iList$.

The generated co-existing item table of the sample database 3.1 is shown in table 3.4. From this table we get an idea regarding item co-occurrence along with the chance of satisfying support threshold during extension. This table contains two columns, $CETable_s$ and $CETable_i$. $CETable_s$ shows the co-occurrence condition for each item as sequence extension and $CETable_i$ shows the co-occurrence condition for each item as itemset extension. For example, if we look at the first row, we can say that $<(a)(a)>$ can be found in 3 sequences, $<(a)(b)>$ can be found in 2 sequences and $<(a)(e)>$ can be found in 3 sequences from the column $CETable_s$. Similary from the column $CETable_i$, we can say that $<(ab)>$ can be found in 7 sequences and $<(ac)>$ can be found in 5 sequences. So, from these

TABLE 3.4: Co-Existing Item Table of sample database

Items	Sequence Extending Items ($CETable_s$)	Itemset Extending Items ($CETable_i$)
a	a:3, b:2, c:5,d:4, e:3	b:7,c:5
b	a:5,b:3,c:6, d:4,e:3	c:4
c	a:7,b:4,c:5, d:3,e:2,f:1	d:4,e:2
d	a:6, b:3, c:4, e:3, f:1	e:2
e	a:5,b:2,c:2	f:1
f	a:2,b:1,c:2, d:1	

information we will be able to prune *sList* and *iList*. Formally the definition of this pruning logic is stated in Lemma 3.4.2.

Lemma 3.4.2 (CETable Based Pruning). An item γ can extend P as SE if and only if $CETable_s[\alpha][\gamma] \geq \delta$ and $CETable_s[\beta][\gamma] \geq \delta$. Similarly $\gamma(\gamma{>}\beta{>}\alpha)$ can extend P as IE if and only if $CETable_i[\alpha][\gamma] \geq \delta$ 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 pattern also needs to satisfy. This was adopted in our solution from [14]. □

Main motivation behind Lemma 3.4.2 is *Apriori* property which states that, for a super pattern P' to be frequent, all of its sub patterns P ($P \subset P'$) need to be frequent.

3.4.2 Breadth-First Based Support Counting Technique

In this section, we will discuss our proposed support counting technique. This technique is designed based on SP-Tree node properties. Using this technique, we perform node extension aka pattern extension during recursive mining. This technique is stated in Algorithm 2. We have used comment to understand the underlying logic behind the statements. Now, we will discuss the complete workflow of the Algorithm 2.

Suppose, we have a pattern P ending at nodes N_P and we want to extend P for γ. Let, the support of P is S which also denotes the maximum possible support for the extended pattern $P\gamma$. With gradual extension, we will reduce this support and find the actual support for $P\gamma$. In line 2, we initialize the variables, $N_{P\gamma}$ will contain the nodes which will perform the extension, S' will denote the actual support for $P\gamma$ and Q is a queue (First in First out) data structure which will be used to perform the breadth-first traversal initially containing the nodes $n \in N$.

We gradually process each node $n_i \in Q$ (line 3). First, using S^{MAL} attribute, we check for the maximum achievable support from n_i's underlying subtree for γ (line 4). This heuristic support helps to reduce searching in underlying subtree for an infrequent pattern resulting in improved mining runtime. Now, we remove n_i from Q, subtract the redundant support (line 5) and consider its underlying subtree to find the extension nodes (line 6). Using $next\_link$ for γ we reach a set of nodes $n_i.next\_link[\gamma] = \{m_1, m_2, m_3, ..\}$ (line 6) and for each $m_j \in n_i.next\_link[\gamma]$ we check the pattern extension conditions (sequence or itemset extension) in line 8. The nodes m_j for which the extension condition is satisfied we stop searching in that subtree and put the extension node in $N_{P\gamma}$ (line 8) otherwise we add m_j in Q and continue searching in its underlying subtree (line 9) with the current updated maximum possible support (line 7) for the extended pattern. If any time the maximum possible support falls under the minimum support count threshold we stop searching (line 10) and identify the pattern as infrequent. Otherwise we discover all the valid extension nodes, $N_{P\gamma}$ and return it with the actual support S' (line 11). In Figure 3.10 (b) we have shown a visualization of level by level traversal using a queue data structure to discover the extension nodes.

This technique is valid because of the Lemma 3.4.1. When we remove a node and go deeper in the subtree the maximum possible support for the extended pattern will stay same or reduce. Suppose, we have to check extension from $X = \{U, V, W\}$ for γ with maximum possible support S. We remove V and add

Algorithm 2 Breadth-First Based Support Counting Technique

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

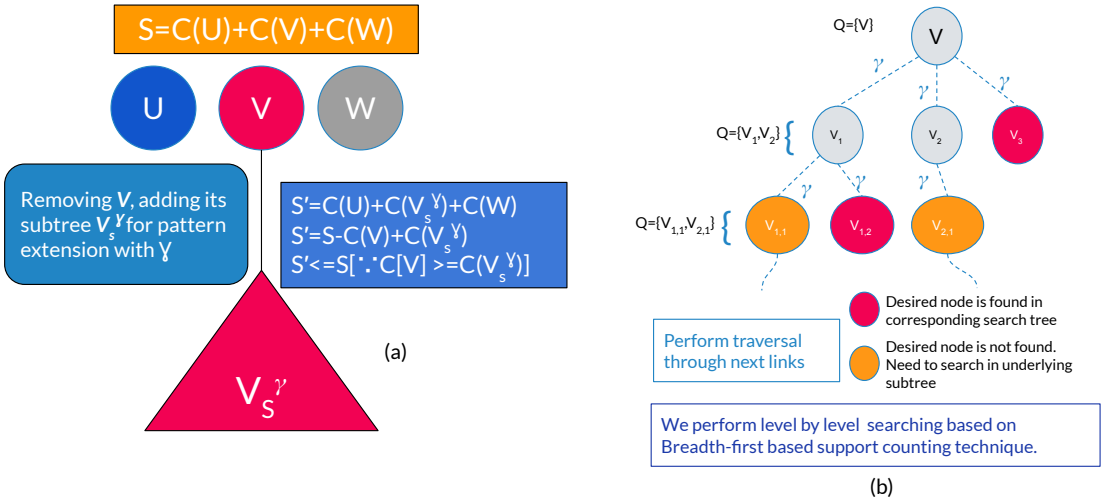


FIGURE 3.10: (a) Breadth-First Based Support Counting Technique (b) Breadth-First Traversal

its disjoint subtree V_s^γ with updated max support $S' = S - C(V) + \sum C(V_s^\gamma)$. As $C(V) \geq \sum C(V_s^\gamma)$, so $S' \leq S$. Visually it is shown in Fig. 3.10 (a). The main advantage of this technique is that we may not need to see the whole projection rather we can very early understand regarding the maximum possible support and decide to prune the pattern.

In Figure 3.11, we have shown a simulation of Breadth-First based Support Counting Technique for pattern $<(a)(b)>$ where we show how we try to discover the final projection nodes in level by level manner and update the maximum possible runtime support for the extended pattern. Our proposed technique directly stops from further projection if it can detect that the pattern's actual support will always be less than the minimum support count threshold. Here δ denotes the

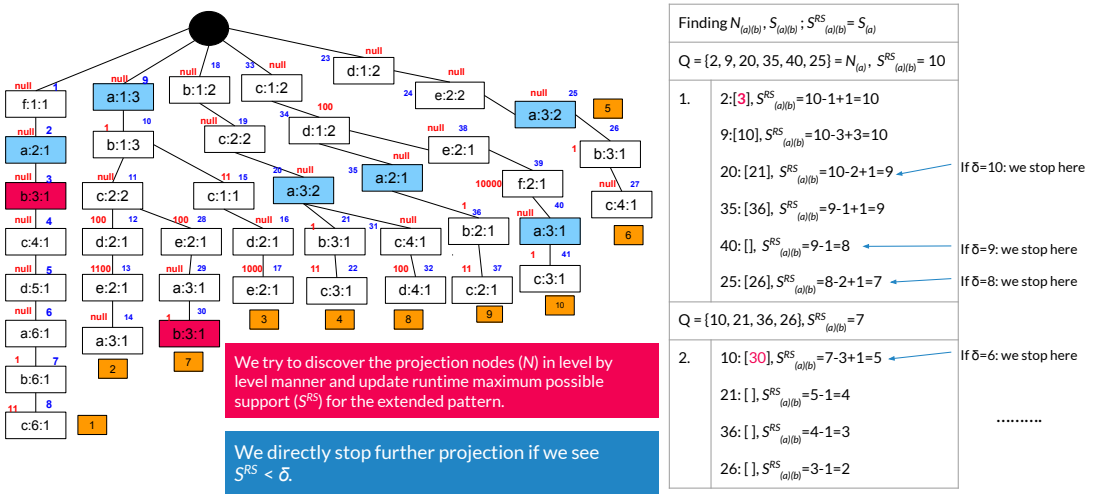


FIGURE 3.11: Example of Breadth-First Based Support Counting Technique for pattern $<(a)(b)>$

minimum support count threshold.

3.4.3 Recursive *sList* and *iList* Pruning

We have already discussed that, with each pattern P we have two lists, $sList$ and $iList$ which denotes that which items or symbols can perform sequence and itemset extension on it respectively. With pattern extension these two lists may shrink because of larger patterns having smaller frequencies compare to smaller patterns with same prefix. Now, based on the mutual dependencies of these two lists a pruning can be performed on them during recursive iterations.

Suppose, for the pattern $P = <(\alpha\beta)>$, initially the two lists were, $sList = \{\alpha, \beta, \gamma, \delta, \psi\}$ and $iList = \{\gamma, \delta, \psi\}$. After support calculation, we found the new pruned or shrinked lists are $sList' = \{\alpha, \beta, \gamma\}, iList' = \{\gamma, \psi\}$. So, for the pattern to be sequentially extended as $P\{\theta\}$ for each $\theta \in sList'$, their corresponding $sList$ will be $sList'$ and $iList$ will be $\{\epsilon | \epsilon \in sList' \cap \epsilon >_{order} \theta\}$ respectively. This observation will work, because, if $<(\alpha\beta)(\delta)>$ is not frequent then definitely $<(\alpha\beta)(\alpha)(\delta)>$ will not be frequent. So, the $sList$ of $<(\alpha\beta)(\alpha)>$ does not need

to contain δ. Similarly if $<(\alpha\beta)(\delta)>$ is not frequent, then definitely $<(\alpha\beta)(\alpha\delta)>$ will not be frequent. So, δ does not need to be in the $iList$ of $<(\alpha\beta)(\alpha)>$.

Now, if we consider the itemset extension for $P = <(\alpha\beta)>$, we will encounter sort of similar phenomena. If we extend P as itemset extension for each symbol $\theta \in iList'$ resulting in $\{P\theta\}$ (meaning θ is added to the last event of P), the new $sList$ for $\{P\theta\}$ will be $sList'$ and the new $iList$ will be $\{\epsilon | \epsilon \in iList' \cap \epsilon >_{order} \theta\}$. For example, the new $sList$ and $iList$ for itemset extended pattern $<(\alpha\beta\gamma)>$ will be $\{\alpha, \beta, \gamma\}$ and $\{\psi\}$ respectively. The idea is same as before, like if $<(\alpha\beta)(\delta)>$ is not frequent then definitely $<(\alpha\beta\gamma)(\delta)>$ will not be frequent, So, δ will not be in included in the corresponding $sList$ of $<(\alpha\beta\gamma)>$. Also, as $<(\alpha\beta\delta)>$ is not frequent, then definitely $<(\alpha\beta\gamma\delta)>$ will not be frequent and δ will not be included in the corresponding $iList$. This is a very popular and renowned pruning technique addressed in numerous related literature which we have adopted in our solution.

3.4.4 Heuristic $iList$ pruning

We have proposed this pruning technique in our literature. The idea of this pruning mechanism is formally stated in Lemma 3.4.3.

Lemma 3.4.3 (Heuristic $iList$ Pruning). Suppose, We have an item $\epsilon (\epsilon \in sList \cap iList)$. During support calculation as SE for ϵ, we collectively reach nodes $M = \{m_{1,1}, m_{2,1}, .., m_{1,2}, m_{2,2}, .., m_{1,x}, .., m_{2,y}, .., m_{k,z}\}$ through single next link traversals for ϵ in disjoint subtrees for each $n \in N_P$. Let $H_{P\{\epsilon\}} = \sum_{m \in M} C(m)$. If heuristic support, $H_{P\{\epsilon\}} < \delta$, then ϵ can not pefrorm IE over P and can be removed from $iList$.

Proof. Intuition lies behind Lemma 3.4.1. 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. This pruning is proposed in our literature. □

Based on the Lemma discussed above, we can interpret the Figure 3.12 to understand the pruning strategy where we have shown that how first level nodes

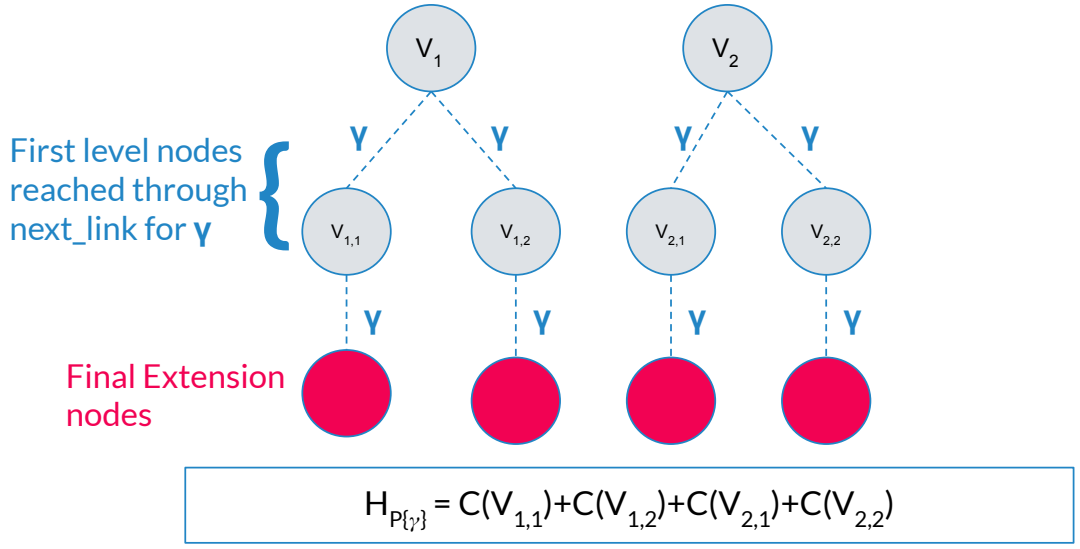


FIGURE 3.12: Heuristic *iList* Pruning

are considered to calculate the heuristic support based on which we decide to prune the *iList*.

In Figure 3.13, we have shown an example of heuristic iList Pruning for pattern $<(a)(b)>$. Here we have shown the first level nodes reached through *next_link* for b from the projection nodes of $<(a)>$ ($N_{(a)}$). We have also shown the calculated heuristic support, $H_{(a)(b)}$ which will always be greater or equal to $S_{(a)(b)}(= 2)$ and $S_{(ab)}(= 7)$. Based on the heuristic support we may directly prune an infrequent item from *iList* without any checking.

3.4.5 Loop Reduction

This pruning technique helps us to reduce searching in a node's underlying subtree by providing a heuristic idea regarding the maximum achievable support from the subtree for an extended pattern. Formally this concept is stated in Lemma 3.4.4. The usage of this pruning strategy has been shown in the pseudo-code of Breadth-First Support Counting Technique in Algorithm 2. We have proposed this pruning strategy in our study using SP-Tree node properties.

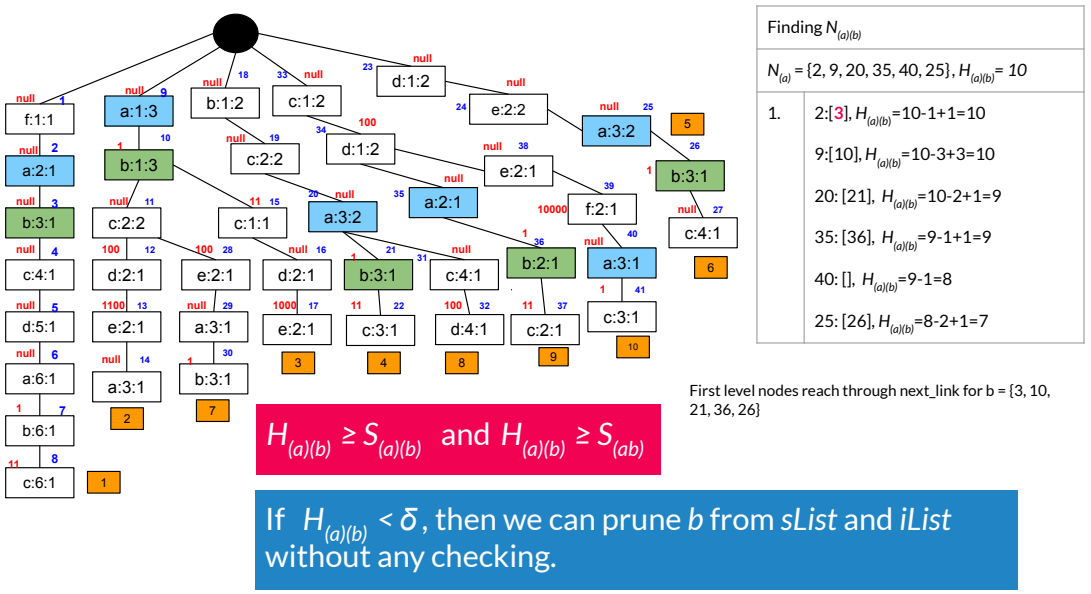


FIGURE 3.13: Example of Heuristic *iList* Pruning for pattern $<(a)(b)>$

Lemma 3.4.4 (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. If, $(S - C(n) + n.S^{MAL}[\epsilon])<\delta$, then $P\epsilon$ can be detected as infrequent and we can stop further checking in subtrees.

Proof. Suppose, from n, we can reach nodes $M = \{m_1, m_2, m_3, .., m_p\}$ for ϵ through single next link traversals and $n.S^{MAL}[\epsilon] = \sum_{m \in M} C(m)$. Also, suppose $M' = \{m_a, m_b, m_c, .., m_q\}$ are valid extension nodes for ϵ from n in underlying subtrees and $\sum_{m' \in M'} C(m') \leq n.S^{MAL}[\epsilon]$ because of Lemma 3.4.1. □

Suppose a pattern P ends in a node n in a branch. And we want to extend P for γ in the underlying subtree. Let the support of $P\{\gamma\}$ (SE for P with γ) and $\{P\gamma\}$ (IE for P with γ) is $S_{P\gamma}$ and $S_{\{P\gamma\}}$ respectively. Then always $n.S^{MAL}[\gamma] \geq S_{P\{\gamma\}}$ and $n.S^{MAL}[\gamma] \geq S_{\{P\gamma\}}$ will hold. Because S^{MAL} stores the summation of the count attribute of first level nodes reached through next link where as to find the actual extension nodes we may need to go deeper in the subtree which denotes

that the actual support may be less than the support stored in S^{MAL}. So, S^{MAL} denotes the maximum achievable local support for the extended pattern.

3.5 Tree-Miner: Complete Algorithm

Now, based on the previous discussed concepts we will talk about our proposed mining algorithm *Tree-Miner* based on *SP-Tree*. Our mining approach is based on forward mining with suffix extension. We start with a single item pattern P with the information in which nodes N_P it ends along with two supporting lists, $sList$ and $iList$, which give idea regarding the items which might perform SE and IE over it. To extend P for an item γ, we find the desired nodes for each $n \in N_P$ in the underlying subtrees and recursively perform the extensions with the shrunken $sList$ and $iList$ through our proposed support counting technique and pruning strategies. We have provided the pseudo-code for Tree-Miner in Algorithm 3. Here, S_P and H_P denotes a pattern P's actual and heuristic support respectively.

Algorithm 3 Tree-Miner

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

In line 4, we perform CETable based pruning based on the discussion of section 3.4.1. In line 5, we initialize the variables to save the extended patterns' projection information (N_{SE} and N_{IE}). In lines (6-10), we perform sequence extension for each item $\gamma \in sList$. First, we perform projection for each $n \in N_P$ for

γ through next link. Our projection and counting approach always works based on the proposed Breadth-First based technique. Then based on the calculated support ($S_{P\{\gamma\}}$) we check for the frequency of the pattern (line 8). As, our proposed support counting technique can early detect a pattern's infrequency. So, it may not perform the complete projection and return very early with an approximate infrequent support. But still based on the returned support, we can take the decision of infrequency. Now, if $S_{P\{\gamma\}}$ is less than the minimum support requirement (δ), then we can say that $P\{\gamma\}$ is an infrequent pattern and thus why we remove γ from $sList$. We do not need to perform any extension from $P\{\gamma\}$. In line 9, we perform heuristic pruning based on the discussion stated in section 3.4.4, where we consider the first level nodes reached through next link for γ, calculate the support and check against δ. If the constraint fails then, we can simply remove γ from $iList$, because actual support of $\{P\gamma\}$ (itemset extension of P with γ) will always be lesser or equal to this heuristic support $H_{P\{\gamma\}}$. But, if $S_{P\{\gamma\}}$ satisfies the constraint then we store the projection nodes in $N_{SE}[\gamma]$ for further extension. We follow similar logic in lines 11-14 for itemset extension. First we perform projection and calculate support, $S_{\{P\gamma\}}$ (line 12), then we perform the frequency check (line 13). If the pattern is infrequent then we remove the symbol from $iList$ (line 13), otherwise we store it in $N_{IE}[\gamma]$ (line 14). In lines (15-16) and lines (17-18) we perform recursive sequence and itemset extension respectively.

In Table 3.5, we have shown a small simulation of Tree-Miner to understand how it discovers the projection nodes for the extended patterns. For simplicity, we have shown the frequent patterns with prefix $<(a)>$ and considered the minimum support count threshold as $\delta = 2$. In the first column we have shown the extended pattern (P), then its support (S_P), then its corresponding $sList$ and $iList$ and finally the shrinked $sList'$ and $iList'$ which is found through applying different set of pruning strategies and which extend the pattern P.

TABLE 3.5: Simulation of Tree-Miner for the patterns with prefix $<(a)>$ and $\delta = 2$

P	S_P	N_P	$sList$,$iList$	After Pruning & Support Calculation
$<(a)>$	10	$\{2, 9, 20, 35, 40, 25\}$	$sList = \{a, b, c, d, e\}$ $iList = \{b, c, d, e\}$	$sList'$={a, b ,c, d, e} $iList' = \{b, c\}$
$<(a)(a)>$	3	$\{6, 14, 29\}$	$sList = \{a, b, c, d, e\}$ $iList = \{b, c, d, e\}$	$sList'$={} $iList' = \{b\}$
$<(a)(ab)>$	2	$\{7, 30\}$	$sList = \{\}$ $iList = \{\}$	$sList'$={} $iList' = \{\}$
$<(a)(b)>$	2	$\{3, 30\}$	$sList = \{a, b, c, d, e\}$ $iList = \{c, d, e\}$	$sList'$={} $iList' = \{\}$
$<(a)(c)>$	4	$\{4, 11, 31, 27\}$	$sList = \{a, b, c, d, e\}$ $iList = \{d, e\}$	$sList' = \{a, b\}$ $iList' = \{d, e\}$
$<(a)(c)(a)>$	3	$\{6, 14, 29\}$	$sList = \{a, b\}$ $iList = \{b\}$	$sList' = \{\}$ $iList' = \{b\}$
$<(a)(c)(ab)>$	2	$\{7, 30\}$	$sList = \{\}$ $iList = \{\}$	$sList' = \{\}$ $iList' = \{\}$
$<(a)(c)(b)>$	2	$\{7, 30\}$	$sList = \{a, b\}$ $iList = \{\}$	$sList' = \{\}$ $iList' = \{\}$
$<(a)(cd)>$	2	$\{12, 32\}$	$sList = \{a, b\}$ $iList = \{e\}$	$sList' = \{\}$ $iList' = \{\}$
$<(a)(ce)>$	2	$\{13, 28\}$	$sList = \{a, b\}$ $iList = \{\}$	$sList' = \{a\}$ $iList' = \{\}$
$<(a)(ce)(a)>$	2	$\{14, 29\}$	$sList = \{a\}$ $iList = \{\}$	$sList' = \{\}$ $iList' = \{\}$
$<(a)(d)>$	3	$\{5, 12, 16, 32\}$	$sList = \{a, b, c, d, e\}$ $iList = \{e\}$	$sList' = \{a\}$ $iList' = \{e\}$

$<(a)(d)(a)>$	2	$\{6, 14\}$	$sList = \{a\}$ $iList = \{\}$	$sList' = \{\}$ $iList' = \{\}$
$<(a)(de)>$	2	$\{13, 17\}$	$sList = \{a\}$ $iList = \{\}$	$sList' = \{\}$ $iList' = \{\}$
$<(a)(e)>$	3	$\{13, 28, 17\}$	$sList = \{a, b, c, d, e\}$ $iList = \{\}$	$sList' = \{a\}$ $iList' = \{\}$
$<(a)(e)(a)>$	3	$\{14, 29\}$	$sList = \{a\}$ $iList = \{\}$	$sList' = \{\}$ $iList' = \{\}$
$<(ab)>$	7	$\{7, 10, 21,$ $36, 26\}$	$sList = \{a, b, c, d, e\}$ $iList = \{c\}$	$sList' = \{a, c, d, e\}$ $iList' = \{\}$
$<(ab)(a)>$	2	$\{14, 29\}$	$sList = \{a, c, d, e\}$ $iList = \{c, d, e\}$	$sList' = \{\}$ $iList' = \{\}$
$<(ab)(c)>$	3	$\{11, 27\}$	$sList = \{a, c, d, e\}$ $iList = \{d, e\}$	$sList' = \{a\}$ $iList' = \{e\}$
$<(ab)(c)(a)>$	2	$\{14, 29\}$	$sList = \{a\}$ $iList = \{\}$	$sList' = \{\}$ $iList' = \{\}$
$<(ab)(ce)>$	2	$\{13, 28\}$	$sList = \{a\}$ $iList = \{\}$	$sList' = \{a\}$ $iList' = \{\}$
$<(ab)(d)>$	2	$\{12, 16\}$	$sList = \{a, c, d, e\}$ $iList = \{e\}$	$sList' = \{\}$ $iList' = \{e\}$
$<(ab)(de)>$	2	$\{13, 17\}$	$sList = \{\}$ $iList = \{\}$	$sList' = \{\}$ $iList' = \{\}$
$<(abc)>$	4	$\{8, 15, 22,$ $37\}$	$sList = \{a, c, d, e\}$ $iList = \{\}$	$sList' = \{\}$ $iList' = \{\}$
$<(ac)>$	5	$\{8, 15, 22,$ $37, 41\}$	$sList = \{a, b, c, d, e\}$ $iList = \{\}$	$sList' = \{\}$ $iList' = \{\}$

In Figure 3.14, we have shown a small visualization of how we recursively

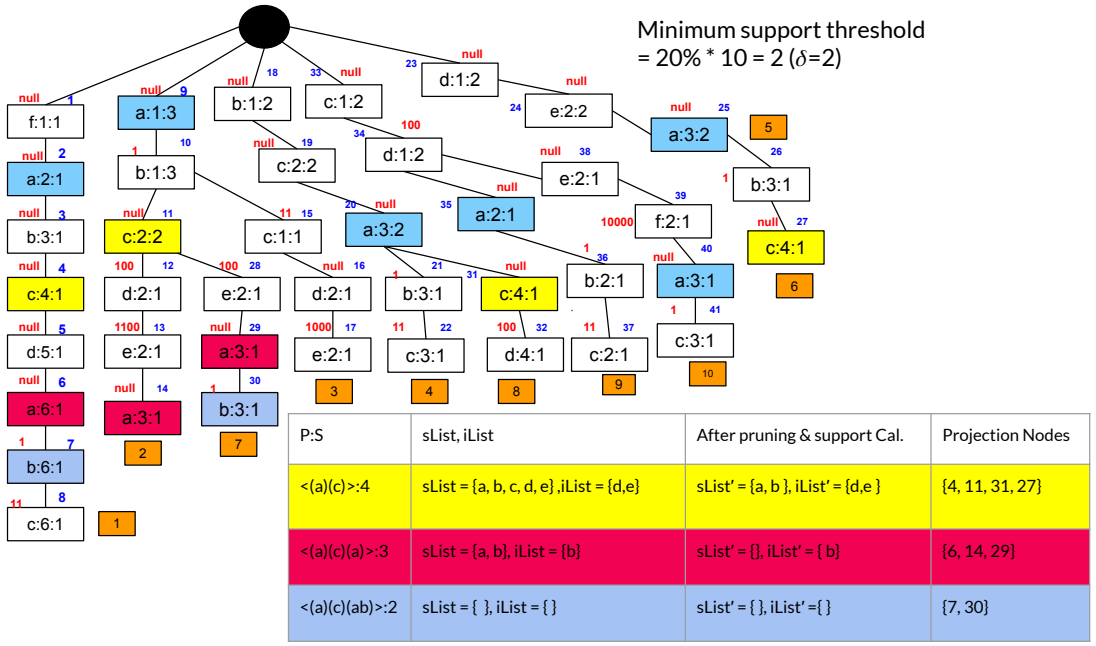


FIGURE 3.14: A Small Visualization of Tree-Miner for Patterns $<(a)> \rightarrow <(a)(c)> \rightarrow <(a)(c)(a))> \rightarrow <(a)(c)(ab))>$

progress and discover the projection nodes for the extended pattern. In this figure, first we find the projection nodes for $<(a)>$, then for $<(a)(c)>$, then for $<(a)(c)(a)>$ and then for $<(a)(c)(ab)>$ recursively. We have used color code to understand the desired projection nodes for the corresponding pattern. We perform faster traversal through next link and use our proposed breadth-first based support counting technique to calculate the support of the patterns.

Up to now, we have discussed our proposal for static databases. We have discussed the proposed tree-based structure, SP-Tree and the designed mining algorithm Tree-Miner to mine the complete set of patterns from the database. But before diving into the discussion of incremental solution, we will talk about an important common supporting structure which is used by both static and incremental solution.

TABLE 3.6: 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 $\alpha \in s$ $\alpha : (bitset_\alpha^{previous}, bitset_\alpha^{current})$

3.6 Sequence Summarizer

In this section we will talk about our proposed supporting structure named Sequence Summarizer which is designed to summarize the information regarding each sequence. We have already shown that co-occurrence information [14, 15] is helpful to guide during pattern extensions and to store this information, we maintain a table named Co-Existing Item Table (CETable) in our solution. It is an important challenge how to efficiently calculate this information. Also, this becomes more challenging when we encounter an incremental environment, e.g., same co-occurrence information (like $(\alpha)(\beta)$ or $(\alpha\beta)$) can not be calculated multiple times for the same sequence and efficiently update the CETable based on the newly appended sequence rather than performing complete calculation again over the full sequence. Sequence summarizer is designed to solve this issue. Also, this structure is helpful to improve performance during *Append* operation. We will provide some more insights regarding *Sequence Summarizer*'s usefulness in an incremental environment in the upcoming section. Here, mainly we will shed light over the definition of the structure and the basic technique to update the CETable for a sequence using Sequence Summarizer.

The theoretical definition of *Sequence Summarizer* for each sequence is given in Table 3.6. The definition is made flexible to directly handle the incremental environment. First, we will provide the flexible definition supporting incremental environment. Then, we will show how it will work in a static environment.

- s_{sid} - The corresponding entry of Sequence Summarizer for the sequence s having sid s_{sid}.

- *last node* - Denotes the last node reference where the sequence s ($sid = s_{sid}$) ended in the SP-Tree structures.

- *positional info* - Here lies three variables for each unique item α belong to the sequence s ($sid = s_{sid}$): s_{α}^{st}, $s_{\alpha}^{en\_previous}$ and $s_{\alpha}^{en\_current}$. s_{α}^{st} denotes the first event where item α was found in s, $s_{\alpha}^{en\_previous}$ and $s_{\alpha}^{en\_current}$ denote the last event where α was found in the previous and current iteration respectively in this sequence s.

- *same event info* - 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. Similar to positional info, $bitset_{\alpha}^{previous}$ and $bitset_{\alpha}^{current}$ denote the information for the previous and current iteration respectively.

Now, we will discuss with some examples to understand how the above definition works in a static environment.

- *last node* - In a static solution, as the database does not get incremented, thus why this information does not play very important role here. Can be kept as empty.

- *positional info* and *same event info*: In a static environment as we have only one pass, so for each sequence s for each item or symbol α the variables related to previous iteration can be skipped ($s_{\alpha}^{en\_previous}$, $bitset_{\alpha}^{previous}$).

For sequence 1 ($sid = 4$) of static database presented in Table 3.1 the corresponding entry in Sequence Summarizer is shown in Figure 3.15 (a). To use the bitset representation for the items we have maintained similar mapping format used during construction of *parent_info*. We have also shown the mapping format in Figure 3.15 (d) using 1 based indexing. For visualization, we have used 1 based indexing to represent the corresponding event number in positional info column. In Figure 3.15 (b), we have shown the technique to update the CETable's columns

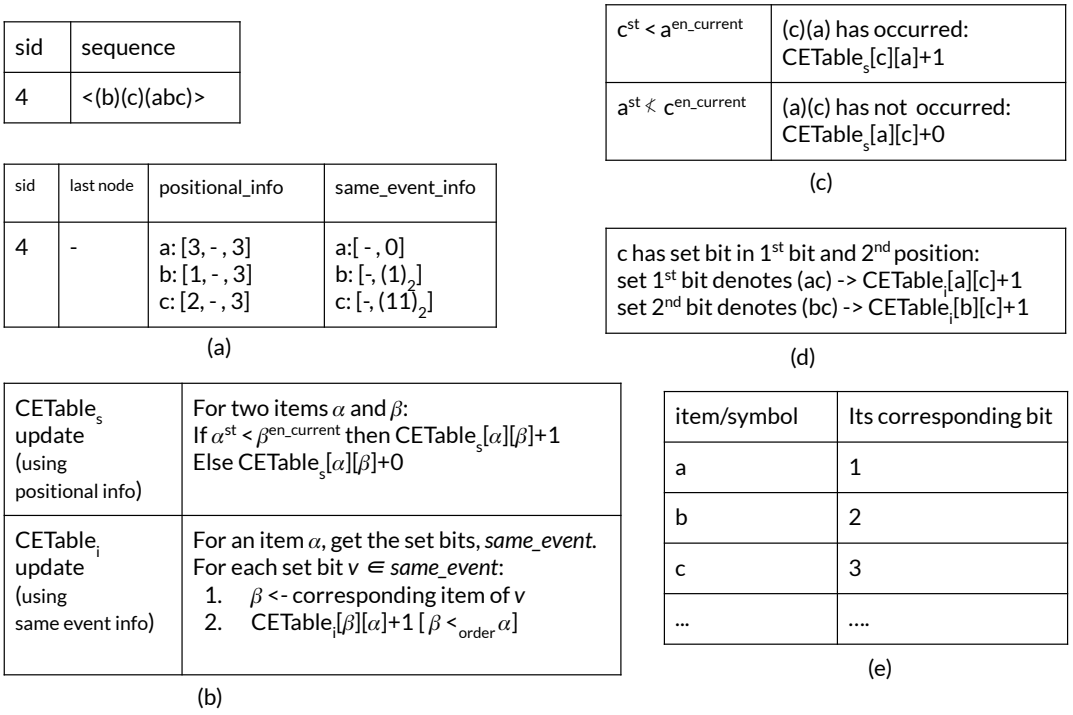


FIGURE 3.15: (a) Sequence Summarizer (seq_sum) for the sequence 4 (b) Strategy to update CETable using Sequence Summarizer in Static Environment (c) An example to update $CETable_s$ column from sequence 4 (d) An example to update $CETable_i$ column from sequence 4 (e) 1 based item mapping format

using Sequence Summarizer in a static environment. In Figure 3.15 (c) and 3.15 (d) we have shown two examples to understand how $CETable_s$ and $CETable_i$ columns will be updated from this sequence for some combinations respectively. We can update Sequence Summarizer directly while inserting sequences in the SP-Tree structures.

3.7 Complexity Analysis of the proposed Static Solution

In this section, we will present some analysis to understand the complexity of our proposal, *Tree-Miner* based on *SP-Tree*. It is very difficult to exactly formulate the Big $\mathcal{O}$ notation because the solution's complexity may vary due to data distribution

TABLE 3.7: Variables used in complexity discussion: static solution

Name	Variables
Database	D
i^{th} sequence in D $(sid = i)$	s_i
Length of s_i	L_i
Number of nodes of SP-Tree	N
Average number of unique symbols in the subtree of each node	u_i
Total number of unique symbols	I
Average number of unique symbols in each sequence	k
Pattern	P
Projection nodes of P	N_P
Support of P	S_P
Total number of patterns tested	Z
Average Projection cost	S^*

and domain characteristics. So, here mainly we will try to provide a cost or number of operations analysis to understand the upper bounds and average bounds. The variables used in this current discussion have been summarized in Table 3.7, we will also further introduce them during discussion.

First, we will try to discuss the space complexity and pre-processing complexity. Here mainly, we have two structures, *SP-Tree* or the tree-based representation of the given sequential database and *Sequence Summarizer* to summarize the information for each sequence. Lets assume, the number of nodes in the SP-Tree is N. SP-Tree gets more compact shape when the sequences get overlapped. But, in the worst case scenario, no prefix sharing will occur and then we will have $N = \sum_{i=1}^{|D|} L_i$ where L_i denotes the length of the i^{th} sequence s_i and D denotes the database. But in practical and average case scenarios, $N << \sum_{i=1}^{|D|} L_i$.

Now, we will talk about $next\_link$ attribute of each node n which is important to assess the memory usage. For each node n, the memory usage caused by $next\_link$ is mainly dependent on the number of unique symbols u in its underlying subtrees. For example, we can consider sequence $<(a)(b)(c)(c)(d)>$. The last or 5^{th} item has no unique symbol or item after it. 4^{th} item has one unique symbol

(d) after it. The 3rd item has two unique symbols (c and d) after it. In the similar manner, the first item has three unique symbols (b, c and d) after it. This analogy can be applied in the tree to understand the dependency over unique symbols. But here we consider the number of unique symbols in the underlying subtree for a node.

For discussion, to formulate the memory usage caused by next links, lets assume that, no prefix sharing occurred. Now, lets take a complete branch b_i consisting nodes $\{n_1, n_2, n_3, n_4,...,n_l\}$ where n_1 is the immediate node after root node in this branch, n_l is the leaf node and n_i is the ancestor node of n_j if $i < j$. So, n_l has no unique symbol underneath. n_{l-1} has one unique symbol (item of n_l), n_{l-p} ($p > 1$) has two unique symbols, n_{l-q} ($q > p$) has three and so on. n_1 will have the most number of unique symbol (lets assume k) underneath in this branch. So, each node in average contains $u_i = \frac{k \times (k+1)}{2 \times l}$ number of unique symbols' information. Now, for simplicity, if we generalize this value for each branch, then the total memory usage caused by next link is $\mathcal{O}(u_i \times N)$. But this value is a very high level upper bound and based on the prefix sharing characteristic (N will reduce) and database characteristics (dense: number of unque symbols will reduce, sparse: sequence length may vary significantly) usage can reduce significantly.

The Co-Existing Item Tables' ($CETable$) memory complexity is dependent on the complete number of unique symbols and found two length patterns' combinations. So, if the total number of unique symbols in the database is I, then the worst case memory usage will be $\mathcal{O}(I^2)$. But in average cases, most combinations do not occur, so the usage will be much less than I^2. Sequence Summarizer's (seq_sum) memory complexity depends on the size of the database ($|D|$) and number of unique symbols in each sequence s_i. If we assume that the average number of unique symbols in each sequence is k, then the memory complexity here is $\mathcal{O}(|D| \times k)$.

Up to now, we have provided an upper bound memory complexity analysis. In Table 3.8 we have shown the summarized discussion of the space complexities of

TABLE 3.8: Static solution's upper bound space complexity

Structural Elements	Upper Bounds
Number of nodes of SP-Tree	$\sum_{i=1}^{\|D\|} L_i$
Total Cost related to *next_link*	$\mathcal{O}(u_i \times N)$
Co-Existing Item Table	$\mathcal{O}(I^2)$
Sequence Summarizer	$\mathcal{O}(\|D\| \times k)$

our static solution with respect to upper bounds and it has already been discussed that the average case scenarios will need much smaller bound than this. Now, we will focus on runtime complexity analysis which consists of two parts, pre-processing runtime complexity and mining runtime complexity.

The cost of inserting the sequence s_i in the tree is L_i where L_i denotes the length of the sequence s_i. So, the total complexity of inserting all the sequence in the tree is $\sum_{i=1}^{|D|} L_i$. Sequence summarizer for each sequence can be updated in the same time having no additional cost. To calculate the $CETable$ from each sequence s_i needs k_i^2 operations where k_i denotes the number of unique symbols in sequence s_i. So, if we assume the average number of unique symbols for each sequence is k. Then the toal complexity to update $CETable$ from the sequence summarizer is $\mathcal{O}(|D| \times k^2)$. All the attributes of the tree can be updated in a single bottom up traversal. The complexity of each node processing here is related to the number of unique symbols it processes. So, from the previous variable set up arrangement, here the total complexity will be $\mathcal{O}(N \times u_i)$ where u_i denotes the average number of unique symbols found from the underneath subtree for a node. And similar as before this is a very high level upper bound and can vary depending on the database characteristics. So, the upper bound of total pre-processing complexity is $\sum_{i=1}^{|D|} L_i + \mathcal{O}(|D| \times k^2) + \mathcal{O}(N \times u_i)$.

Now, we will talk about the mining runtime complexity. Each pattern P has a set of projection nodes N_P and two supporting lists $sList$ and $iList$ to guide it to extend the pattern. Let S_P denotes the support of P and I denotes the total number of unique symbols. The upper bounds regarding the size of N_P, $sList$ and $iList$ are $|N_P| \leq S_P$ (due to node overlapping), $|sList| \leq I$ and $|iList| \leq I$ (initial

set). With recursive gradual pattern extension ($P \rightarrow P\gamma$) corresponding $sList$ and $iList$ get more compacted and support gets reduced $S_{P\gamma} \leq S_P$.

Now, we will perform cost analysis during pattern extension. Suppose, we want to extend P to $P\gamma$. During pattern extension, we start with N_P and finally try to discover the projection nodes for $P\gamma$, $N_{P\gamma}$ using our proposed level by level extension algorithm Breadth-First based support counting technique. The upper bound regarding the size of $|N_{P\gamma}| \leq S_{P\gamma}$. But to reach from $N_P \rightarrow N_{P\gamma}$ we may have to visit some intermediary nodes while traversing through next links. Lets assume the complete set of nodes traversed to reach the final set of projection nodes is $N^*_{P\gamma}$ and $|N^*_{P\gamma}| \leq \mathcal{O}(S_{P\gamma})$. In the best case scenario, through single next link traversals for γ we will reach the desired nodes for each node $n \in N_P$. But in average case scenarios, for some nodes $n'' \in N_P$ we will not reach the desired node through single next link traversals. In that case, we have to search more in the underlying subtrees. In this regard, during sequence extension we will need maximum two level traversals but in itemset extension we may need some more. So, overall $|N^*_{P\gamma}| \leq \mathcal{O}(S_{P\gamma})$ sustains with some constant factors with $S_{P\gamma}$.

In the case of frequent patterns, we have to discover the complete set of projection nodes. But during infrequent patterns, the mining performance will improve with the earlier detection regarding the infrequency of the pattern before performing complete projection ($\mathcal{O}(S_{P\gamma})$). With the early detection of the infrequency we stop from further projecting. We also apply additional set of pruning strategies in this regard. In Line 4 of Algorithm 3, we perfrom CETable based pruning where the total complexity $\mathcal{O}(|e_{last}| \times (|sList| + |iList|))$. Here $|e_{last}|$ denotes the number of items in the last event of P. In Line 9 of Algorithm 3, we perform Heuristic Pruning which can be directly calculated during projection for sequence extension. In Line 15 and 16 of Algorithm 3 we perform recursive $sList$ and $iList$ generation before the next level function call to extend the pattern.

In summary, if we have tested total Z number of patterns and the average time to calculate the projection or frequency detection for each pattern is S^*,

TABLE 3.9: Static solution's upper bound runtime complexity

Function	Upper Bounds
Sequence Insertion	$\sum_{i=1}^{\|D\|} L_i$
Node attributes calculation	$\mathcal{O}(N \times u_i)$
$CETable$ calculation	$\mathcal{O}(\|D\| \times k^2)$
Pre-processing Complexity	$\sum_{i=1}^{\|D\|} L_i + \mathcal{O}(\|D\| \times k^2) + \mathcal{O}(N \times u_i)$
Pattern Projection Complexity of P	$\mathcal{O}(S_P)$
Total Mining Complexity	$\mathcal{O}(Z \times S^*)$ $Z << (2^T - 1)^E$ $S^* << \frac{S_{P_1}+S_{P_2}+S_{P_3}+...+S_{P_Z}}{Z}$

then the total mining complexity is $\mathcal{O}(Z \times S^*)$. Now, if the average number of events in the complete database is E and average number of items in each sequence is T. Then an approximation over the total number of patterns will be $(2^T - 1)^E$. As we have a group of pruning strategies, lots of infrequent patterns will be detected at an early stage through performing a small size projection, using different heuristics and supporting structures which makes $Z << (2^T - 1)^E$. Also, $S^* << \frac{S_{P_1}+S_{P_2}+S_{P_3}+...+S_{P_Z}}{Z}$ because of our proposed level by level manner support counting technique along with other strategies. Similar to space complexity discussion, the summarized version of the runtime complexity discussion has been presented in Table 3.9 with respect to upper bounds. In this section, we have presented the cost analysis of the core and crucial factors of Tree-Miner based on SP-Tree. In the upcoming sections, we will present our incremental solution along with its complexity analysis.

3.8 IncSP-Tree: A Tree-Based Structure to represent the Incremental Database

In this section, we will talk about our incremental tree-based structure IncSP-Tree. IncSP-Tree is an extended version of SP-Tree to represent the incremental database. If SP-Tree is used to solve the incremental mining problem, then for each database update, it has to re-build it from scratch. IncSP-Tree is designed in

such a way, so that it can efficiently capture the incremental database by updating the existing tree having no requirement of building from scratch. We have also developed an incremental mining algorithm, IncTree-Miner based on IncSP-Tree to efficiently track the patterns which are affected due to the addition of incremental database rather than complete re-mining over the full database from scratch.

As IncSP-Tree is an extended version of static SP-Tree, it consists of all the attributes of SP-Tree ($item$, $event\_no$, $child\_node$, $next\_link$, $parent\_info$, S^{MAL}) along with the addition of some new attributes ($created\_at$, $modified\_at$, and $modified\_next\_link$) and updated attributes ($present\_count$, $previous\_count$) to tackle the challenge of re-building and re-mining from scratch. Now, we will talk about the newly added attributes of IncSP-Tree.

- *created_at, modified_at:* These attributes are used to store the information that in which pass the node was first created and last modified respectively.

- *present_count, previous_count:* Count related attributes represent the number of sequences that have overlapped this node. Here *present_count* and *previous_count* denote the total number of overlapping of this node up to the current pass and previous last mined pass respectively. These attributes are helpful to track the change in support from the last mined iteration for a pattern. For the static SP-Tree there was only one single count attribute.

- *modified_next_link:* This attribute has similar characteristics like *next_link* except that it only tracks the modified nodes found as the first occurrence in its disjoint subtrees. This attribute helps to efficiently and implicitly track the incremental database. Modified nodes are those nodes that are affected due to the addition of the incremental database.

Now, we will talk about the construction mechanism of IncSP-Tree. As IncSP-Tree is an extended version of SP-Tree having some additional attributes, both

Algorithm 4 Insert sequence into IncSP-Tree

1: **Input:** pass p, node N, sequence s, *event_no*, *item_no*, *prev_max_event*, *seq_sum*, *event_bitset*. ▷ *event_bitset*: Current Event's bitset represent.
2: **Output:** s inserted into the tree, last node is returned, *seq_sum* updated.
3: **procedure** InsertIncSPTree
4: **if** (*event_no* == len(s)) **then Return** N ▷ All the events been inserted
5: **if** (*item_no* > len(s[*event_no*])) **then** ▷ Add next event
6: **Return** InsertIncSPTree(p,N,s,*event_no* + 1,1,*prev_max_event*, *seq_sum*, 0) ▷ 1 Based Indexing
7: *item* = s[*event_no*][*item_no*]
8: *ev* = *event_no* + *prev_max_event* + 1 ▷ Actual Event No.
9: **if** (N.*child_node*[{*item*, *ev*}] == {}) **then** ▷ No child found
10: ▷ create a child node n and initialize the attributes
11: n.item=item, n.event_no=ev,n.created_at=p,n.previous_count=0,n.present_count=0,
12: n.parent_info=event_bitset,n.modified_at=0 ▷ modified_at will be set as p in Algo. 5,(initialized now)
13: N.*child_node*[{*item*, *ev*}] = n ▷ Setting n as child node
14: n = N.*child_node*[{*item*, *ev*}]
15: **if** (*seq_sum*[*item*] == {}) **then** $seq\_sum[item]_{st} = ev$
16: $seq\_sum[item]_{en\_previous} = ev, seq\_sum[item]_{en\_current} = ev$ ▷ Updating *seq_sum* (Positional Info)
17: **else** $seq\_sum[item]_{en\_current} = ev$ ▷ Updating the ending position
18: $v = seq\_sum[item]_{bitset_{current}}$ OR *event_bitset* ▷ set bit/items are in same event
19: $seq\_sum[item]_{bitset_{current}} = v$ ▷ Updating *seq_sum* (Same Event Info)
20: **Return** InsertIncSPTree(p,n,s,*event_no*,*item_no* + 1, *prev_max_event*, *seq_sum*,
21: *event_bitset* **OR** *item*) ▷ Set bit for *item* in event_bitset for next call

maintain same mechanism to construct the tree and update the attributes. In Algorithm 4, we have provided the pseudo-code of tree construction and in Algorithm 5, we have provided the pseudo-code to update the tree attributes.

Algorithm 4 is called with the pass number p, a node N from which we will extend, a sequence s which will be inserted, current *event_no* and *item_no* to specify the item which is being inserted, *prev_max_event* denotes the last maximum event number for the corresponding *sid* of the appended sequence s in the previous iteration to get the actual event number for the appended new event, Sequence Summarizer *seq_sum* to update the summarizing information for the total sequence with the corresponding *sid* and the bitset representation of the corresponding event (*event_bitset*) whose item is being inserted. This function is designed in such a way so that a new appended sequence (*Append* operation) and a complete new sequence (*Insert* operation) can easily be inserted. Inserting a sequence in the static SP-Tree is just a sub task of inserting a sequence in the incremental SP-Tree or IncSP-Tree.

Line 4 means all the items of all the events of s are inserted into the tree, so we can stop from further recursive calling. Line 5 indicates that current event's

all the items are inserted. So, we start to insert the next event's items through recursive calling (line 6). In 7, we extract the current item (*item*) which will be inserted into the tree and in line 8, we calculate the actual event number for the corresponding item. In case of *Append* operation, the actual event number for the item is calculated by summing with previous maximum event number. In case of $Insert$ operation, we can simply initialize $prev\_max\_event$ as -1. In line 9, we check, if we have any child from N with item and event number combination. If there is no child then we create a child for the combination and initialize the node attributes (line 10-13). In line 15-19, we update the sequence summarizer of the corresponding *sid* for the current item. In lines 15-17, we update the first starting and last ending position of the *item* in the full sequence. In line (18-19) we gather the information of the other items β which have occurred with *item* in the same event in bit based format by numbering the items and allocating a bit position for it. We have already seen that these information are helpful to calculate the co-occurrence information of the items which guide during mining. We will get more idea regarding the usefulness of this information in section 3.9. Finally, we again recursively call the function (line 20-21) to insert the next item in the tree by going to child node n (set in line 14). In the recursive call, we set the corresponding bit for *item* ($event\_bitset$ **OR** *item*) in $event\_bitset$ to indicate with items β are already in the same event with the next item to be inserted. For compactness and operational efficiency issues, we have used bitset representation for saving item information, but other representations will also work.

After inserting the sequence we return the last node of the tree where the sequence ended to save it in the sequence summarizer for further *Append* operations. In $Insert$ operation, we start from the root ($N = root$) and in *Append* operation we start appending from the last node where the earlier sequence with the corresponding *sid* ended in the previous iterations. The last position where the sequence ended prior can be found from the Sequence Summarizer for the corresponding sequence in the *last node* column. The function's recursion call ends

Algorithm 5 Updating Attributes of IncSP-Tree

Input: node N, Pass p, $next\_link$, $modified$, $type$, $last\_node$, $last\_node\_flag$, S^{MAL}.
Output: Update of the concerned branch's attributes.
procedure UPDATE PATH
  if (N is **None**) **then Return** ▷ All the node's attributes are updated
  $last\_modified = N.modified\_at$
  if ($N.modified\_at < p$) **then** $N.modified\_at = p$
    $N.previous\_count = N.present\_count$ ▷ Support Tracking
    $N.modified\_next\_link = \{\}$ ▷ Runtime memory clear
  for (item $i \in modified$) **do** ▷ Tracking underlying modified nodes
    $N.modified\_next\_link[i] \cup modified[i]$
  for (item $i \in S^{MAL}$) **do** $N.S^{MAL}[i]++$ ▷ to reduce loop
  if ($last\_modified \neq N.modified\_at$) **then** ▷ N was not tracked
    $modified[N.item] = N.item$ ▷ Need to track N from ancestors
  else Delete $modified[N.item]$ ▷ Already Been Tracked
  if ($last\_node == N$) **then** $last\_node\_flag = True$ ▷ Initially $False$
  if ($type$=$Insert$) **then** $N.present\_count$++,$S^{MAL}[N.item]$=1
  else ▷ Append Operation
    if ($last\_node\_flag == False$) **then** ▷ new node encountered
      $N.present\_count++, S^{MAL}[N.item] = 1$
    else Delete $S^{MAL}[N.item]$ ▷ Old nodes, Already Tracked
  for (item $i \in next\_link$) **do** $N.next\_link[i] \cup next\_link[i]$ ▷ Tracking
  if ($N.created\_at == p$) **then** ▷ Tracking nodes for next link
    if ($last\_modified == N.modified\_at$) **then**
      Delete $next\_link[N.item]$ ▷ Already Tracked for next link
    else $next\_link[N.item] = N$ ▷ Tracking node for next link
  else Delete $next\_link[N.item]$ ▷ Already Tracked by ancestors
  UpdatePath($N.parent, p, next\_link, modified, type, last\_node,$
  $last\_node\_flag, S^{MAL}$) ▷ Update ancestor nodes' attributes

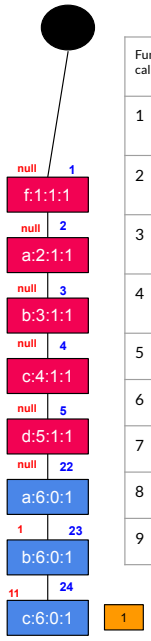
when all the items of the newly added sequence is inserted into the tree.

Now, we will discuss how the node attributes will be calculated in the tree using Algorithm 5. Both SP-Tree and IncSP-Tree follow similar procedure to update the attributes where SP-Tree contains a subset of attributes from IncSP-Tree. The function UpdatePath is called with a node N, current pass number p, a map structure named $next\_link$ to calculate the next link information of node N, a map structure named $modified$ to update the modified next link information of N, a variable $type$ to indicate the category of operation ($Insert$ or $Append$), a variable named $last\_node\_flag$ which is used to perform $Append$ operation to identify the newly considered nodes due to the addition of the appended sequence and a map structure S^{MAL} to calculate the S^{MAL} attribute of N. We start from the last node N which occurred due to the addition of the sequence in the tree and gradually update the attributes' information in bottom up recursive manner. We stop when all the attributes of all the nodes from the last node (considered

due to the insertion of the complete sequence) to the root node is calculated.

Line 4 indicates all the nodes are updated so the function returns. In Line 6, we check if N is already been updated in the current pass or not. If it is not then we update the $modified\_at$ attribute (line 6), update the previous count attribute to track the change from last modification (line 7) and perform runtime clearing of $modified\_next\_link$ attribute of N (line 8). In line (9-10) we update modified next link attribute of N for each modified item or symbol. Similarly, we update S^{MAL} attribute in line 11. In line 12, we check if N is already tracked as a modified node by its ancestor nodes or not. If it is not then we update the corresponding entry for N's representing item in line 13. But if N is already tracked as a modified node, we delete the entry from $modified$ in line 14.

Line 15 is important to perform *Append* operation where we check if the last node for the corresponding *sid* of previous iteration is encountered or not. Line 16, is for $Insert$ operation, where we simply increase the *present_count* attribute and update entry for S^{MAL} for N's item because all the nodes of the corresponding path is newly considered for the sequence. Block 17-20 executes if the operation is *Append* where if N is a newly considered node we update *present_count* and S^{MAL} (line 19) otherwise if N is already considered in the previous iterations for the existing sequence we remove S^{MAL} for N's item as it already been tracked (line 20). In line 21, we update the next link information of N based on the returned information from the successor nodes. In lines 22-26, we update the *next_link* map structure to send information upwards in the tree. If N was created in earlier iterations, then it is already tracked and so we do not need to send this information upwards. Thus, we delete the entry for N's item from next link (line 26). If N was created in current pass (p) but was not earlier tracked then we update the information in next link map structure (line 25), otherwise we remove the entry (line 24-25). Finally we move upwards in the tree recursive by going into N's parent node (line 27-28) with the updated information. Our proposed solution does not have dependency of mining after each iteration. We just need to



Function call	N	previous_count : present_count	Created_at (pass no.)	next_link	S^{MAL}	last _modified	Updated modified_at	modified_next_link
1	24	0:1	2	c:[24]	c	0 (initialized)	2	c:[24]
2	23	0:1	2	c:[24],b[23]	c, b	0 (initialized)	2	c:[24],b[23]
3	22	0:1	2	c:[24],b[23], a:[22]	c,b,a	0 (initialized)	2	c:[24],b[23],a:[22]
4	5	1:1	1	c:[24],b[23], a:[22]	c,b,a	1	2	c:[24],b[23],a:[22],d:[5]
5	4	1:1	1	b[23],a:[22]	b,a	1	2	c:[4],b[6],a:[22],d:[5]
6	3	1:1	1	a:[22]	a	1	2	c:[4],b[3],a:[22],d:[5]
7	2	1:1	1	-	-	1	2	c:[4],b[3],a:[2],d:[5]
8	1	1:1	1	-	-	1	2	c:[4],b[3],a:[2],d:[5],f:[1]
9	Root	5:5	1	-	-	1	2	-

Modified nodes whose support got changed/has modified nodes underneath.

Modified nodes which were newly added in the tree.

FIGURE 3.16: A simulation for updating the node attributes for $sid = 1$ in 2^{nd} pass

carefully update the count and $modified\_next\_link$ attributes for a node to track the changed support and underlying modified subtrees from the last mined pass. In the algorithm we have assumed to mine after each pass.

In Figure 3.16, we have shown a simulation of node attribute update for the leftmost branch where an *Append* operation is performed for $sid = 1$. We have maintained similar named variables as used in Algorithm 5. Here the columns show the value which is found at the end of the corresponding function call. To summarize, new nodes are tracked in $next\_link$, newly modified nodes are tracked in $modified\_next\_link$ and S^{MAL} is updated when underlying support changes for an item. $present\_count$ and $previous\_count$ attributes' value for root node is shown as $5 : 5$ because for simplicity we have assumed that incremental database gets updated in the ascending order of sid (no $Insertion$ operation occurred before appending new sequence to $sid = 1$).

In Figure 3.17, we have shown the intermediary steps and the complete IncSP-Tree after second iteration for the incremental database shown in Table 3.2. White

nodes are created in the first pass and blue nodes are created in the second. We have numbered the nodes (in blue color) for discussion purposes and also we have numbered in such a way which depict their creation order (for simplicity and discussion purpose we have considered that sequences come based on ascending order of *sid*). The red color information of each node represents the $parent\_info$. Inside each node, we have shown 4 values, $item, event\_no, previous\_count$ and $present\_count$ respectively. Before updating an existing branch, first we update the previous count with the present to track the support in the previous pass and then we perform update where in $Insert$ operation every node's count increases but in *Append* operation only the nodes which were considered due to the new items' arrival in the incremental sequence get incremented for the concerned sequence. For example nodes (22-24) get incremented but nodes (1-5) do not get incremented after the 2nd pass. This is an *Append* operation over the sequence with sid 1. We have used orange value to indicate which sequence ends where after completion.

Modified next links are similar to normal next links except the prior is used to implicitly track the incremental database or modified subtrees. It ultimately helps to perform faster tracking of the patterns which are affected patterns due to the addition of incremental database. In Figure 3.18, we have shown an example of $next\_link$ and $modified\_next\_link$ for some of our nodes of IncSP-Tree shown in Figure 3.19 to understand how through $modified\_next\_link$ we can perform faster traversal in the incremental database or modified subtrees. In Figure 3.19, we have shown the modified nodes of the complete IncSP-Tree after second iteration. Basically patterns consisting of the colored nodes are affected due to the addition of new incremental database and our proposed incremental mining algorithm only considers these nodes and find the affected frequent patterns.

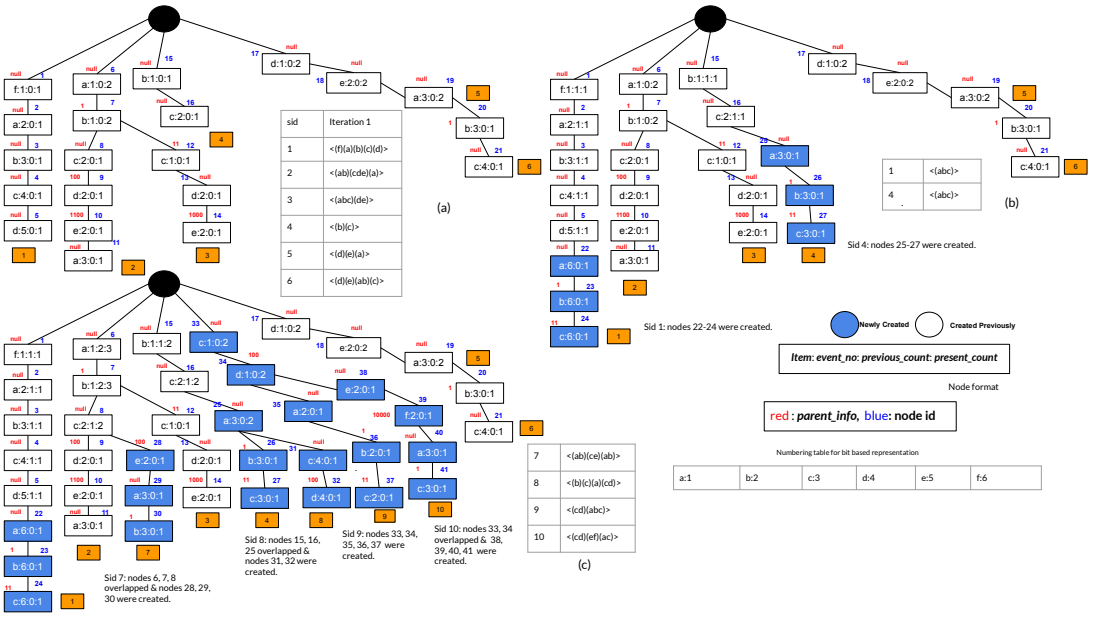


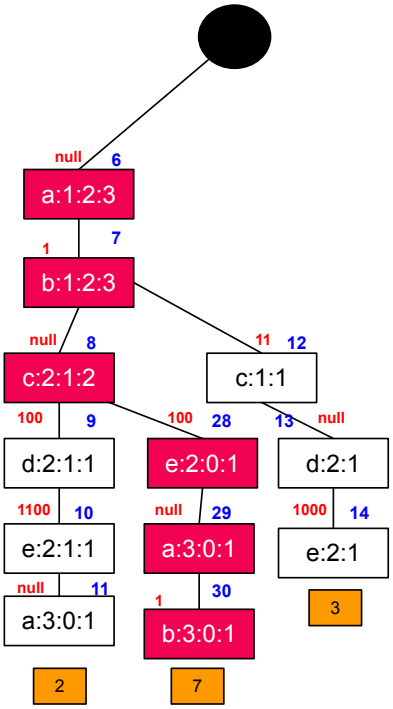
FIGURE 3.17: (a) IncSP-Tree after iteration 1, (b) IncSP-Tree after insertion of appended sequences (sid = 1, 4), (c) IncSP-Tree after insertion of new sequences (sid = 7, 8, 9, 10)

3.9 Usage of Sequence Summarizer in an Incremental Environment

We have already given the general definition of Sequence Summarizer in Section 3.6. We have discussed the variables and how it helps to calculate the co-occurrence information or update the CETable. We have already given a small highlight that, sequence summarizer is most useful in an incremental environment. Here, we will state how it improves performance in an incremental environment along with the strategy to efficiently and accurately update the existing CETable in the arrival of new information or incremental database.

First, we will pin point the issues which is improved by this structure in an incremental environment.

- *last node* - Using last node column we can efficiently perform *Append* operation. During *Append* operation, using this column, we get the information



node	*next_link*	*modified_next_link*	S^{MAL}
6	a:[11, 29], b:[7], c:[8,12], d:[9,13], e:[10, 28, 14]	a:[29],b:[7], c:[8], e:[28]	a:2,b:3 c:3,d:2 e:3
7	a:[11, 29], b:[30], C:[8,12], d:[9,13] e:[10, 28, 14]	a:[29], b:[30],c:[8], e:[28]	a:2,b:1,c:3, d:2,e:3
8	a:[11, 29], b:[30], d:[9] ,e:[10, 28]	a:[29], b:[30],e:[28]	a:2,b:1,d:1, e:2
9	a:[11], e:[10]	NULL	a:1,e:1
10	a:[11]	NULL	a:1
11	NULL	NULL	NULL
28	a:[29],b:[30]	a:[29],b:[30]	a:1,b:1
29	b:[30]	b:[30]	b:1
30	NULL	NULL	NULL
12	d:[13],e:[14]	NULL	d:1,e:1
13	e:[14]	NULL	e:1
14	NULL	NULL	NULL

FIGURE 3.18: Example of $modified_next_link$s for some nodes

where the sequence ended prior and directly start appending from that node rather than searching the existing sequence to reach that node again.

- *positional info* and *same event info*: To update the CETable's columns for this sequence, we consider only the newly appeared unique items and prior occurred unique items. Then using these two columns we find which new combinations (as SE or IE) have occurred completely new and update $CETable_s$ and $CETable_i$ accordingly. We have shown the incremental updating approach in Figure 3.20. Our strategy do not over count and wrongly count any combination and as we do not have to scan the complete sequence

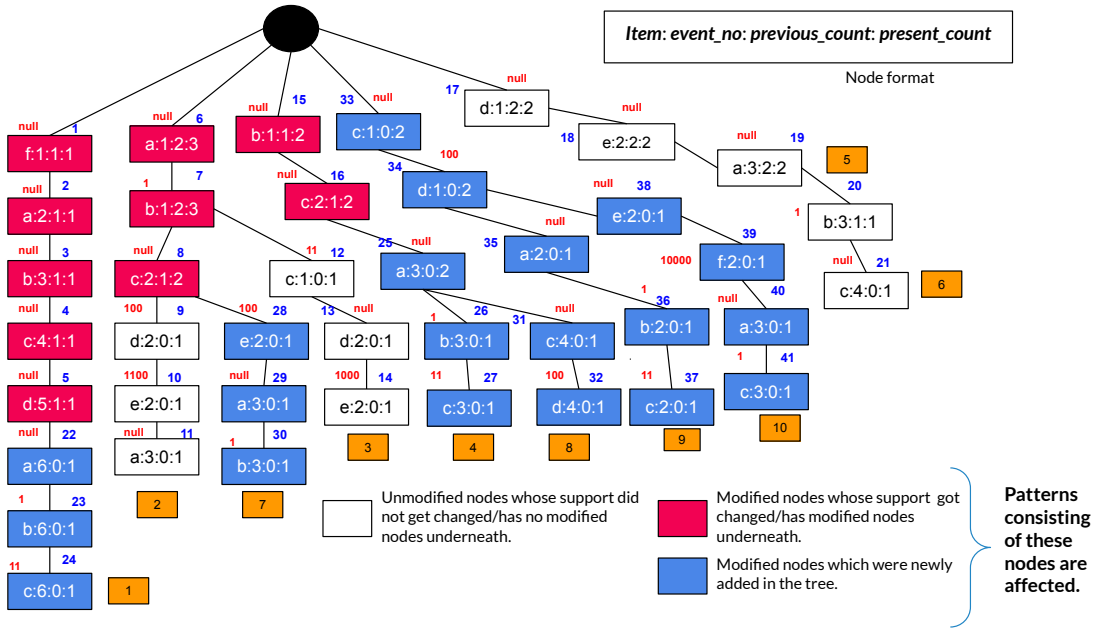


FIGURE 3.19: Modified nodes of IncSP-Tree after second iteration

rather only consider the unique symbols which ultimately improves performance.

Now, we will show an example of how the table's values are stored. For sequence with sid 4 from TABLE 3.2, after pass 2 the *positional info* and *same event info* of c will be $(2, 2, 3)$ and $(0, 11)$[1]. After first iteration of the database the node reference stored in *last node* column was 16 (Figure 3.17 (a)) and after second iteration the value was updated to 27 (Figure 3.17 (b)) as the new position where the sequence $(sid = 4)$ ended. After updating CETable, we update summarizer's information for the following iterations ($s_\alpha^{en\_previous} = s_\alpha^{en\_current}$ and $bitset_\alpha^{previous} = bitset_\alpha^{current}$). In Fig. 3.20, we describe the idea of updating $CETable_s$ and $CETable_i$ from a sequence summarizer structure for a updated sequence s based on the newly appended or inserted part. After *Append* or *Insert*

[1]order wise $a = 0, b = 1$, so we set bit in the 1^{st} and 2^{nd} position getting $11_2 = 3_{10}$

operation, this function is executed over the concerned sequence. In the algorithms, capital and bold *AND, OR, XOR* represents bitwise and, or and xor operation respectively. This convention is maintained throughout the discussion. Through bitwise operations, we make our computations significantly faster. The idea represented in Figure 3.20 is a generic strategy, so it can also be applied in a static environment.

Now, we will discuss the workflow of the Figure 3.20 to understand how summarizer can update CETable. First we will talk about how $CETable_s$ is updated (Figure 3.20 (a)). For a sequence s, for each new symbol α and β occurred in s in current iteration, we test the condition $s_\alpha^{st} < s_\beta^{en\_current}$ which denotes the occurrence of $(\alpha)(\beta)$ in s. Now, we need to check if this occurrence has already been counted for $CETable_s$ or not. If $s_\alpha^{st} \geq s_\beta^{en\_previous}$, it denotes that α might occurred first in the current iteration or $(\alpha)(\beta)$ did not occur before. Similarly, if $s_\alpha^{st} < s_\beta^{en\_previous}$ and $s_\beta^{st} >$ previous maximum event no, it denotes that β first occurred in the current pass. Both of these cases indicate that $(\alpha)(\beta)$ occurrence was not calculated before for this sequence. So, we update $CETable_s[\alpha][\beta]$.

Now, we will talk about how, we can update $CETable_i$ (Figure 3.20 (b)). Here, the main goal is to discover all the new $(\beta\alpha)$ occurrences where $\beta <_{order} \alpha$. For each new symbol α appeared in the current iteration for s, we perform the calculation. First, we perform bitwise XOR operation between $bitset_\alpha^{previous}$ and $bitset_\alpha^{current}$ which gives some set bits in $unique_{bits}$. For each set bit (or item β) in $unique_{bits}$ we get a $(\beta\alpha)$ occurrence for the corresponding item β (Items can be mapped). So, a certain number or a position corresponds to an individual item). Then, we perform bitwise AND operation between $unique_{bits}$ and $bitset_\alpha^{current}$ which provides us the new $(\beta\alpha)$ occurrences in s that were not found before. Finally, we track the new $(\beta\alpha)$ occurrences in $CETable_i$. Bitwise operations provide us implementational efficiency and compact representation. So, we have used them in our implementation, but other implementations will also

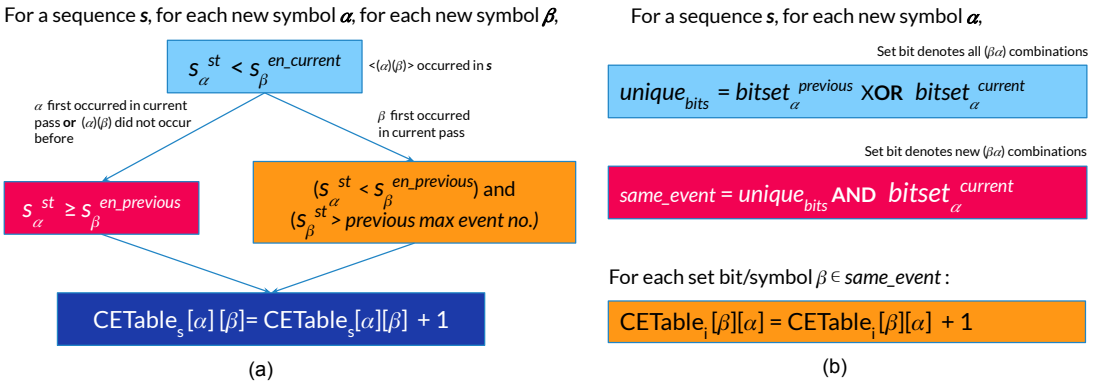
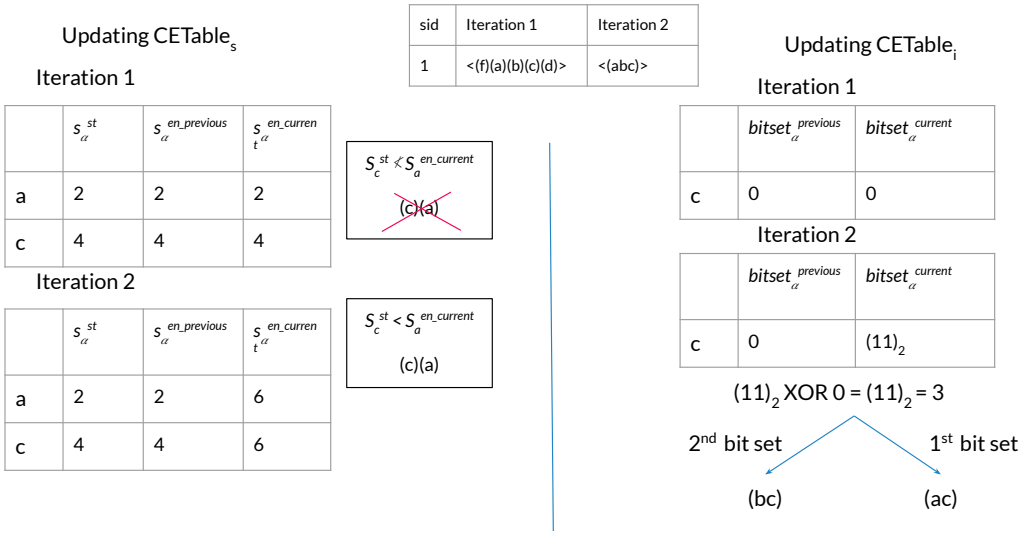


FIGURE 3.20: (a) Generic Strategy to update $CETable_s$ from Sequence Summarizer, (b) Generic Strategy to update $CETable_i$ from Sequence Summarizer



sid	Iteration 1	Iteration 2
1	<(f)(a)(b)(c)(d)>	<(abc)>

	s_α^{st}	$s_\alpha^{en\_previous}$	$s_\alpha^{en\_current}$
a	2	2	2
c	4	4	4

	s_α^{st}	$s_\alpha^{en\_previous}$	$s_\alpha^{en\_current}$
a	2	2	6
c	4	4	6

	$bitset_\alpha^{previous}$	$bitset_\alpha^{current}$
c	0	0

	$bitset_\alpha^{previous}$	$bitset_\alpha^{current}$
c	0	$(11)_2$

FIGURE 3.21: An Example of Sequence Summarizer for sequence 1 (sid=1) for the Incremental Database shown in Table 3.2.

work. We just have to efficiently calculate the new $(\beta\alpha)$ occurrences. This summarizer structure is helpful to calculate co-occurrence information for both static and incremental mining. But it plays a very important role during incremental mining by efficiently merging the co-occurrence information of previous and new iterations.

A small example to understand how sequence summarizer works in an incremental environment can be found in Figure 3.21.

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

In ISPM problem, after each iteration, we need to report the current list of frequent patterns with their support. So, to store the frequent patterns traditional literature maintains a tree-like structure [10, 38]. In our study, we propose a new tree-based structure named *Bi-Directional Projection Pointer Based Frequent Sequential Pattern Tree (BPFSP-Tree)* to store the patterns with their support and projections using *IncSP-Tree* node pointers. An important point to note that generally in most of the cases (or after some iterations) the size of the incremental database gets quite smaller compared to the size of the existing database. Thus why, generally after some iterations, a pattern's transition from frequent to infrequent or vise-versa occurs slowly. Moreover, it is the normal characteristic that either an existing frequent pattern's super pattern will get infrequent to frequent or a current frequent pattern along with its some of the sub-patterns will get frequent to infrequent also with the addition of some completely new patterns with new prefix as frequent. So, it is important to give focus on the frequent patterns' representation along with the mechanism of pruning the infrequent patterns from the tree to free up the redundant space. In summary, the main features and characteristics of our proposed *BPFSP-Tree* is as follows -

1. *Pattern and Support:* Each node represents an item and combining all the node's items from root up to this node a unique pattern is formed. A node can be appended to its parent node either as sequence extended or itemset extend, so both types of patterns can be formed through this tree. Each node also has a support that denotes the pattern's support.

2. *Projection Pointer:* SP-Tree structures (*SP-Tree* and *IncSP-Tree*) provide an efficient, compact and structured representation of the database along

with projection pointers to represent a pattern (the first set of positions where the pattern has ended). BPFSP-Tree saves this projection information using IncSP-Tree node references for a pattern which ultimately helps to reduce the number of tree scans during recursive pattern extensions.

3. *Non-Frequent Item Buffer(NIB):* Suppose for a node in BPFSP-Tree, we have the pattern P and we want to extend it for γ. But after support calculation over the complete database, we found that extending with γ does not satisfy *min_sup*. So, $P\gamma$ will not be frequent. But as we have calculated the actual support over the complete database for $P\gamma$, we store only the support information for γ for this node/pattern in a buffer named *Non-Frequent Item buffer(NIB)*. This information helps to minimize cost and implicitly merge $P\gamma$'s support over the updated DB in the following iterations by combining the solution of the modified nodes' aka additional support and unmodified nodes' aka old support.

4. *Bottom up pruning using end-link*: Each leaf node of BPFSP-Tree has an attribute named *end-link* to traverse through the leaf nodes or maximum length patterns for each recursive call. During infrequent pattern removal from BPFSP-Tree we start from bottom or leaf nodes and progress upwards. Apriori property wise in case of previously frequent patterns' transition to infrequency super patterns or larger length patterns will become infrequent first. Due to this approach, we need to traverse lesser patterns to remove the complete set of infrequent patterns.

   An example of BPFSP-Tree is shown in Figure 3.22 with *end-link*s for traversing through the leaf nodes. Each node represents a pattern (by concatenating the symbols from root to up to this node) along with its support and projection pointers. Its projection pointers are represented by node numbers. Using *end-links* we can traverse through the leaf nodes and can prune infrequent patterns from bottom.

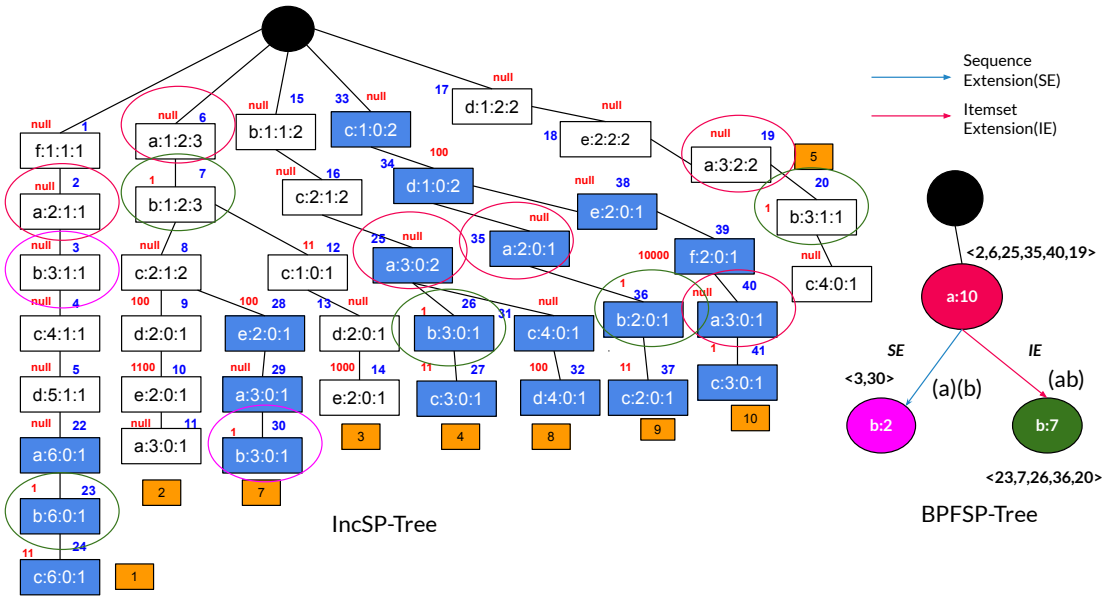


FIGURE 3.22: An Example of BPFSP-Tree to store the patterns' projections using IncSP-Tree node references

3.11 Additional Pruning Strategies

Our proposed IncTree-Miner based on IncSP-Tree is an extended version of Tree-Miner based on SP-Tree to efficiently solve the incremental mining problem. IncTree-Miner includes all the aforementioned pruning strategies including some new ones suitable for incremental mining problem. We will discuss these strategies in this section.

3.11.1 Non-Frequent Item Buffer (NIB)

We have talked about this attribute in brief in Section 3.10 while introducing the attributes of BPFSP-Tree. In this section, we will investigate the advantage of it in detail.

Suppose, for an infrequent symbol α we perform complete projection and find support S. Then, we store this support information in the item buffer so that, we can use it in the successive mined iteration. Merging S with the found support

from the incremental database we may get a complete idea regarding a previously infrequent pattern during successive iterations.

This strategy ultimately helps to reduce the cost of some infrequent patterns' searching during pattern generation by providing support from previous mined iteration. Based on the information, we can get the complete idea regarding the pattern and ultimately decide if we need to perform any projection for it in the unmodified subtrees or not which ultimately helps improve mining time by reducing the number of projections.

3.11.2 Bottom-up Pruning Strategy in BPFSP-Tree

We have discussed this strategy while talking about the node attributes of BPFSP-Tree in brief. In this section, we will talk about this in detail with an example to understand the concept properly.

During removal of the previously frequent but currently infrequent patterns from the BPFSP-Tree, we start from bottom or from the leaf nodes. Apriori property wise in case of previously frequent patterns' transition to infrequent, super patterns or larger length patterns will become infrequent first. It helps to search lesser patterns during removal from the BPFSP-Tree. For example, suppose, we want to remove the infrequent patterns from the BPFSP-Tree shown in Figure 3.23 where the red nodes denote infrequent patterns. Here, if we start from top then we need to visit 12 nodes to remove complete set of infrequent patterns whereas if we start from top then we need to check 9 nodes.

3.12 IncTree-Miner

In this section, we will discuss our incremental mining algorithm IncTree-Miner based on IncSP-Tree. SP-Tree alike structure based IncTree-Miner includes all the concepts and strategies from Tree-Miner including some new ones suitable for incremental mining.

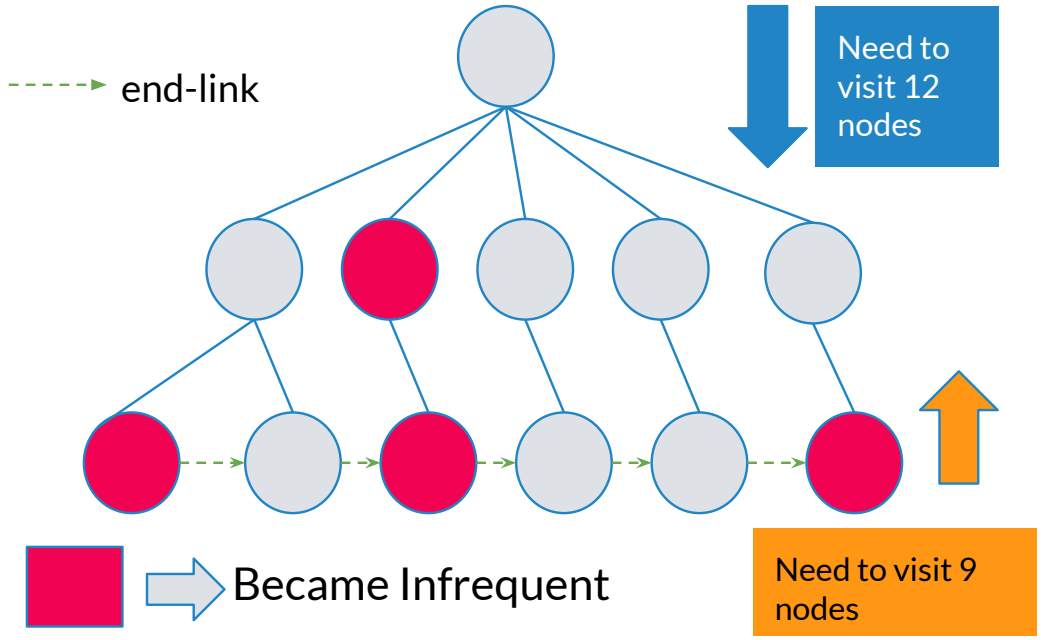


FIGURE 3.23: Bottom-up Pruning Strategy in BPFSP-Tree

IncTree-Miner algorithm finds the complete set of frequent patterns from the updated database. After each iteration, basically three types of events can occur - some infrequent patterns become frequent (patterns with completely new prefix or new suffix with the existing ones), some frequent patterns become infrequent and some frequent patterns' support get incremented. The main challenge of an incremental mining algorithm is how to efficiently track those patterns which are affected due to the addition of incremental database rather than complete mining. In Table 3.10, we have shown an example of each case. Besides including the aforementioned three cases, we also shown an extra case that there may be some infrequent patterns which still remain as infrequent even after the addition of the incremental database. Here δ (or δ_D) and δ' (or $\delta_{D'}$) was 2 and 3 respectively where δ denotes the minimum support count to be frequent in the corresponding databases. IncTree-Miner is efficiently designed to identify these transitions. There are some important observations to highlight in IncTree-Miner,

TABLE 3.10: Pattern Transition cases of IncTree-Miner ($\delta = 2, \delta' = 3$)

P	S_P^D	$S_P^{D'}$	Transition
$<(a)(e)>$	2	3	Frequent $\rightarrow$ Frequent
$<(c)(a)>$	1	7	Infrequent $\rightarrow$ Frequent
$<(de)>$	2	2	Frequent $\rightarrow$ Infrequent
$<(a)(b)>$	1	2	Infrequent $\rightarrow$ Infrequent

Definition 3.7 (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 incremental database. 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' where $C = present\_count$ attribute's value and $S_P^{db} = C'(n_2) + C'(n_4)$ denotes the additional support of the pattern occurred due to the incremental database (db) where C' is $present\_count - previous\_count$ attribute's value of each modified node.

Now, based on the Definition 3.7 and Figure 3.24, we will discuss the mining concepts of IncTree-Miner. 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, which ultimately helps track the changed patterns efficiently. In Fig. 3.24, we have shown the idea of our pattern mining approach regarding the extensions in modified and unmodified subtrees. The terminologies used in this discussion are summarized in Table 3.11.

Suppose, we have a pattern P ending at nodes N_P which basically indicates the projection nodes of P. N_P consists of two types of projection nodes, modified nodes (N_P^M) and unmodified nodes (N_P^U). IncTree-Miner targets to find these two types of projection nodes separately. The total updated support of P in D' is $S_P^{D'}$, which is the summation of all the projection nodes' $present\_count$ attribute's value. $S_P^{D'}$ consists of two supports, old support of P in D (S_P^D) and the additional support (S_P^{db}) occurred due to the addition of incremental database (db). S_P^{db} is

TABLE 3.11: Terminologies used in IncTree-Miner

Term	Description
D	Old Database
D_{Append}	Append: $\{s \mid s_{sid} \in D\}$
D_{Insert}	Insert: $\{s \mid s_{sid} \notin D\}$
$db = D_{Append} \cup D_{Insert}$	Incremental Database
$D' = D \cup db$	Updated Database
$\delta = \lceil min\_sup \times \lvert D \rvert \rceil$	Minimum support count to be frequent in D.
$\delta' = \lceil min\_sup \times \lvert D' \rvert \rceil$	Minimum support count to be frequent in D'.
S_P^{db}	Additional support occurred due to the addition of db.
S_P^D	Support of P in D.
$S_P^{D'} = S_P^D + S_P^{db}$	Complete support of P in D'.
$C(N)$	*present_count* attribute of node N
$C'(N)$	*previous_count* attribute of node N
N_P^M	Set of modified projection nodes for pattern P
N_P^U	Set of unmodified projection nodes for pattern P for pattern P
$N_P = N_P^M \cup N_P^U$	Complete set of projection nodes for pattern P
T_P^M	Total support calculated from the modified nodes, $\sum C(n); n \in N_P^M$
T_P^U	Total support calculated from the unmodified nodes, $\sum C(n); n \in N_P^U$
$(*)^+$	+ signed variables are used for intermediary calculations, starting from the upper bound.

the summation of modified nodes' changed support that ultimately gives the additional occurred support from the last mined iteration. Changed support of a node means, the difference between the *previous_count* and *present_count* attribute's value of that node. $S_P^{D'}$ can also be visualized as the support ($\sum present\_count$) calculated from the unmodified ($total\_sup_P^U$) and modified ($total\_sup_P^M$) projection nodes to calculate the final updated support. We have also shown the calculation of heuristic support($heuristic\_support_P$) in the figure. We have already mentioned in the earlier section that, heuristic support is the summation of the nodes' count

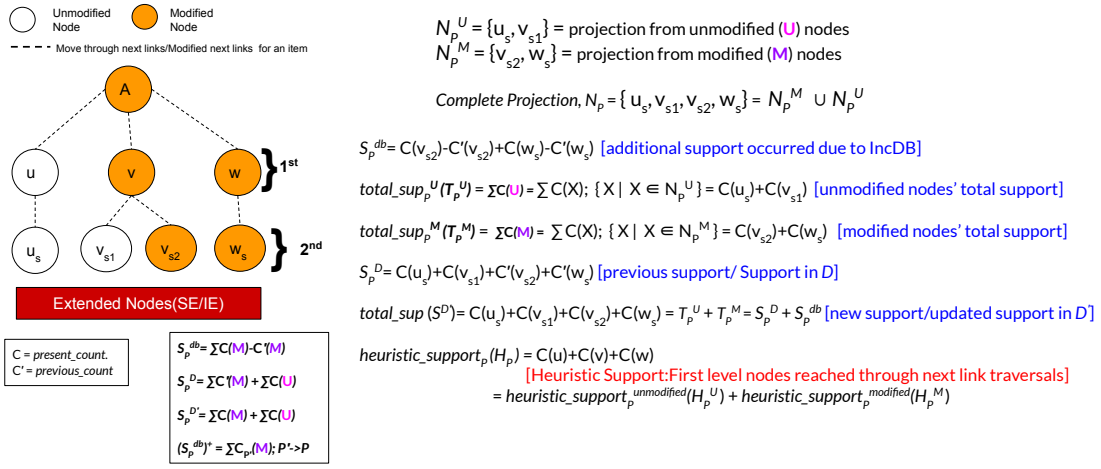


FIGURE 3.24: Mining Concepts of IncTree-Miner

attributes' values reached through first level next link traversals. During incremental mining we can also calculate this support ($heuristic\_support_P$) by merging the heuristic support from the unmodified ($heuristic\_support_P^U$) and modified ($heuristic\_support_P^M$) subtrees separately maintaining the rule of considering the first level nodes reached through next links. In this discussion, we have used $(*)^+$ signed variables to denote the upper bound during intermediary calculations.

Now, we will talk about an important property which controls the projection for a pattern in the unmodified subtrees.

Definition 3.8 (Infrequent to Frequent Transition Property). Suppose, we had an infrequent pattern P (up to previous pass), minimum support threshold 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 incremental database 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.

This property gives us an indication that, if a previously infrequent pattern has a chance to become frequent after this current iteration or not. If this property does not satisfy, then we can simply stop searching for a pattern in the unmodified

subtrees. For example, suppose a pattern $<(\alpha)(\beta)>$ was infrequent in the previous old database D where the minimum support requirement to be frequent was 2 (δ). As, $<(\alpha)(\beta)>$ was infrequent, its previous maximum support could be 1 ($\delta - 1$). Suppose now the previous database D has been updated to D' and the minimum support requirement to be frequent has been increased to 5 (δ'). Now, if $<(\alpha)(\beta)>$ wants to become frequent, then it has to achieve at least an additional support amount of 4 ($\delta' - \delta + 1$) from the modified subtrees or incremental database. For the initial pass as the old database was empty, the constraint is simply $\geq \delta'$.

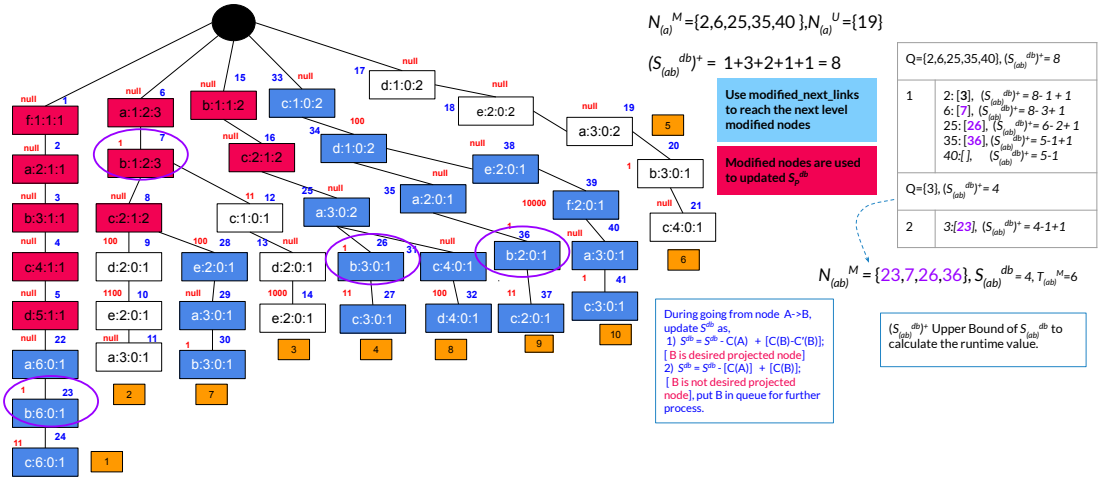


FIGURE 3.25: Modified Projection of $<(ab)>$

In Figure 3.25 and Figure 3.26 we have shown two examples of how to perform modified projections (N_P^M) for a pattern and update the additional support (S_P^{db}) in gradual manner through Breadth-First based support counting technique.

In Figure 3.27 and 3.28 we have shown two examples to perform unmodified projections (N_P^U). We perform unmodified projection for a pattern only when it is necessary. During performing unmodified projection for a pattern we try to discover only those nodes which project the pattern and are unmodified. An important crucial part here is, there can be unmodified projection node beneath a modified node. So, to find the unmodified projection nodes we may also have to see in the subtrees of a modified node.

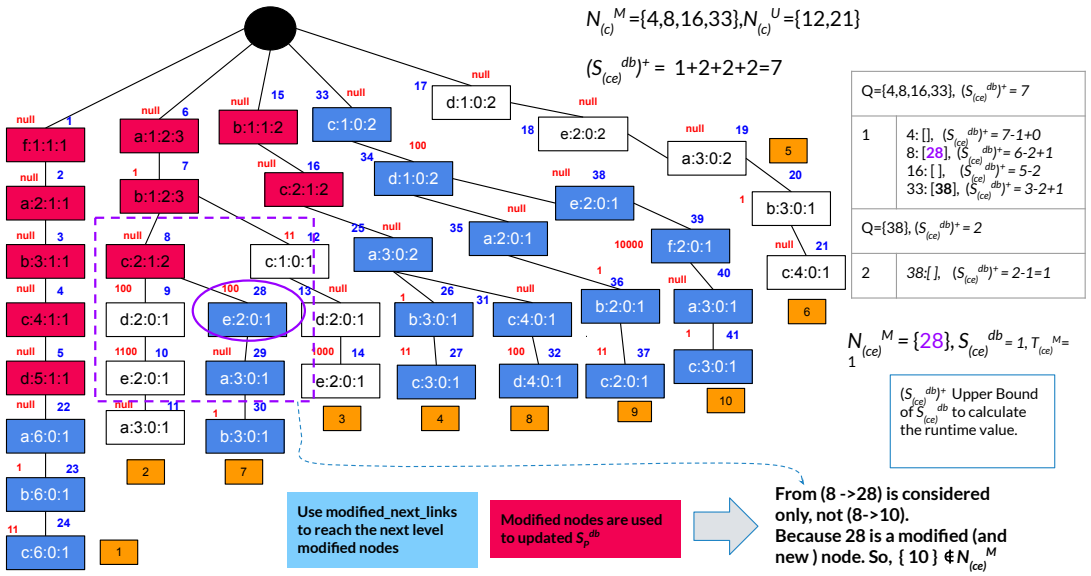


FIGURE 3.26: Modified Projections of $<(ce)>$

During searching for unmodified projection nodes in the subtrees of a modified node, we can use a set of optimizations which will help to reduce searching in that subtree by detecting that there is no unmodified node beneath, through simple but efficient observations.

Proposition 3.12.1 (Optimization 1). Suppose, we want to find the unmodified projection nodes from V for γ. Now, if $C(V) == |V.modified\_next\_link(\gamma)|$, then, there is no need to search for unmodified nodes for γ from V.

This will work because, all the underneath branches of V are modified. E.g., During Finding $N^{U}_{(ac)}$ or $N^{U}_{(a)(c)}$ from node 25, we do not need to look for c. Because, $C(25) = 2$ and $|25.modified\_next\_link(c)| = |\{27, 31\}| = 2$.

Proposition 3.12.2 (Optimization 2). Suppose, we want to find the unmodified projection nodes from V for γ. If V is a modified node and $C(V) = 1$, then we do not need to search for unmodified nodes for γ from V.

All the underlying nodes are modified here and arrived to a single branch. E.g., During Finding $N^{U}_{(ce)}$, we actually do not need to look from node 38 for e underneath.

For simplicity and to understand the projection concept, in Figure 3.27 and 3.28 we did not perform any optimizations during searching for the unmodified projection nodes. But as per discussion, a good amount of optimizations can be performed while searching for unmodified projection nodes beneath a modified node. And, from the construction mechanism it can be understood that, all the underlying nodes beneath an unmodified node are unmodified.

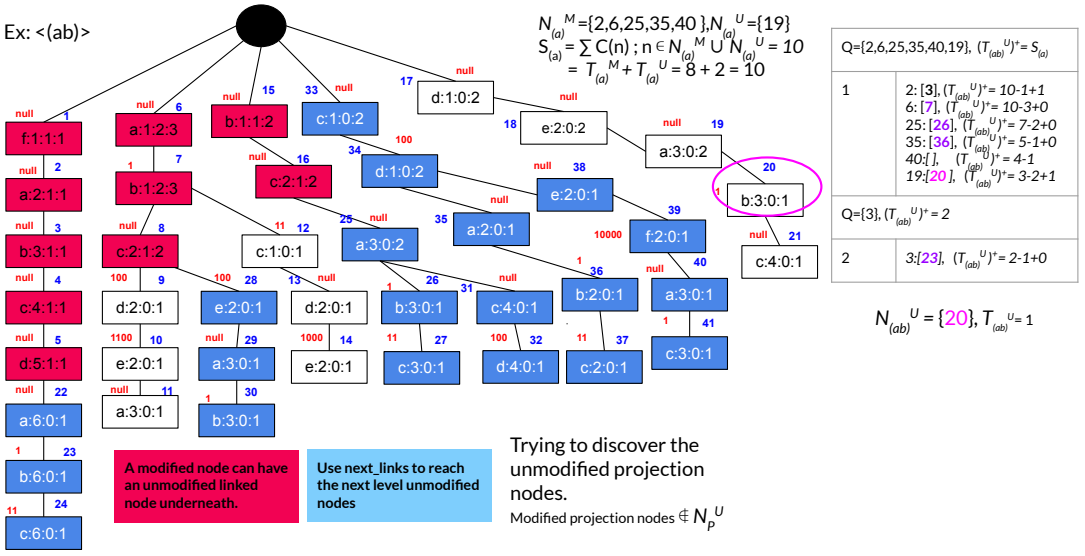


FIGURE 3.27: Unmodified Projection of $<(ab)>$

Based on the results of Figure 3.25, 3.27 and 3.26, 3.28, we can find the full picture for the patterns $<(ab)>$ and $<(ce)>$ respectively. The merge result is shown in Table 3.12. We have also shown an example of NIB to understand the essence of when the support of an infrequent pattern is stored in NIB of a node in BPFSP-Tree in Figure 3.29. This support is stored to mitigate the cost of some infrequent pattern's searching by giving the complete support of the pattern up to last iteration to merge and then to calculate the complete picture.

Now, based on all the aforementioned concepts, we will provide a discussion of our incremental mining algorithm, *IncTree-Miner* through the pseudo code of Algorithm 6. But before diving into the discussion of detailed workflow Of IncTree-Miner algorithm, we will provide a summarized and high level overview of

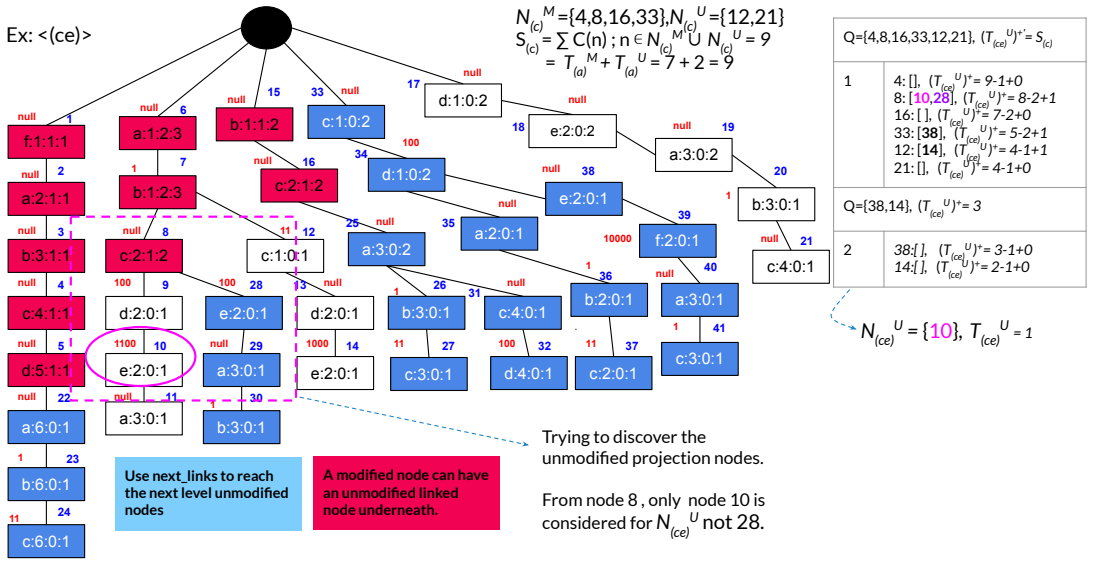


FIGURE 3.28: Unmodified Projection of $<(ce)>$

TABLE 3.12: Merging Results of modified and unmodified projections

P	N_P	$S_P^{D'}$	S_P^{db}
$<(ab)>$	$= N^M_{(ab)} \cup N^U_{(ab)}$ $= \{23, 7, 26, 36\} \cup \{20\}$ $= \{23, 7, 26, 36, 20\}$	$= T^M_{(ab)} + T^U_{(ab)}$ $= 6 + 1$ $= 7$	4
$<(ce)>$	$= N^M_{(ce)} \cup N^U_{(ce)}$ $= \{28\} \cup \{10\}$ $= \{28, 10\}$	$= T^M_{(ce)} + T^U_{(ce)}$ $= 1 + 1$ $= 2$	1

the algorithm. In Figure 3.30, we have shown the logical block representation of the algorithm. There, we have shown the three cases which IncTree-Miner checks to find the affected patterns. Similar to Tree-Miner, IncTree-Miner also performs projections using Breadth-First Based Support Counting Technique through upper bounding over the maximum possible support. But its projection process is divided into two parts, modified and unmodified projection. In each case, first it performs projection in the modified subtrees and based on that, it decides to perform projection in the unmodified subtrees. As in general scenarios, the size of the incremental database is comparatively much smaller than the existing database, the size of modified subtrees will be smaller compared to the unmodified subtrees.

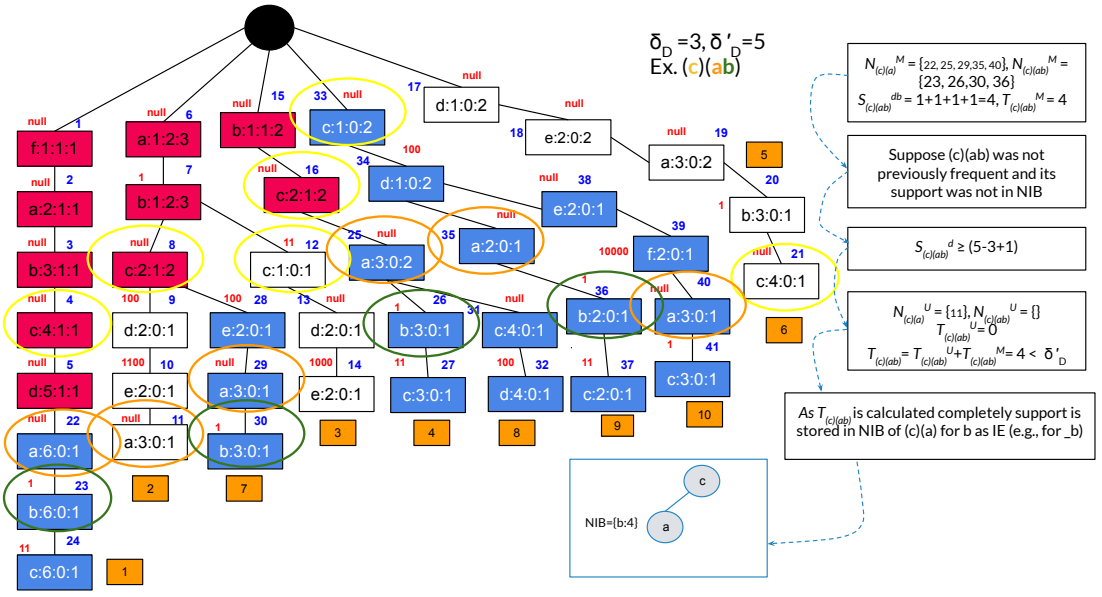


FIGURE 3.29: An Example of NIB for pattern $(c)(ab)$ with $\delta = 3$ and $\delta' = 5$

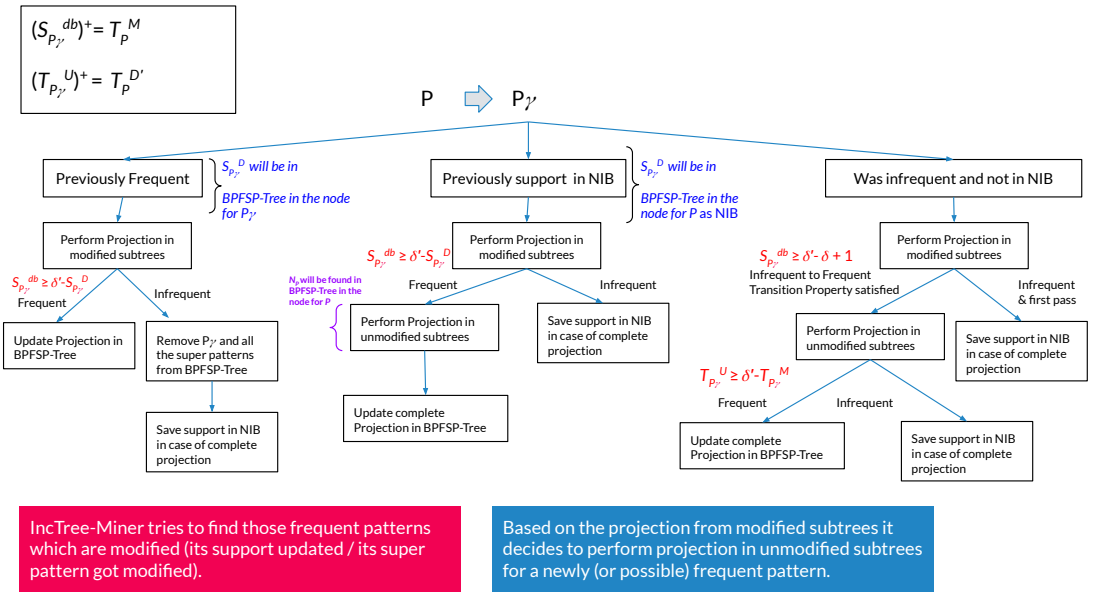


FIGURE 3.30: Logical Block Representation of IncTree-Miner Algorithm

So, in most of the cases the current status of the patterns will be detected through performing a small size projection. In the figure we have also shown the initial upper bound values for the extended pattern $P\gamma$ during different partial support

calculations $(S^{db}_{P\gamma})^{+}(\geq S^{db}_{P\gamma})$ and $(T^{U}_{P\gamma})^{+}(\geq T^{U}_{P\gamma})$. We have also shown the usage of BPFSP-Tree during the calculation.

Algorithm 6 IncTree-Miner

procedure PatternFormation(Pattern $P\gamma$, N^M_P,BPFSP-Tree nodeB_P)
  if ($P\gamma \in B_P.child$) **then** ▷ previously frequent, $B_{P\gamma}$ node exists
    ∀n $\in N^M_P$ perform implicit projection in modified subtrees for γ
    if ($S^{D'}_{P\gamma} \geq \delta'$) **then** update support and projection in $B_{P\gamma}$.
    else Remove $B_{P\gamma}$ and its subtree, adjust end-links. ▷ Got Infrequent
      if (complete projection done) **then**
        $B_P.NIB[\gamma] = S^{D'}_{P\gamma}$ ▷ Save support in NIB
  else if ($\gamma \in B_P.NIB$) **then** ▷ Previous complete support in NIB
    ∀n $\in N^M_P$ perform implicit projection in modified subtrees for γ.
    if ($S^{D'}_{P\gamma} \geq \delta'$) **then** Perform projection in unmodified subtrees for γ.
      Create node $B_P.child[\gamma]$, adjust end-links, remove $B_P.NIB[\gamma]$
    else ▷ Still Infrequent
      if (complete projection done) **then** $B_P.NIB[\gamma] = S^{D'}_{P\gamma}$
      else Remove $B_P.NIB[\gamma]$ ▷ Not full projection calculated
  else if ($P\gamma \notin B_P.child \cap \gamma \notin B_P.NIB$) **then**
    ∀n $\in N^M_P$ perform implicit projection in modified subtrees for γ.
    if ($S^{db}_{P\gamma} \geq \delta' - \delta + 1$) **then** ▷ Infrequent to Frequent Transition
      Perform projection in unmodified subtrees for γ.
      if ($S^{db}_{P\gamma} + S^{D}_{P\gamma} = S^{D'}_{P\gamma} \geq \delta'$) **then** Create $B_P.child[\gamma]$,adjust links
      else ▷ Infrequnet Pattern
        if (complete projection done) **then** $B_P.NIB[\gamma] = S^{D'}_{P\gamma}$
        if ($H^M_{P\gamma} + H^U_{P\gamma} < \delta'$) **then** ▷ Heuristic Pruning during SE
          Remove γ from $iList$ of pattern P
  if ($P\gamma$ is frequent $\cap$ $N^M_{P\gamma} \neq \{\}$) **then** Return $N^M_{P\gamma}$ ▷ modified projection
    ▷ $P\gamma$ has update in subtree, need further checking
  else Return *null* ▷ infrequent/no update in underlying subtree for $P\gamma$
procedure IncTreeMiner(Pattern P,N^M_P,$sList^M$,$iList^M$,δ,δ',B_P)
  Reduce $sList^M$ and $iList^M$ based on CETable
  $N_{SE}, N_{IE} = \{\}, \{\}$ ▷ Next iteration nodes
  for each item $\gamma \in sList^M$ **do** ▷ SE, Tracking the modified patterns
    $N_{SE}[\gamma]$ = PatternFormation($P\{\gamma\}$,N^M_P, B_P);[iff $N^M_{P\{\gamma\}} \neq \{\}$]
    if ($N^M_{P\{\gamma\}} = \{\}$) **then** $sList^M - \{\gamma\}$ ▷ Infrequent/No modifications
  for each item $\gamma \in iList^M$ **do** ▷ IE, Tracking the modified patterns
    $N_{IE}[\gamma]$ = PatternFormation($\{P\gamma\}$,N^M_P, B_P);[iff $N^M_{\{P\gamma\}} \neq \{\}$]
    if ($N^M_{\{P\gamma\}} = \{\}$) **then** $iList^M - \{\gamma\}$ ▷ Infrequent/No modifications
  Remove all γ from $B_P.NIB$ [if $P\gamma$ is not projected] ▷ For Consistency
  $sList' = sList^M$,$iList' = iList^M$ ▷ Reduced lists
  for each modified item $\gamma \in N_{SE}$ **do** ▷ Recursive SE for effected patterns
    IncTreeMiner($P\{\gamma\}, sList', \{\epsilon | \epsilon \in sList' \cap \epsilon > \gamma\}, \delta, \delta', B_{P\{\gamma\}}$).
  for each modified item $\gamma \in N_{IE}$ **do** ▷ Recursive IE for effected patterns
    IncTreeMiner($\{P\gamma\}, sList', \{\epsilon | \epsilon \in iList' \cap \epsilon > \gamma\}, \delta, \delta', B_{\{P\gamma\}}$).
procedure RemoveInfrequentPatterns
  Traverse through *end-link*s of leaf nodes in BPFSP-Tree and remove the infrequent patterns ($S^{D'}_P < \delta'$) in bottom up manner.

Our IncTreeMiner function is called with a pattern P, its modified projection nodes N^M_P, its corresponding lists $sList^M$ and $iList^M$, previous minimum

support requirement δ, current minimum support requirement δ' and its corresponding node B_P in BPFSP-Tree which represents the pattern. Our algorithm IncTree-Miner considers only those patterns which have modified projection nodes and recursively tries to discover its modified or affected super patterns from the underlying modified subtrees. So, the corresponding $sList^M$ and $iList^M$ contain those symbols or items which are modified or affected due to the addition of incremental database and might extend P.

In line 28, we perform CETable based pruning on $sList^M$ and $iList^M$. Then, in line 29, we initialize the variables (N_{SE}, N_{IE}) to save the next iteration modified nodes for recursive pattern extension or projection. In line (30-32) we perform the sequence extension for each symbol $\gamma \in sList$. Similarly in line (33-35) we perform the itemset extension for each symbol $\gamma \in iList$. We call PatternFormation function (line 31 and 34) with the extended pattern $P\gamma$ ($P\{\gamma\}$ or $\{P\gamma\}$), N_P^M and B_P. PatternFormation function check the status (frequent or infrequent) of $P\gamma$ and returns the modified projected nodes ($N_{P\{\gamma\}}^M$ or $N_{\{P\gamma\}}^M$) in line 31 and 34. This function can also return an empty projection nodes' list ($N_{P\{\gamma\}}^M = \{\}$ or $N_{\{P\gamma\}}^M = \{\}$) if the extended pattern is detected as infrequent or unaffected after the addition of incremental database having no modified projection nodes. So, based on the returned empty lists the symbol or item is removed from the corresponding list (line 32 and 35). In line 36, we remove all the symbols γ from the Non-Frequent Item Buffer (NIB) for which pattern extension is not checked to maintain the support consistency over successive iterations. In lines (38-39) and (40-41) we perform recursive sequence and itemset extension respectively for the modified or affected patterns with their corresponding updated $sList'$ and $iList'$ based on the pruning strategy stated in section 3.4.3.

Now, we will talk about PatternFormation function. This function is called with an extended pattern $P\gamma$, the modified projection nodes of P (N_P^M) and the corresponding BPFSP-Tree node of P (B_P) to check the current status of $P\gamma$. There are basically three logical blocks in the function to check the status of

$P\gamma$. First block (lines 2-7) works if $P\gamma$ was previously frequent in the last mined iteration. Second block (lines 8-14) works if $P\gamma$'s previous complete support was saved in NIB buffer. Third block works if $P\gamma$ was neither previously frequent nor its complete support was saved in NIB.

In the first block, first we perform implicit projection in the modified subtrees for γ for all $\forall n \in N_P^M$ through modified next links (line 3). Our projection and support counting mechanism always works based on the proposed Breadth-First Support Counting Technique. If the calculated total support of $P\gamma$, $S_{P\gamma}^{D'}$ satisfies the current minimum support requirement, δ', then we can say that $P\gamma$ is still frequent and we update its support and projection information in the corresponding BPFSP-Tree node, $B_{P\gamma}$ (line 4). As, $P\gamma$ was previously frequent, so we will have a corresponding node for it in BPFSP-Tree ($B_{P\gamma}$). But if $S_{P\gamma}^{D'}$ fails to satisfy δ' then we can say that $P\gamma$ has become infrequent in D' (line 5). Then we can remove $B_{P\gamma}$ along with its underlying subtree from the BPFSP-Tree and adjust the leaf nodes' end-links (line 5). As, our proposed breadth-first based support counting technique can early identify a pattern's infrequency, it may very early stop performing projection and return. But if it has performed complete projection for an infrequent pattern, then we can store this support in NIB for using in successive mining iterations (line 6-7).

Now, we will discuss the second block where the previous support was saved in NIB (line 8). Similar to first block, first we perform projection for γ in modified subtrees (line 9). Then we get the total updated support $S_{P\gamma}^{D'}$ ($S_{P\gamma}^{D}$ from NIB and $S_{P\gamma}^{db}$ from modified subtrees, $S_{P\gamma}^{D'} = S_{P\gamma}^{D} + S_{P\gamma}^{db}$) and based on it we identify $P\gamma$ has become frequent or not (line 10). If $S_{P\gamma}^{D'} \geq \delta'$ then to get the complete projection nodes for $P\gamma$ we perform projection in the unmodified subtrees $\forall n \in N_P^M \cup N_P^U$ through next link for γ. The result of this action is, we find a set of unmodifed nodes which project $P\gamma$. Then, we create a new entry for $P\gamma$ in BPFSP-Tree, ($B_P.child[\gamma] = B_{P\gamma}$), save the complete projection nodes with updated support and adjust end-links (line 11). If $S_{P\gamma}^{D'} < \delta'$, then we can say that $P\gamma$ is still infrequent

(line 12). Now, if we have performed complete projection for $P\gamma$ through Breadth-First support counting, then we can store the support in NIB (line 13). Otherwise we remove the entry from NIB to maintain the support consistency over successive mining iterations (line 14).

Now, we will talk about the third logical block (line 15-23). Here first we perform implicit projection in the modified subtrees for γ and get the support $S^{db}_{P\gamma}$ (line 16). If $S^{db}_{P\gamma}$ satisfies the infrequent to frequent transition property (line 17) then we understand that $P\gamma$ has chance to become a frequent pattern and perform projection in the unmodified subtrees for γ (line 18). Now, if $S^{D'}_{P\gamma}$ satisfies the minimum support requirement then we can say $P\gamma$ has become a new frequent pattern and we create a corresponding node in BPFSP-Tree ($B_{P\gamma}$) and adjust the end-links (line 19). But if $S^{D'}_{P\gamma}<\delta'$, then we can say that $P\gamma$ is not a frequent pattern (line 20) and update its complete support $S^{D'}_{P\gamma}$ in NIB if we have performed complete projection (line 21). line 22-23 is applicable only during sequence extension to perform heuristic pruning discussed in section 3.4.4. We calculate the heuristic support from the unmodifed ($H^{U}_{P\gamma}$) and modified ($H^{M}_{P\gamma}$) subtrees separately and check against δ' (line 22). If the condition fails then we can simply say that $\{P\gamma\}$ will never be frequent and can be removed from the corresponding $iList$ of P (line 23).

Now, if we have detected $P\gamma$ as frequent having modified projection nodes then, we return the modified projection nodes to IncTreeMiner function to further check in the corresponding underlying modified subtrees to discover the super patterns of $P\gamma$ which are affected due to the addition of incremental database (line 24). Otherwise we return an empty or null list to indicate that, we do not need to perform any extension from $P\gamma$ (line 26). The pseudo-code is written in such a way so that the logical blocks can easily be understood. During implementation, we may need to bring some implementational strategy to distinguish the actions between sequence and itemset extension.

After finding all the patterns which had modification or update in their projection nodes or underlying subtrees, we start to remove the unchanged previously frequent but currently infrequent patterns in bottom up manner traversing through the end-links of leaf nodes using RemoveInfrequentPatterns function (line 41-42). In our discussion, we have mentioned multiple times regarding adjusting the end-links of leaf nodes. This action is done basically to maintain the consistency so that we can traverse through the leaf nodes. There are basically two types of cases which can be encountered -

1. *A new leaf node is created:* In this case, we have to create an end-link from the last created leaf node to this newly created leaf node. A leaf node in the BPFSP-Tree can be created in two ways, either a completely new node is created for a new frequent pattern or an old non-leaf node looses all of its underlying nodes because of their transition from frequent to infrequent.

2. *A leaf node is removed:* Suppose, we have three leaf nodes, V_1, V_2 and V_3 having end-links from $V_1 \rightarrow V_2$ and $V_2 \rightarrow V_3$. Suppose, V_2 is removed as a leaf node. Then to maintain the end-link consistency, we have to remove previous end-links from $V_1 \rightarrow V_2$ and $V_2 \rightarrow V_3$ and create a new end-link from $V_1 \rightarrow V_3$. There are two scenarios when a leaf node is removed from the BPFSP-Tree. First one is, if a pattern ending at a leaf node becomes infrequent. Then this pattern or the corresponding leaf node will be removed from the tree. Second one is, if a pattern ending at a leaf node gets extended. Then, its super patterns (having the pattern as prefix) will be attached underneath and this node will become a non leaf node. So, then we need to remove the end-link form V_2. There are some trivial cases, such as V_3 or V_1 is null. In these cases, we just have to adjust the ending or starting end-link pointer respectively to maintain the consistency through end-link traversals.

Now, we will present a small simulation to understand how IncTree-Miner discovers the patterns that are affected due to the addtion of the incremental database. IncTree-Miner reports those affected patterns (which are also frequent)

after complete mining. Similar to Tree-Miner, for each pattern P we have two corresponding lists $sList^M$ and $iList^M$, but here they contain the modified symbols that may have affected the super patterns of P due to the new database addition. $sList'$ and $iList'$ are found after applying different sets of pruning strategies and contain the modified symbols which have extend P as frequent. In Table 3.13, we have shown the patterns that are affected due to the addition of the incremental database and start with prefix c. Our IncTree-Miner discovers these patterns.

TABLE 3.13: Simulation of IncTree-Miner for the modified patterns with prefix $<(c)>$, $\delta = 2$ and $\delta' = 3$

P	N_P^M	N_P^U	S_P^D	$S_P^{D'}$	$sList^M$ $iList^M$	$sList'$, $iList'$
$<(c)>$	$\{4, 8, 16, 33\}$	$\{12, 21\}$	9	5	$sList^M = \{a, b, c, e\}, iList^M = \{d, e\}$	$sList' = \{a, b, c, d\}, iList' = \{d\}$
$<(c)(a)>$	$\{22, 25, 29, 35, 40\}$	$\{11\}$	7	1	$sList^M = \{a, b, c, d\}$ $iList^M = \{b, c, d\}$	$sList' = \{\}$ $iList' = \{b, c\}$
$<(c)(ab)>$	$\{23, 26, 36\}$	$\{\}$	3	0	$sList^M = \{\}$ $iList^M = \{c\}$	$sList' = \{\}$ $iList' = \{c\}$
$<(c)(abc)>$	$\{\}$	$\{\}$	3	0	$sList^M = \{\}$ $iList^M = \{\}$	$sList' = \{\}$ $iList' = \{\}$
$<(c)(ac)>$	$\{24, 27, 37, 41\}$	$\{\}$	4	0	$sList^M = \{\}$ $iList^M = \{\}$	$sList' = \{\}$ $iList' = \{\}$
$<(c)(b)>$	$\{23, 30, 26, 36\}$	$\{\}$	4	0	$sList^M = \{a, b, c, d\}$ $iList^M = \{c, d\}$	$sList' = \{\}$ $iList' = \{c\}$
$<(c)(bc)>$	$\{24, 27, 37\}$	$\{\}$	3	0	$sList^M = \{\}$ $iList^M = \{\}$	$sList' = \{\}$ $iList' = \{\}$
$<(c)(c)>$	$\{24, 27, 31, 37,$	$\{\}$	5	0	$sList^M = \{a, b, c, d\}, iList^M = \{d\}$	$sList' = \{\}$ $iList' = \{\}$

	$41\}$					
$<(c)(d)>$	$\{5, 31\}$	$\{13\}$	3	2	$sList^M = \{a, b, c,$ $d\}, iList^M = \{\}$	$sList' = \{\}$ $iList' = \{\}$
$<(cd)>$	$\{34\}$	$\{9\}$	3	1	$sList^M = \{a, b, c,$ $d\}, iList^M = \{\}$	$sList' = \{a\}$ $iList' = \{\}$
$<(cd)(a)>$	$\{35, 40\}$	$\{11\}$	3	1	$sList^M = \{a\}$ $iList^M = \{\}$	$sList' = \{\}$ $iList' = \{\}$

Now, we will talk about the extendibility of IncTree-Miner from various aspects. IncSP-Tree provides structural advantage to perform faster pattern search in the database. So, it can also help to improve mining performance in buffer based solutions by performing faster pattern generation. Moreover we can also develop a memory resilient IncTree-Miner by not keeping the projection information and performing runtime necessary pattern search using IncSP-Tree. A short visualization of memory resilient version is shown in Figure 3.31.

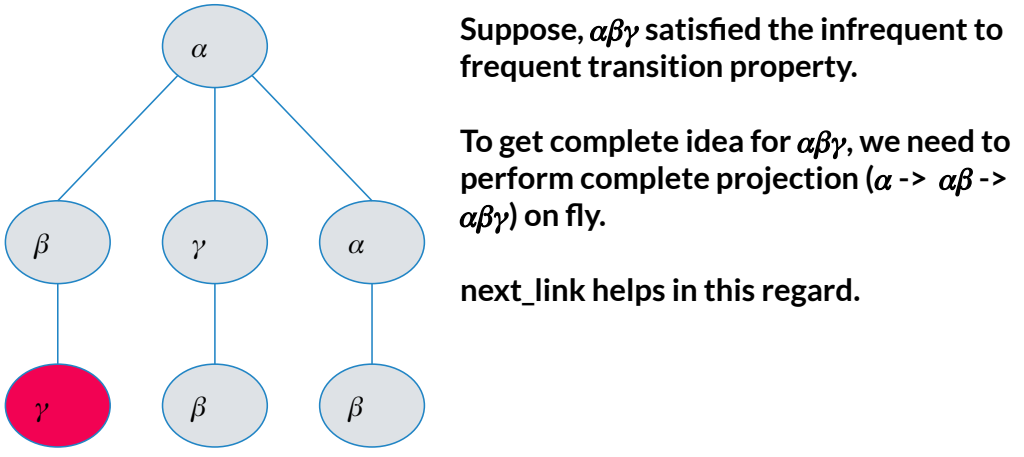


FIGURE 3.31: Memory Resilient IncTree-Miner

In Figure 3.31, we have shown a pattern storage structure which does not keep any projection information rather only support value. So, it reduces memory to a great extent. In this memory resilient version, we perform faster projection for

the necessary patterns on fly using next links which helps to traverse faster in the database for a symbol or an item resulting in faster pattern generation. We can also embed some buffer based concepts in this regard to reduce the number of tree scans by performing some additional pre-computation. In summary, our designed IncTree-Miner based on IncSP-Tree can be extended into multiple directions.

3.13 Complexity Analysis of Incremental Solution

In Section 3.7, we have provided a complexity analysis of our proposed static solution, *Tree-Miner* based on *SP-Tree*. Our incremental solution, *IncTree-Miner* based on *IncSP-Tree* is an extended version of our static solution. So, it shares the backbone of the static solution, e.g., tree structure and supporting structure similarities, pruning strategies, pattern formation concepts, etc. including some new strategies to efficiently address the incremental mining problem. In Table 3.14, we have shown the variables which we have used in this section. We also introduce them during further discussion and analysis.

Here, the main addition is division of modified (M) and unmodified nodes (U). Lets assume, old database was D, incremental database is db and updated database is D'. After database update, suppose, the number of nodes of IncSP-Tree got increased from N to $N'(N' \geq N, N' = |U| + |M|)$. Modified nodes are divided into two parts, newly added modified nodes (M_{new}) and existing updated modified nodes (M_{old}) and $M = M_{old} \cup M_{new}$. As, in general and average case scenarios $|db|<|D|$, so $|M|<|N|<|N'|$ and $|M| = \sum_{i=1}^{|db|}(L_i^D + L_i^{db})$ where L_i^D and L_i^{db} denotes the length of sequence s_i ($sid = s_i$) in old and incremental database respectively.

$modified\_next\_link$ is always a subset of $next\_link$ for each corresponding node. In Section 3.7, we have shown the upper bound estimation for $next\_link$ space requirement which is $\mathcal{O}(u_i \times N')$ (as N has got increased to N'). But in

TABLE 3.14: Variables used in complexity discussion: incremental solution

Name	Variables
Old Database	D
Incremental Database	db
Updated Database	D'
i^{th} sequence ($sid = i$)	s_i
Length of s_i in D	L_i^D
Length of s_i in db	L_i^{db}
Number of nodes of IncSP-Tree	N'
Modified nodes of IncSP-Tree	M
Unmodified nodes of IncSP-Tree	U
Average number of unique symbols in the subtree of each node	u_i
Average number of modified unique symbols in the subtree of each modified node	$\widetilde{u}$
Total number of unique symbols	I
Average number of unique symbols for each sequence of D	k
Average number of unique symbols for each sequence of db	$\widetilde{k}$
Pattern	P
Projection nodes of P Modified Projection nodes of P Unmodified Projection nodes of P	N_P N_P^M N_P^U
Support of P in D' Total support of P from N_P^M Total support of P from N_P^U	S_P T_P^M T_P^U
Total set of tested patterns Total number of frequent patterns	Z Z_F
Average Projection cost	S^*

average case scenarios, the space requirement for *next_link* is smaller compared to $\mathcal{O}(u_i \times N')$, so is $modified\_next\_link$. Also, for $modified\_next\_link$, the complexity will be $\mathcal{O}(\widetilde{u} \times M)$. Because, we calculate the $modified_next_link$s only for the modified nodes M (not for full N'). So, the average number of unique symbol also may reduce to $\widetilde{u}$ ($\widetilde{u} \leq u_i$).

The memory complexity of BPFPS-Tree is dependent on the number of frequent patterns (Z_F) generated. So, if the average support value for the patterns is S^* then the upper bound of the memory complexity here will be $\mathcal{O}(Z_F \times$

TABLE 3.15: Incremental solution's upper bound space complexity

Structural Elements	Upper Bounds
Number of modified nodes of IncSP-Tree	$\|M\| = \sum_{i=1}^{\|db\|}(L_i^D + L_i^{db})$
Total number of nodes of IncSP-Tree	$N' = \|U\| + \|M\|$
Total Cost related to *next_link*	$\mathcal{O}(u_i \times N')$
Total Cost related to $modified\_next\_link$	$\mathcal{O}(\widetilde{u} \times M)$
Co-Existing Item Table	$\mathcal{O}(I^2)$
Sequence Summarizer	$\mathcal{O}(\|D'\| \times k)$
BPFPS-Tree with projection information	$\mathcal{O}(Z_F \times S^*)$ $S^* << \frac{S_{P_1}+S_{P_2}+S_{P_3}+...+S_{P_{Z_F}}}{Z_F}$
BPFPS-Tree with only support	$\mathcal{O}(Z_F)$

S^*). But, due to the node overlapping characteristic of IncSP-Tree, $S^* << \frac{S_{P_1}+S_{P_2}+S_{P_3}+...+S_{P_{Z_F}}}{Z_F}$. In the memory resilient IncTree-Miner, we do not store the projection information, so the space complexity reduces to $\mathcal{O}(Z_F)$. In Table 3.15, we have shown the summary of above space complexity discussion with respect to upper bounds.

Now, we will talk about the runtime complexity. Similar to previous complexity analysis section, the discussion will be divided into two parts, pre-processing complexity and mining complexity.

To insert the all the sequences $s_i \in db$ we need to visit $\sum_{i=1}^{|db|} L_i^{db}$ number of nodes. We try to update the nodes' attributes after each *Insert* or *Append* operation. So, based on the previous variable setup arrangement if the average number of unique symbol for each modified node is $\widetilde{u}$, then the upper bound complexity to update the nodes' attributes is $\mathcal{O}(\widetilde{u} \times \{\sum_{i=1}^{|db|} L_i^{db} + L_i^D\})$. But, as an already created node does not need to be tracked again and similarly an already tracked modified node does not need to be tracked as modified again, the average complexity is much smaller than $\mathcal{O}(\widetilde{u} \times \{\sum_{i=1}^{|db|} L_i^{db} + L_i^D\})$. Now, lets assume the average number of unique symbols in each sequence $s_i \in D'$ is k and the average number of unique symbols in each sequence $s_i \in db$ is $\widetilde{k}$. So, the upper bound complexity to update existing $CETable$ is $\mathcal{O}(|db| \times k \times \widetilde{k})$. But, as the number of unique symbols for a sequence is completely dependent on that sequence's pattern,

so the actual complexity to update $CETable$ is much smaller than $\mathcal{O}(|db| \times k \times \widetilde{k})$. In summary, the upper bound of the pre-processing complexity will be $\sum_{i=1}^{|db|} L_i^{db}$ $+ \mathcal{O}(\widetilde{u} \times \{\sum_{i=1}^{|db|} L_i^{db} + L_i^D\}) + \mathcal{O}(|db| \times k \times \widetilde{k})$.

Now, we will discuss the mining complexity. In Section 3.7, we have shown the upper bound complexity of projecting an extended pattern $P\gamma$ from a pattern P is $O(S_{P\gamma})$. IncTree-Miner shares this concept but it divides the projection into two parts modified $(N_{P\gamma}^M, S_{P\gamma}^{db}, T_{P\gamma}^M)$ and unmodified projection $(N_{P\gamma}^U, S_{P\gamma}^D, T_{P\gamma}^U)$. So similar to the previous discussion of Tree-Miner, the upper bound complexity to perform modified projections for a pattern $P\gamma$ will be $\mathcal{O}(T_{P\gamma}^M)$. As, $T_{P\gamma}^M = \sum_{n \in N_P^M} C(n)$ where C denotes the present count attribute's value. Similarly to perform unmodified projections for a pattern the upper bound complexity will be $\mathcal{O}(T_{P\gamma}^U)$.

IncTree-Miner's speciality is that, it first decides to perform projection in the modified subtrees for a pattern and based on that it decides to perform projection in the unmodified subtrees. In Table 3.16, we have shown the upper bound projection complexities during different types of pattern transitions. IncTree-Miner checks the modified nodes or modified patterns and resolves the transition cases. To mitigate the costs, it also uses the BPFSP-Tree, NIB, Bottom-up previously frequent but currently infrequent pattern removal along with other strategies. Also, our incremental solution adopts all the prior pruning strategies including Breadth-First support counting technique. So, the complexities shown in Table 3.16 are very high level upper bounds and lots of infrequent patterns can be detected very early. To search for the modified projection nodes, we use modified next links and to search for the unmodified projection nodes we use next links along with the optimizations discussed in Section 3.12.

Now, lets assume, all the tested modified patterns is in Z. So, Z will contain patterns tested from different types of transitions shown in Table 3.16 along with a group of patterns (Z_{Prune} and Z_{Bottom}) which were directly pruned using $CETable$, Heursitic strategy, mutual information (contained in Z_{Prune}) and

TABLE 3.16: Projection runtime Complexities during different types of pattern transitions from P to $P\gamma$

Serial	*Old Status in D*	Infrequent to frequent property satisfied	New status in D'	Total Cost
1	Frequent	Yes	Frequent	$\mathcal{O}(\mathrm{T}^{M}_{P\gamma})$
2	Frequent	No	Infrequent	$\mathcal{O}(\mathrm{T}^{M}_{P\gamma})$
3	Infrequent ($S^{D}_{P\gamma} \in NIB$)	No	Infrequent	$\mathcal{O}(\mathrm{T}^{M}_{P\gamma})$
4	Infrequent ($S^{D}_{P\gamma} \in NIB$)	Yes	Infrequent	$\mathcal{O}(\mathrm{T}^{M}_{P\gamma})$
5	Infrequent ($S^{D}_{P\gamma} \in NIB$)	Yes	Frequent	$\mathcal{O}(\mathrm{T}^{M}_{P\gamma})$+O($T^{U}_{P\gamma}$)
6	Infrequent ($S^{D}_{P\gamma} \notin NIB$)	No	Infrequent	$\mathcal{O}(\mathrm{T}^{M}_{P\gamma})$
7	Infrequent ($S^{D}_{P\gamma} \notin NIB$)	Yes	Infrequent	$\mathcal{O}(\mathrm{T}^{M}_{P\gamma})$+O($T^{U}_{P\gamma}$)
8	Infrequent ($S^{D}_{P\gamma} \notin NIB$)	Yes	Frequent	$\mathcal{O}(\mathrm{T}^{M}_{P\gamma})$+O($T^{U}_{P\gamma}$)

recursive Bottom-up traversals in BPFSP-Tree (contained in Z_{Bottom}) without initiating any projection searching. So, $Z = Z_1 \cup Z_2 \cup ... \cup Z_8 \cup Z_{Prune} \cup Z_{Bottom}$. Lets assume the average complexity to perform projection of infrequency detection is S^* where $S^* << \frac{\sum_{P\in Z_1} T^M_P + \sum_{P\in Z_2} T^M_P + \sum_{P\in Z_3} T^M_P + ... + \sum_{P\in Z_8} T^M_P + T^U_P + \sum_{P\in Z_{Prune}} |P|+|B|}{|Z|}$. As the infrequent patterns ($P \in Z_{Prune}$) pruned using $CETable$, Heuristic strategy and mutual information are mostly related to the pattern's length and some bi-product calculation the upper bound complexity for such patterns can be set as $|P|$ (actual complexity is much less than this). To remove the infrequent patterns ($P \in Z_{Bottom}$) which were previously frequent but currently infrequent we start from the leaf nodes of BPFSP-Tree and go upwards till we can remove a pattern, so here the upper bound complexity can be set as $|B|$ where it denotes that total number of nodes in BPFSP-Tree. Actual complexity will be much less than this value. Similarly other terms ($\sum_{P\in Z_1} T^M_P$,..,$\sum_{P\in Z_8} T^M_P + T^U_P$) are also very high level estimation and infrequent patterns can be detected at various stages before performing complete projection. So, the actual value of S^* is much less than the mathematical equation established above. Also, the upper bound complexity of

TABLE 3.17: Incremental solution's upper bound runtime complexity

Function	Upper Bounds
Sequence Insertion	$\sum_{i=1}^{\|db\|} L_i^{db}$
Node attributes Update	$\mathcal{O}(\widetilde{u} \times \{\sum_{i=1}^{\|db\|} L_i^{db} + L_i^D\})$
$CETable$ Update	$\mathcal{O}(\|db\| \times k \times \widetilde{k})$
Pre-processing Complexity	$\sum_{i=1}^{\|db\|} L_i^{db} + \mathcal{O}(\widetilde{u} \times \{\sum_{i=1}^{\|db\|} L_i^{db} + L_i^D\}) + \mathcal{O}(\|db\| \times k \times \widetilde{k})$
Removal of previously frequent but currently infrequent patterns from BPFSP-Tree	$\mathcal{O}\|B\|$
Total Mining Complexity	$\mathcal{O}(\|Z\| \times S^*)$ $Z << (2^T - 1)^E$ $S^* << \frac{\sum_{P \in Z_1} T_P^M + ... + \sum_{P \in Z_8} T_P^M + T_P^U + \sum_{P \in Z_{Prune}} \|P\| + \|B\|}{\|Z\|}$

$|Z|$ is $(2^T - 1)^E$ where T denotes the average number of items in each sequence and and E denotes the average sequence length. As, we consider only the modified patterns to find and update its projection, $|Z| << (2^T - 1)^E$ and the total incremental mining complexity is $\mathcal{O}(|Z| \times S^*)$. In Table 3.16, we have already shown the upper bound runtime complexities during projection for each transition cases. The remaining runtime complexities with respect to upper bound has been shown in Table 3.17.

Up to now, we have discussed the generic IncTree-Miner. In the memory resilent version time complexity gets increased. For some patterns to get the actual status we have to perform some extra projections for its ancestor or sub patterns on the fly which adds some more cost to case 7 and 8 in Table 3.16 as $\mathcal{O}(T_{P\gamma}^M + T_{P\gamma}^U)$ to $\mathcal{O}(\sum T_{P^*}^U + T_{P\gamma}^M + T_{P\gamma}^U)$ where P^* denotes the sub patterns of $P\gamma$.

3.14 Application in Interactive Mining

Due to the structural design, our proposed tree-based structures hold the "build-once-mine-many" property. This property basically states that, our tree-based structures are able to perform multiple mining iterations for different minimum

support thresholds (min_sup) provided by users without performing any modification to the existing base structures. In this section, we will state how using our tree-based structures (SP-Tree and IncSP-Tree) and pattern storage structure (BPFSP-Tree), we can develop an incremental mining strategy.

To discuss this interactive mining concept, lets assume that we have already performed mining for a minimum support count threshold δ ($\delta = \lceil |D| \times min\_sup \rceil$). Now, assume the user has provided new minimum support threshold min_sup'', so the new minimum support count threshold becomes $\delta'' = \lceil |D| \times min\_sup'' \rceil$. Based on this defined δ and δ'', we can point out the cases as follows -

- $\delta'' > \delta$: As our current minimum support count threshold (δ'') is greater than our previous support count threshold (δ), we have already mined all the patterns satisfying the current threshold. In this case, we do not have to perform any mining iteration and just have to remove those patterns from the BPFSP-Tree which have support less than δ''. In this case, the bottom-up pruning strategy of BPFSP-Tree can be very useful. By using this concept we will need to traverse fewer patterns, compared to top-down strategy.

- $\delta'' < \delta$: In this case, we have already mined a group of patterns which already satisfy the current minimum support count threshold (δ''). But, in this case, we have to mine a new additional group of patterns which failed to satisfy the previous count threshold but satisfy the current one. Using BPFSP-Tree's projection information and NIB buffer, we can efficiently mine this additional group of patterns.

In the following chapter, we will evaluate the effectiveness of our proposed solutions to solve the approached problems.

Chapter 4

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 have also designed a new support counting strategy which helps detect patterns' infrequency early along with some additional pruning strategies. In this section, we will evaluate our proposed solutions based on various metrics.

To compare Tree-Miner, we have considered three popular state-of-the-art mining algorithms, PrefixSpan [22], CM-SPAM and CM-SPADE [14] and to compare IncTree-Miner we considered IncSP [36] and PBIncSpan [10]. 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 have similar key factors, there is a solid ground to compare with. We have introduced a new mining approach, so we considered those which did the same. All the implementations were in Python language and the experiments were conducted on a 64-bit machine having Intel Core i5-8265U CPU @ 3.90GHz $\times$ 8, 32 GB RAM, Windows 10 OS.

4.1 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 4.1. All the real-life datasets are collected from SPMF: A Java Open Source Data Mining Library[1] 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. 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 [36]. 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 4.2. To understand how the variables work here, we can construct an example using Table 3.1 and Table 3.2. Here Table 3.1 will work as D_{raw}, first iteration of Table 3.2 will work as D and second iteration of Table 3.2 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 the experiments to evaluate the performance of our proposed incremental solution, for each dataset, we have chosen $R_{new} = 10\%, R_{com} = 50\%$ and $R_{prev} = 80\%$ similar to [36]. To evaluate our solution, We have considered both dense ($\frac{\text{avg. sequence length}\times100}{(\#)\text{unique items}} \geq 4\%$) and sparse datasets.

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

Table 4.1: Dataset Summary

Name	Database Description	Sequence Count	Unique Items	Average Sequence Length	Verdict
Bible	Conversion of Bible	36369	13905	21.64	Sparse
MSNBC	Click Stream Data	31790	17	13.23	Dense
BMSWebView1	Click Stream Data	59601	497	2.43	Sparse
Sign	Sign Language Data	730	267	51.997	Dense
Kosarak	Click Stream Data	990000	41270	8.1	Sparse
L20k-C10-T5-N1K	Synthetic Dataset	20000	1000	44.21	Dense
L50k-C10-T2.5-N10K	Synthetic Dataset	50000	10000	21.79	Sparse

Table 4.2: Variables' Description

Variable	Description
D_{wraw}	Raw Database
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}. $\|D_{Insert}\| = R_{new} \times \|D_{raw}\|$
R_{com}	*Common Sequence Ratio:* Percentage of common sids picked from D_{raw} to put into D and D_{Append}. $\|D_{Append}\| = R_{com} \times \|D_{raw}\|$.
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	Average number of itemsets per sequence
T	Average Number of items in each itemset
N	Number of distinct Items

4.2 Performance Evaluation of Tree-Miner

In this section, we will evaluate our static solution, Tree-Miner based on SP-Tree on various metrics to understand the performance.

4.2.1 Runtime Analysis

In Figure 4.1 we present the runtime performance of Tree-Miner. Figures (a)-(e) represents the results in real-life datasets and (f)-(g) represents the results in synthetic datasets. From the corresponding figures, it is obvious that our proposed Tree-Miner based on SP-Tree performs comparatively better than other static mining algorithms. SP-Tree is a compact representation of the database which provides huge structural control over it. Through the next links, we perform efficient traversals in the database which ultimately help to generate the patterns faster. We also use a set of pruning techniques that significantly reduce the search space. We adopted all the previous pruning techniques and also incorporated some newer ones which makes it a very efficient mining algorithm. In the figures, in lower thresholds, our performance improvement is quite visible because we need to encounter a huge number of patterns and Tree-Miner efficiently discovers them whereas in upper thresholds the performance is quite close because the number of patterns is very small. The performance gets improved in dense datasets through node overlapping characteristics and in sparse datasets through faster database reduction and projection though next links. Our SP-Tree structure is enough to calculate the complete support of the patterns having no necessity to maintain additional structure for each pattern which is also an important factor to improve performance.

4.2.2 Memory Analysis

Due to the structured representation of the information, our solution needs comparatively more memory. We maintain SP-Tree and co-occurrence information

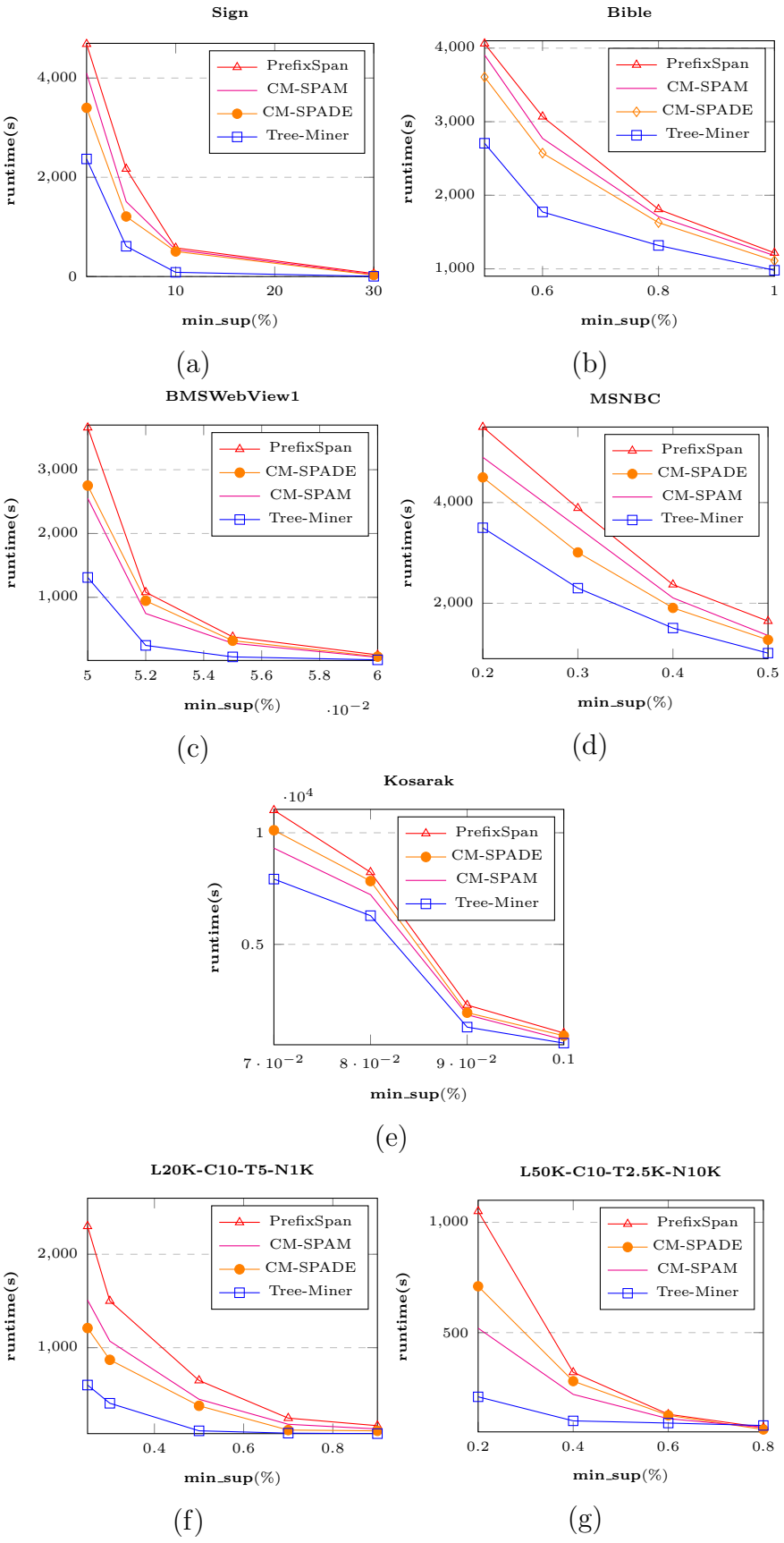


FIGURE 4.1: Runtime Comparisons of Tree-Miner

which gives control over the database and helps significantly during pattern extension. Other related approaches use pseudo projection or other structural support to calculate the frequency of the pattern which causes their memory usage. But, considering the runtime improvement, this should be negligible. We present an analysis in Figure 4.2.

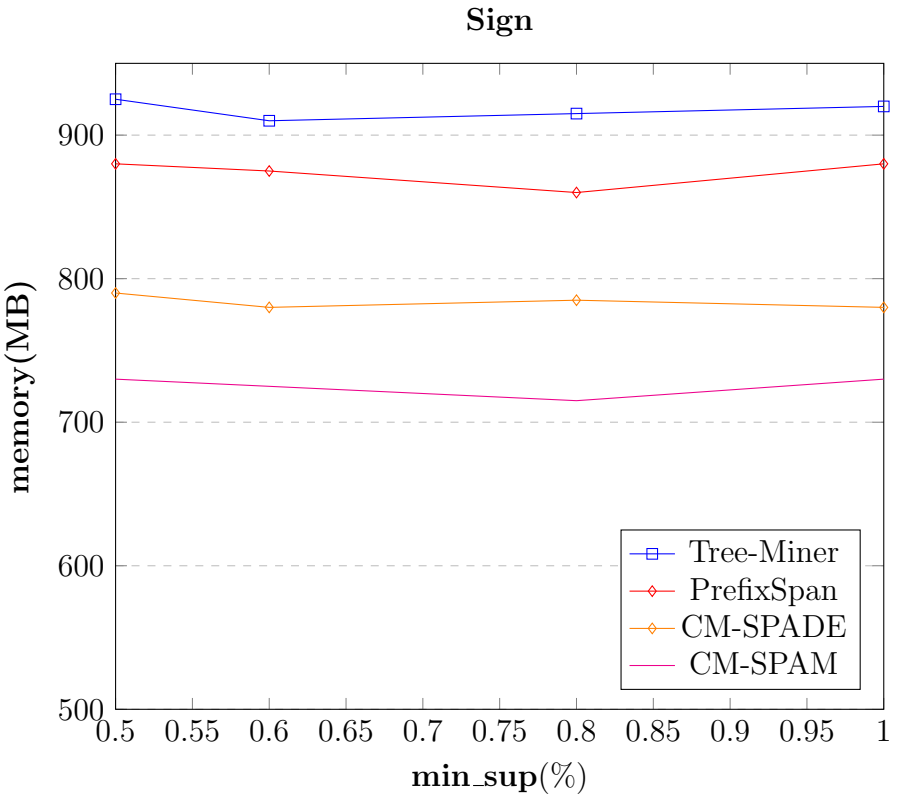


FIGURE 4.2: Memory Comparison for Tree-Miner

4.2.3 Scalability Analysis

We have also conducted a scalability analysis of Tree-Miner and presented the result in Figure 4.3 experimenting over the large *Kosarak* dataset. We started with 100000 transactions, gradually increased it and recorded the performance for various *min_sup* or minimum support threshold. The corresponding figure shows the linear scalability of the solution with the increased database size.

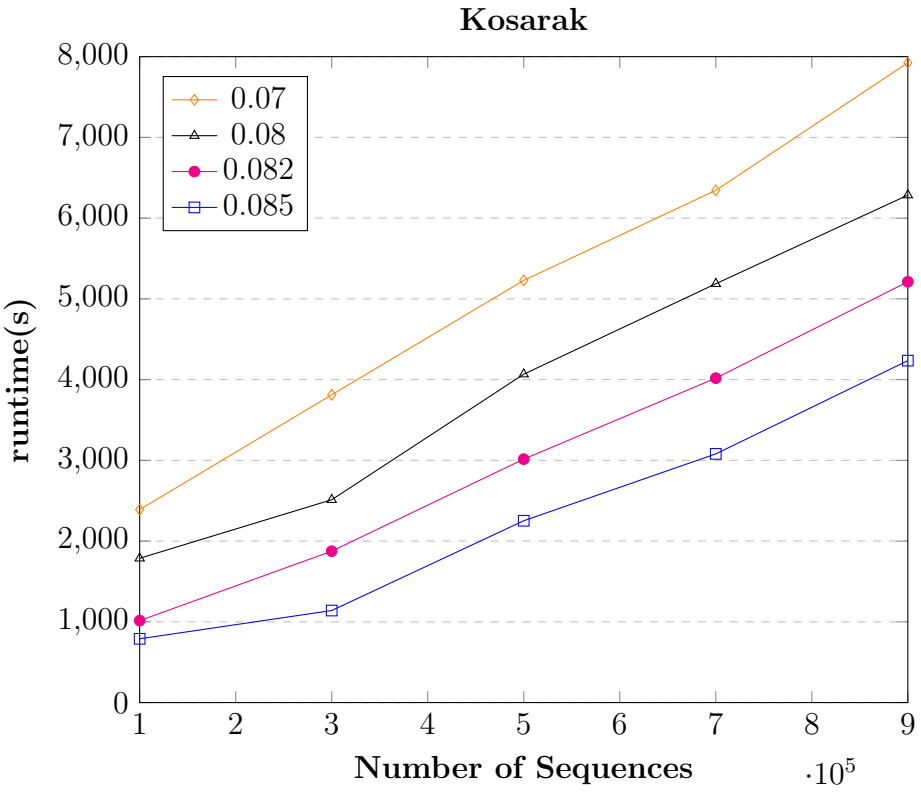


FIGURE 4.3: Scalability of Tree-Miner

4.2.4 Evaluation on Construction Time

Our proposed solution is a new structural approach to solve the sequential mining problem. So, an important concern can arise that, if the construction time creates any bottleneck during mining. For this purpose we present an analysis in Table 4.3. From the table, it can be understood that the total construction time is quite insignificant compared to full mining time. So the structured solution does not create any bottleneck rather provides a significant improvement in mining time. In the last three columns of Table 4.3 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 cost in pre-processing.

TABLE 4.3: Construction Time Vs Mining Time

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

4.3 Performance Evaluation of IncTree-Miner

In this section, we will evaluate our incremental solution, IncTree-Miner based on IncSP-Tree. We will consider various metrics and analyze our solution's performance.

In Section 4.1, we have already stated the ratio ($R_{new} = 10\%, R_{com} = 50\%$ and $R_{prev} = 80\%$) which we have used to create our incremental environment similar to [36]. In Table 4.4, we have shown the summary of such division for each datasets of Table 4.1. To create the incremental environment, first we randomly shuffled the raw database and then performed random splitting maintaining the chosen ratios. Here u_D and U_{db} denote the total number of unique symbols in D and db respectively.

TABLE 4.4: Incremental Environment of the Datasets

Name	$\|D_{raw}\|$	$\|D\|$	u_D	$\|D_{Append}\|$	$\|D_{Insert}\|$	$\|db\|$	u_{db}
Bible	36369	32732	12742	18184	3637	21821	7755
MSNBC	31790	28611	17	15895	3179	19074	17
BMSWebView1	59601	53640	488	29800	5961	35761	477
Sign	730	656	258	365	74	439	240
Kosarak	990000	891000	37537	462625	99000	561626	30702
L20k-C10-T5-N1K	20000	18000	915	10000	2000	12000	907
L50k-C10-T2.5-N10K	50000	45000	9113	25000	5000	30000	8648

4.3.1 Runtime Analysis

Our IncTree-Miner contains all the novelty of Tree-Miner and design 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 $modified\_next\_link$ 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 chances 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 but 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 and efficiently perform *Append* operations.

In Figure 4.4, we have shown the runtime performance of IncTree-Miner with $PBIncSpan$ and $IncSP$. Figures (a)-(e) show the results in real-life datasets and Figures (f)-(g) show the results in synthetic datasets. 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 which 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.

4.3.2 Memory Analysis

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 Figure 4.5 over *Bible* dataset. IncSP stores the support of the prior frequent patterns. So, its memory usage is dependent on the number of frequent patterns. PBIncSpan and IncTree-Miner both keep the previous frequent patterns' support along with their projection information where PBIncSpan uses pseudo projection and IncTree-Miner uses the compact IncSP-Tree node references. But IncTree-Miner stores the sequential database in tree format along with maintaining some data structures leading to comparatively more memory usage than PBIncSpan. As memory usage is directly related to the compactness of the tree, so in dense datasets, this metric's performance is comparatively closer to PBIncSpan.

4.3.3 Scalability Analysis

Similar to Tree-Miner, we have also conducted a scalability test for IncTree-Miner and presented the result over the *Kosarak* dataset in Figure 4.6. We started with 100000 transactions, gradually increased it and recorded the results by varying *min_sup*. The corresponding figure shows the linear scalability of the solution. To conduct the experiment, we chose $R_{new} = 10\%, R_{com} = 50\%$ and $R_{prev} = 80\%$ over the considered number of transactions and created the incremental environment described as before.

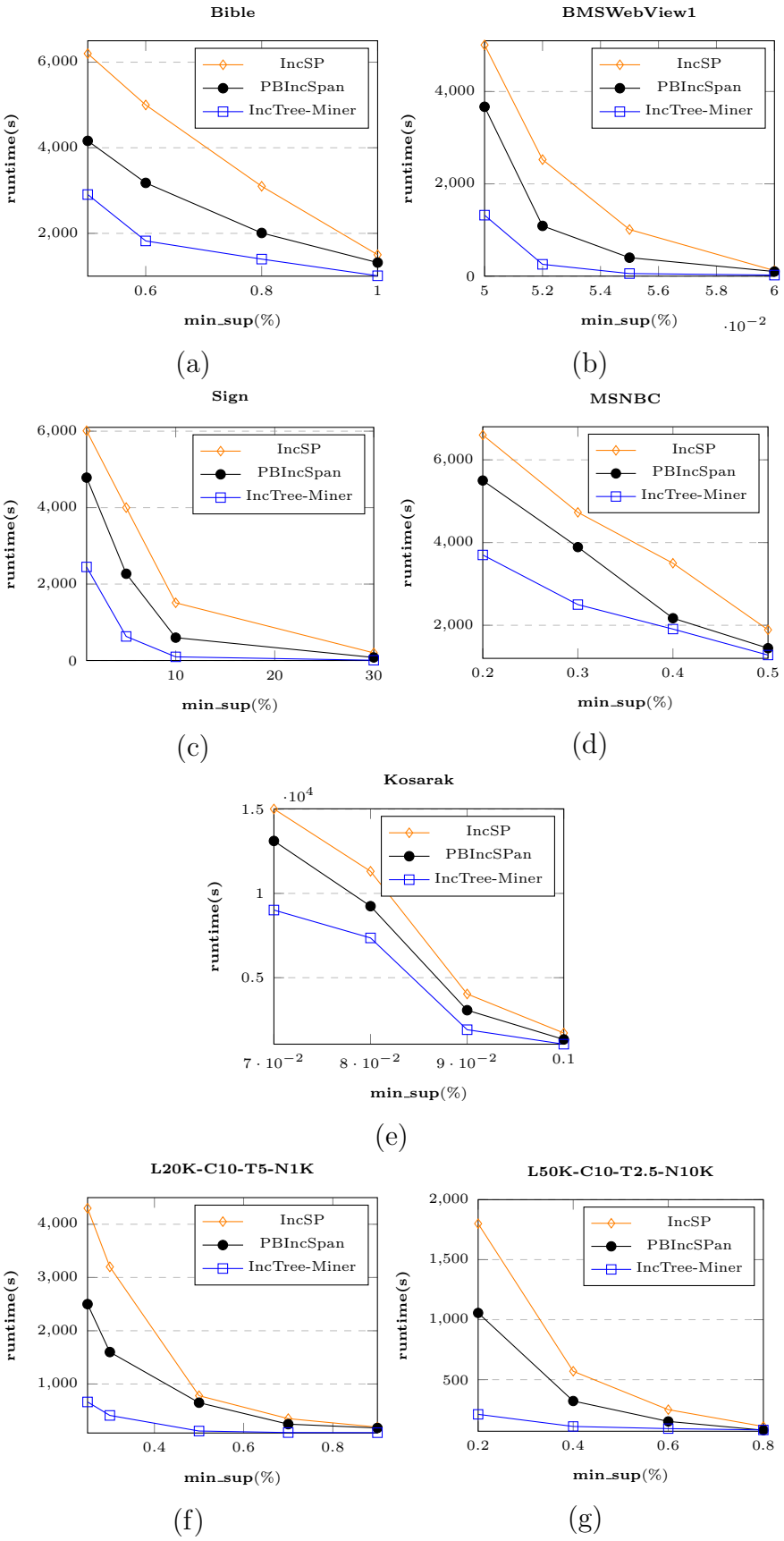


FIGURE 4.4: Runtime Comparisons of IncTree-Miner

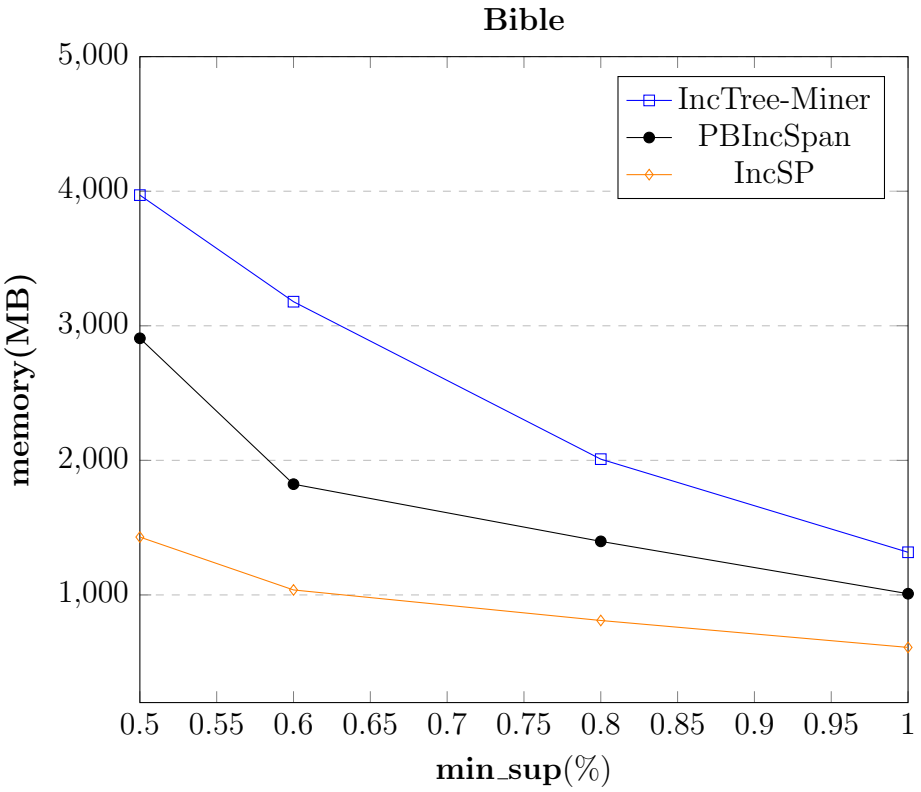


FIGURE 4.5: Memory Comparison for IncTree-Miner

4.3.4 Evaluation on Construction Time

In the Tree-Miner section, we have presented an analysis to understand the effect of construction time compared to mining time. Using Table 4.3, we have shown that, the structure construction time is quite insignificant compared to mining time. So, the additional construction time does not create any bottleneck rather improve mining performance by reducing runtime.

Our second proposal IncTree-Miner based on IncSP-Tree is an extended solution of the first proposal Tree-Miner based on SP-Tree to solve the incremental mining problem. Our incremental solution inherits all the characteristics of the static solution and maintains similar sets of data structures, *SP-Tree alike tree structure*, *Sequence Summarizer*, *Co-Existing Item Table*. So, the structure construction time also does not create any bottleneck for the incremental solution.

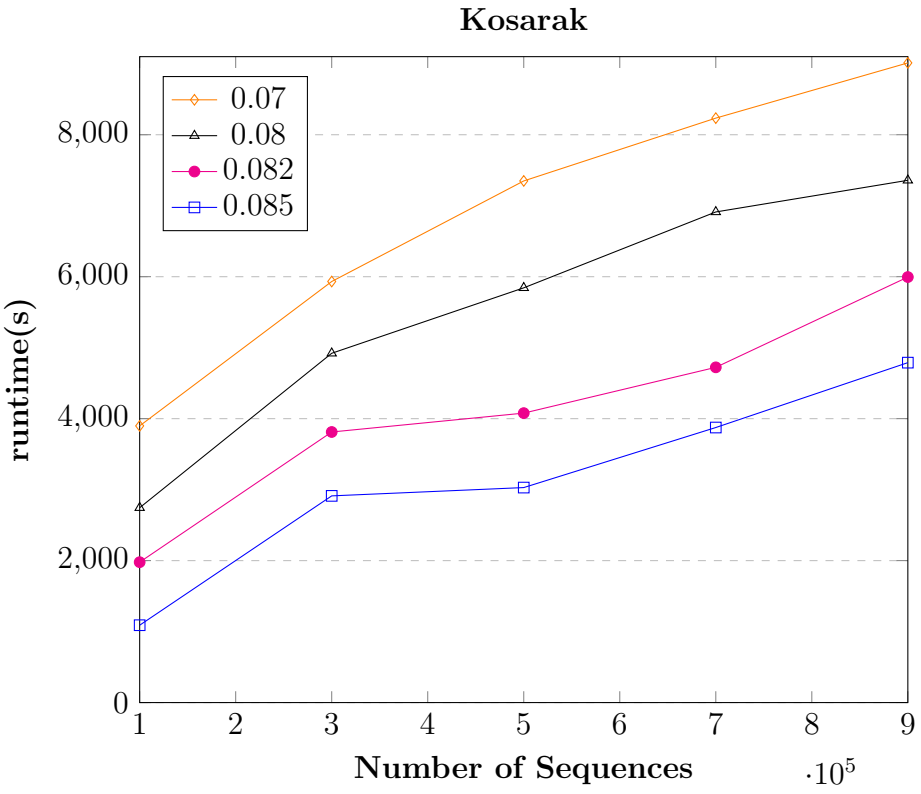


FIGURE 4.6: Scalability of IncTree-Miner

In our incremental solution, we also maintain a frequent pattern storage structure (BPFSP-Tree) parallelly. It gets parellelly updated with pattern's gradual update and does not impose any major time consuming parameter.

4.3.5 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 by keeping $R_{com} = 50\%$, $R_{prev} = 80\%$ and $min\_sup = 0.3\%$. We have shown the result in Figure 4.7. We started with 1% and gradually increased it to 30% and we always had two iterations to output the final set of patterns. We wanted to observe if the increased ratio of R_{new} affects the solution's performance or not. From Figure 4.7 it is clear that IncTree-Miner's performance does not degrade due to the R_{new}'s increasing ratio compared to PBIncSpan and IncSP.

With R_{new}'s increment the number of newer patterns' and updated patterns' gets increased in the second pass and the number of frequent patterns' gets decreased in the first pass. So, in our described scenario, the ISPM algorithm's performance should be almost linear and Figure 4.7 supports our intuition.

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

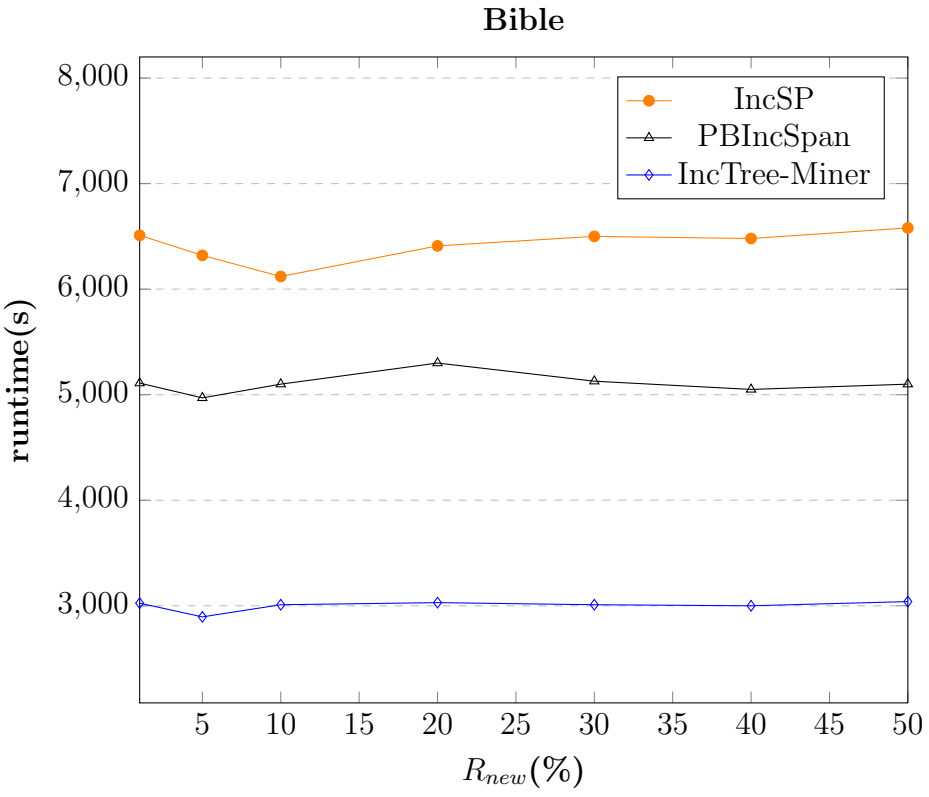


FIGURE 4.7: Performance over R_{new}

4.4 Evaluation of Breadth-First Based Support Counting Technique

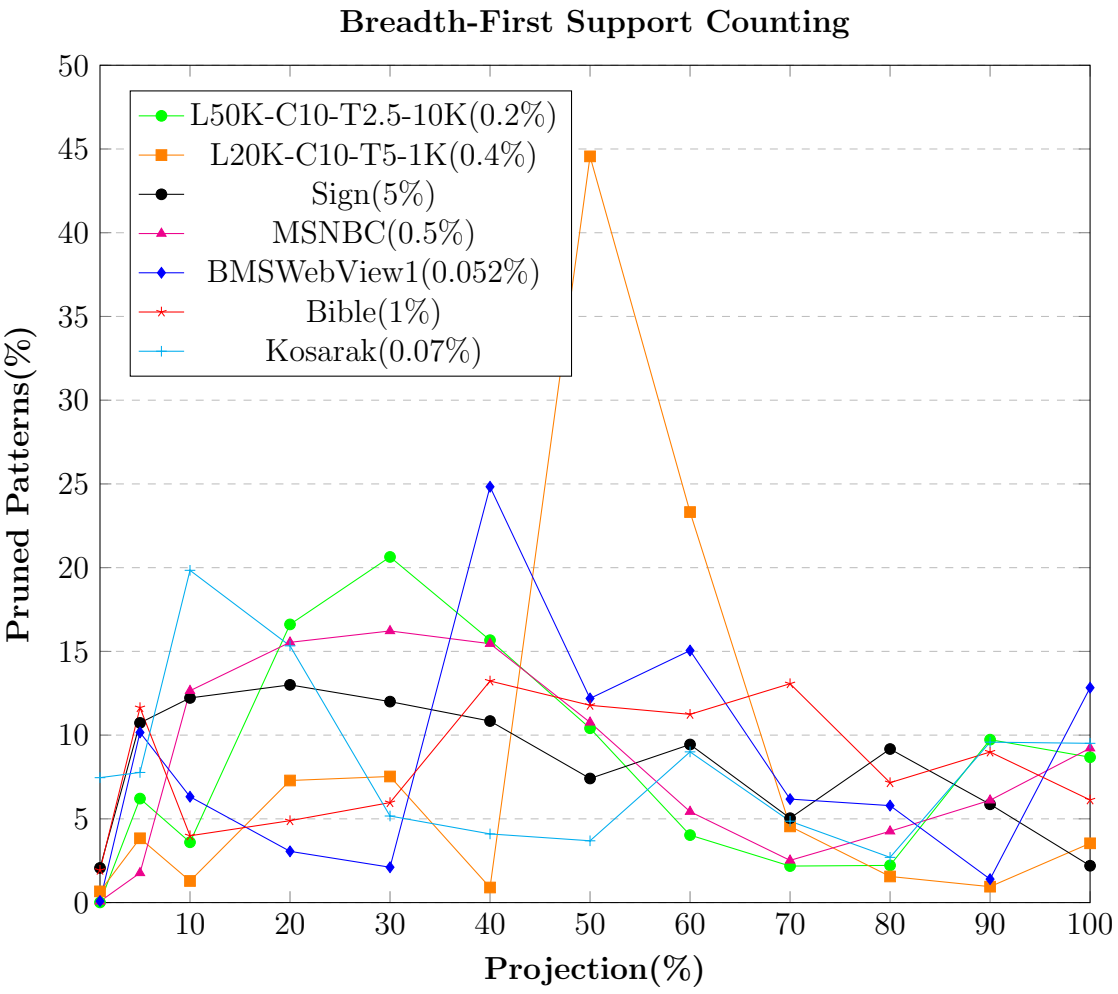


FIGURE 4.8: Evaluation of Breadth-First Based Support Counting Technique

In this section, we will evaluate the performance of the proposed breadth-first based support counting technique to understand how fast it can detect an infrequent pattern and stop support counting. To guide our mining process we use the co-occurrence information of the items stored in *sList* and *iList*. With the suffix extension of the pattern these two lists shrink. During support calculation of the extensions, we discover some patterns' infrequency. The algorithm's runtime improves depending on how quickly it is able to detect the infrequency

rather than performing complete projection. In Figure 4.8, we present an analysis to understand how quickly our proposed pruning technique detects infrequent patterns. In the x-axis, we have provided the percentage value of the projection size and in the y-axis, we have provided the percentage of infrequent patterns been detected. For example, in the *L20-C10-T2.5-N10K* dataset, at 50% value in the x-axis, we get a 45% value in the corresponding y-axis, which denotes that, of all the infrequent patterns been tested, 45% of them are detected performing only 50% projection of their corresponding complete projected database. From, Figure 4.8, it is clear that our proposed support counting technique is able to detect most of the infrequent patterns without performing a complete projection of the underlying sub-database. We have shown the corresponding *min_sup* value beside each legend in Figure 4.8. The figure is generated by executing Tree-Miner.

An example can be used to understand the concept. Suppose, in a database, we have a pattern $(a)(b)$ lying in 100 transactions. We have minimum support threshold value as 95 and we want to check extension for c $(e, g., (a)(b)(c))$ in $(a)(b)$'s projected database where $(a)(b)(c)$'s actual support value is 70. In our breadth-first solution, we remove a node and add its subtree during pattern extension, gradually reducing the maximum possible support. So, our approach might check only 10% or 10 transactions ($\sim$ 10%) or 10% of the total underlying subtrees and understand that $(a)(b)(c)$'s support will fall under 90 and thus will stop from further projecting.

4.5 Effectiveness in Interactive Mining

We have already stated that, due to the structural design our solutions hold the "build-once-mine-many" property. This basically indicates that, our structures are capable of performing multiple mining iterations based on users' requests without bringing any change to the existing structures for different minimum support thresholds (*min_sup*). In this section, we will provide some results to understand the proposed designs' effectiveness in this regard.

First, we will provide some intuitive analysis to understand how the performance can be improved, then we will provide some results to support the argument. It has already been stated that, in interactive mining, multiple mining iterations are performed for different minimum support thresholds. The most difficult scenario during interactive mining is, the gradual decrease in minimum support thresholds (20% $\rightarrow$ 10% $\rightarrow$ 5%, etc). In this scenario, with decrease in threshold, the minimum support requirement for a pattern to become frequent decreases. So, with gradual decrease, more and more patterns are generated and we need a mining iteration to discover the newly added patterns. This scenario, kind of resembles to the incremental scenario, where we already had a group of frequent patterns and we need to discover a new group of frequent patterns. BPFSP-Tree's projection information can be quite useful in this regard. Because, when we have to check the projection of a previously considered infrequent pattern, we can directly get its immediate sub pattern's projection information from the BPFSP-Tree and using it can calculate the new projection information for the concerned pattern. In Table 4.5, we have shown the results found from this interactive mining scenario to understand the runtime variation. From the table, it can be pointed out that, if we had to begin mining from scratch then definitely we needed much larger mining time. As here, a group of frequent patterns have already been calculated, a large portion of mining time is saved.

The best case scenario, during interactive mining comes from the gradual increase in minimum support thresholds (5% $\rightarrow$, 10% $\rightarrow$ 20%, etc) where with this increase, the minimum support requirement for a pattern to become frequent increases. In this case, we already had a larger group of frequent patterns for smaller threshold and we just have to remove the patterns which have failed to satisfy the updated increased threshold without performing any mining iteration. In this case, the bottom-up pruning strategy of BPFSP-Tree can be very effective which will help to traverse fewer patterns to remove the previously frequent but currently infrequent patterns from the tree. The first iteration, in this scenario

TABLE 4.5: Interactive Mining Runtime Variation (Descending *min_sup*)

Dataset	Descending *min_sup*				
Bible	Threshold (%)	10	1	0.8	0.5
	Runtime (sec.)	228.15	579.16	675	935.5
BMSWebView1	Threshold (%)	0.06	0.055	0.052	0.05
	Runtime (sec.)	19.8	71	226.5	865.2
Sign	Threshold (%)	30	10	5	1
	Runtime (sec.)	4	40.34	260.68	1610.6
MSNBC	Threshold (%)	0.5	0.4	0.3	0.2
	Runtime (sec.)	1037.44	492.72	829	742.2
Kosarak	Threshold (%)	0.085	0.08	0.075	0.07
	Runtime (sec.)	3037.3	2512	1231.4	1021.34
L20-C10-T5-N1K	Threshold (%)	0.04	0.035	0.03	0.025
	Runtime (sec.)	212.6	105.3	289.3	310.5
L50-C10-T2.5-N10K	Threshold (%)	0.031	0.03	0.025	0.02
	Runtime (sec.)	313.7	221.5	325.3	710

will be the most costly one but with gradual increase, the successive iterations will be extremely beneficial from the prior iterations. In Table 4.6, we have shown the results found from this interactive mining scenario to understand the changing runtime. From the table, the drastic time reduction in mining can simply be observed due to not performing any projection.

TABLE 4.6: Interactive Mining Runtime Variation (Ascending *min_sup*)

Dataset	Ascending *min_sup*				
Bible	Threshold (%)	0.5	0.8	1	5
	Runtime (sec.)	2907	8.2×10^{-4}	6.6×10^{-4}	3.4×10^{-4}
BMSWebView1	Threshold (%)	0.05	0.052	0.055	0.06
	Runtime (sec.)	1321	5.72	1.13	0.72
Sign	Threshold (%)	1	5	10	30
	Runtime (sec.)	2370	1.74	0.29	0.17
MSNBC	Threshold (%)	0.2	0.3	0.35	0.4
	Runtime (sec.)	3700	0.41	0.31	0.28
Kosarak	Threshold (%)	0.07	0.075	0.08	0.085
	Runtime (sec.)	7925	5.14	3.05	0.98
L20-C10-T5-N1K	Threshold (%)	0.025	0.03	0.035	0.04
	Runtime (sec.)	1012	1.14	0.08	0.07
L50-C10-T2.5-N10K	Threshold (%)	0.02	0.025	0.03	0.031
	Runtime (sec.)	2010	0.18	0.11	0.05

4.6 Analytical Novelty of the Proposal

Our proposed SP-Tree and IncSP-Tree provides an efficient structural manner to store the the sequential database leading to an overall improved runtime during mining. Our proposed Tree-Miner is an efficient algorithm to mine sequential patterns from static database and our proposed IncTree-Miner is an efficient algorithm to solve the incremental mining problem. IncTree-Miner's performance depends on single support threshold parameter where as some literature have adopted buffering concepts [11], multiple threshold based concepts [35] 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 request 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.

In this chapter, we have provided analysis of our proposed solutions based on various metrics to understand their novelty and effectiveness. In the following and last chapter of this study, we will summarize all the discussions presented up to now.

Chapter 5

Conclusions

Pattern mining problem intends to discover interesting patterns from a database. Traditional pattern mining or itemset mining problem discovers the frequently occurring patterns or itemsets from a database based on the minimum support threshold set by the users. Main limitation of itemset mining problem is, it does not consider the ordered relationship among the elements. So, to solve this limitation, *Sequential Pattern Mining* domain was introduced which discovers the frequently occurring ordered sequences from the database.

Due to the problem nature and complexity, the existing solutions of traditional itemset mining could not be used to solve the sequential mining problem. So, a completely separate research domain was created for this purpose. In our current investigation, we have approached the generic sequential mining problem which has found its usage and variations in numerous applications. Main challenge of the complete pattern mining domain is to discover the concerned or frequent patterns and efficiently discard the searching for unnecessary or infrequent patterns. Our complete set of contribution presented in this study can be broadly categorized into three groups, a novel solution to efficiently mine the frequent sequential patterns from the static databases, an extended version of the prior solution to efficiently discover the frequent sequential patterns from the incremental databases and addition of some new pruning strategies and supporting structures

to improve the mining runtime performance. Also, various issues related to the implementation and challenges have been discussed in rigorous details.

5.1 Summary of Research

In this study, we have proposed two novel tree-based solutions, *Tree-Miner* based on *SP-Tree* and *IncTree-Miner* based on *IncSP-Tree* to solve the *Sequential Pattern Mining* problem for static and incremental databases respectively. The tree-based structure provides structural advantage which ultimately helps to improve mining performance and handle manipulation over the database. We have also presented a new *Breadth-First Based Support Counting Technique* which helps detect the infrequent patterns early and a *Heuristic Pruning Strategy* to reduce redundant search space. We have also discussed the newly proposed pattern storage structure *BPFSP-Tree* for the incremental sequential mining problem based on *IncSP-Tree* and its efficient bottom-up strategy to remove previously frequent but currently infrequent patterns.

Our proposed solutions are designed based on a single support threshold parameter and able to mine the complete set of frequent sequential patterns. In our study we have also discussed the extendability of our approaches based on different types of scenarios along with to other solutions. Our proposed incremental solution also does not have any dependency of mining after each iteration and can mine any time based on users' requests. Also it is not severely affected due to pattern distribution variance and over computation alike buffer based or multiple parameter based solutions. Our designed structures hold the "build-once-mine-many" property which provides the scope to serve multiple mining requests of users without bringing any change to the existing structures. In the performance evaluation section, we have provided analysis for both of our solutions and showed their efficiency in improving mining runtime by performing experiments on both real-life and synthetic datasets.

5.2 Future Work

Up to now, we have designed our novel tree-based solutions both for static and incremental databases where we have flexibility of mining at any time based on users' requirements and works on single support threshold parameter. In the incremental mining problem, we had to solve the challenge of efficiently inserting sequences to the existing database and detecting the affected patterns at any time. As an ongoing and future work, we have planned to extend our solution to solve the sequential mining problem over high-speed data streams in real-time. We are planning to use different types of windowing techniques. e.g., sliding-window, runtime variable length window, etc. for mining and analyse the recent behavior of the users. Furthermore, we are planning to design extended solution to efficiently solve different types of compressed and representative sequential pattern mining problems, e.g., *maximal*, *closed*, *constraint-based*, etc., using our established tree-based structures.

We have also additional plan to extend the current tree-based solution to solve other specialized attribute based databases, such as, weighted and uncertain databases. Basically, our solution is a new viewing approach to solve the sequential mining problem. So, it provides scope to design new extended solutions to numerous related variations and applications of sequential pattern mining problem.

Bibliography

[1] Agrawal, R., Imieliński, T., & Swami, A. (1993). Mining association rules between sets of items in large databases. In *Proceedings of the 1993 ACM SIGMOD international conference on Management of data* (pp. 207–216).

[2] Agrawal, R., Srikant, R., et al. (1994). Fast algorithms for mining association rules. In *Proc. 20th int. conf. very large data bases, VLDB*, volume 1215 (pp. 487–499).

[3] Ahmed, C. F., Tanbeer, S. K., Jeong, B.-S., & Lee, Y.-K. (2011). Huc-prune: an efficient candidate pruning technique to mine high utility patterns. *Applied Intelligence*, 34(2), 181–198.

[4] Ahmed, C. F., Tanbeer, S. K., Jeong, B.-S., Lee, Y.-K., & Choi, H.-J. (2012). Single-pass incremental and interactive mining for weighted frequent patterns. *Expert Systems with Applications*, 39(9), 7976–7994.

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

[6] 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).

[7] Cagliero, L. & Garza, P. (2013). Infrequent weighted itemset mining using frequent pattern growth. *IEEE transactions on knowledge and data engineering*, 26(4), 903–915.

[8] 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.

[9] Chen, J. (2009). An updown directed acyclic graph approach for sequential pattern mining. *IEEE TKDE*, 22(7), 913–928.

[10] 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.

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

[12] Dawar, S., Sharma, V., & Goyal, V. (2017). Mining top-k high-utility itemsets from a data stream under sliding window model. *Applied Intelligence*, 47(4), 1240–1255.

[13] Ezeife, C. I. & Liu, Y. (2009). Fast incremental mining of web sequential patterns with plwap tree. *Data mining and knowledge discovery*, 19(3), 376–416.

[14] Fournier-Viger, P., Gomariz, A., Campos, M., & Thomas, R. (2014). Fast vertical mining of sequential patterns using co-occurrence information. In *Pacific-Asia Conference on Knowledge Discovery and Data Mining* (pp. 40–52).: Springer.

[15] 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(1), 54–77.

[16] Gan, W., Lin, J. C.-W., Fournier-Viger, P., Chao, H.-C., & Yu, P. S. (2019). A survey of parallel sequential pattern mining. *ACM Transactions on Knowledge Discovery from Data (TKDD)*, 13(3), 1–34.

[17] Grahne, G. & Zhu, J. (2005). Fast algorithms for frequent itemset mining using fp-trees. *IEEE TKDE*, 17(10), 1347–1362.

[18] Guidotti, R., Rossetti, G., Pappalardo, L., Giannotti, F., & Pedreschi, D. (2019). Personalized market basket prediction with temporal annotated recurring sequences. *IEEE TKDE*, 31(11), 2151–2163.

[19] Guns, T., Nijssen, S., & De Raedt, L. (2011). Itemset mining: A constraint programming perspective. *Artificial Intelligence*, 175(12-13), 1951–1983.

[20] Han, J., Cheng, H., Xin, D., & Yan, X. (2007). Frequent pattern mining: current status and future directions. *Data mining and knowledge discovery*, 15(1), 55–86.

[21] Han, J., Pei, J., & Kamber, M. (2011). *Data mining: concepts and techniques*. Elsevier.

[22] Han, J., Pei, J., Mortazavi-Asl, B., Pinto, H., Chen, Q., Dayal, U., & Hsu, M. (2001). Prefixspan: Mining sequential patterns efficiently by prefix-projected pattern growth. In *proceedings of the 17th international conference on data engineering* (pp. 215–224).: Citeseer.

[23] Han, J., Pei, J., & Yin, Y. (2000). Mining frequent patterns without candidate generation. *ACM sigmod record*, 29(2), 1–12.

[24] Hong, T.-P., Wang, C.-Y., & Tao, Y.-H. (2001). A new incremental data mining algorithm using pre-large itemsets. *Intelligent Data Analysis*, 5(2), 111–129.

[25] Hong, T.-P., Wang, C.-Y., & Tseng, S.-S. (2011). An incremental mining algorithm for maintaining sequential patterns using pre-large sequences. *Expert Syst Appl*, 38(6), 7051–7058.

[26] 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(9), 1153–1167.

[27] Huang, Y., Xiong, H., Wu, W., Deng, P., & Zhang, Z. (2007). Mining maximal hyperclique pattern: a hybrid search strategy. *Information Sciences*, 177(3), 703–721.

[28] Kim, H. & Choi, D.-W. (2020). Recency-based sequential pattern mining in multiple event sequences. *Data Mining and Knowledge Discovery*, (pp. 1–31).

[29] Lakshmanna, K. & Khare, N. (2016). Constraint-based measures for dna sequence mining using group search optimization algorithm. *International Journal of Intelligent Engineering & Systems*, 9(3), 91–100.

[30] Lan, G.-C., Hong, T.-P., Tseng, V. S., & Wang, S.-L. (2014). Applying the maximum utility measure in high utility sequential pattern mining. *Expert Systems with Applications*, 41(11), 5071–5081.

[31] Leung, C. K.-S. & Khan, Q. I. (2006). Dstree: a tree structure for the mining of frequent sets from data streams. In *Sixth International Conference on Data Mining (ICDM'06)* (pp. 928–932).: IEEE.

[32] 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(3), 287–311.

[33] Leung, C. K.-S. & Tanbeer, S. K. (2013). Puf-tree: a compact tree structure for frequent pattern mining of uncertain data. In *Pacific-Asia Conference on Knowledge Discovery and Data Mining* (pp. 13–25).: Springer.

[34] Lin, J. C.-W., Gan, W., Fournier-Viger, P., Hong, T.-P., & Tseng, V. S. (2016). Weighted frequent itemset mining over uncertain databases. *Applied Intelligence*, 44(1), 232–250.

[35] 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(5), 1071–1089.

[36] Lin, M.-Y. & Lee, S.-Y. (2004). Incremental update on sequential patterns in large databases by implicit merging and efficient counting. *Information Systems*, 29(5), 385–404.

[37] Lin, N. P., Hao, W.-H., Chen, H.-J., Chueh, H.-E., Chang, C.-I., et al. (2007). Discover sequential patterns in incremental database. *Internal Journal of Computers Vol. 1*, (4), 196–201.

[38] 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.

[39] Liu, Y., Liao, W.-k., & Choudhary, A. (2005). A fast high utility itemsets mining algorithm. In *Proceedings of the 1st international workshop on Utility-based data mining* (pp. 90–99).

[40] Mallick, B., Garg, D., & Grover, P. (2013). Incremental mining of sequential patterns: Progress and challenges. *Intelligent Data Analysis*, 17(3), 507–530.

[41] Masseglia, F., Poncelet, P., & Teisseire, M. (2003). Incremental mining of sequential patterns in large databases. *Data & Knowledge Engineering*, 46(1), 97–121.

[42] Okolica, J. S., Peterson, G. L., Mills, R. F., & Grimaila, M. R. (2020). Sequence pattern mining with variables. *IEEE TKDE*, 32(1), 177–187.

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

[44] 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 Transactions on knowledge and data engineering*, 16(11), 1424–1440.

[45] 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(6), 759–772.

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

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

[48] Salvemini, E., Fumarola, F., Malerba, D., & Han, J. (2011). Fast sequence mining based on sparse id-lists. In *International Symposium on Methodologies for Intelligent Systems* (pp. 316–325).: Springer.

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

[50] Srikant, R. & Agrawal, R. (1995). Mining generalized association rules.

[51] Srikant, R. & Agrawal, R. (1996). Mining sequential patterns: Generalizations and performance improvements. In *International Conference on Extending Database Technology* (pp. 1–17).: Springer.

[52] Tanbeer, S. K., Ahmed, C. F., Jeong, B.-S., & Lee, Y.-K. (2008). Cp-tree: a tree structure for single-pass frequent pattern mining. In *Pacific-Asia Conference on Knowledge Discovery and Data Mining* (pp. 1022–1027).: Springer.

[53] Tanbeer, S. K., Ahmed, C. F., Jeong, B.-S., & Lee, Y.-K. (2009). Efficient single-pass frequent pattern mining using a prefix-tree. *Information Sciences*, 179(5), 559–583.

[54] Tarus, J. K., Niu, Z., & Yousif, A. (2017). A hybrid knowledge-based recommender system for e-learning based on ontology and sequential pattern mining. *Future Generation Computer Systems*, 72, 37–48.

[55] Wang, K. (1997). Discovering patterns from large and dynamic sequential data. *JIIS*, 9(1), 33–56.

[56] Wang, Y.-N., Wang, J., Fan, X., & Song, Y. (2020). Network traffic anomaly detection algorithm based on intuitionistic fuzzy time series graph mining. *IEEE Access*, 8, 63381–63389.

[57] Xiong, H., Tan, P.-N., & Kumar, V. (2006). Hyperclique pattern discovery. *Data Mining and Knowledge Discovery*, 13(2), 219–242.

[58] Yang, Z. & Kitsuregawa, M. (2005). Lapin-spam: An improved algorithm for mining sequential pattern. In *21st International Conference on Data Engineering Workshops (ICDEW'05)* (pp. 1222–1222).: IEEE.

[59] Yin, J., Zheng, Z., & Cao, L. (2012). Uspan: an efficient algorithm for mining high utility sequential patterns. In *Proceedings of the 18th ACM SIGKDD international conference on Knowledge discovery and data mining* (pp. 660–668).

[60] Yun, U. (2007). Efficient mining of weighted interesting patterns with a strong weight and/or support affinity. *Information Sciences*, 177(17), 3477–3499.

[61] Yun, U., Nam, H., Kim, J., Kim, H., Baek, Y., Lee, J., Yoon, E., Truong, T., Vo, B., & Pedrycz, W. (2020). Efficient transaction deleting approach of pre-large based high utility pattern mining in dynamic databases. *Future Generation Computer Systems*, 103, 58–78.

[62] Zaki, M. J. (2001). Spade: An efficient algorithm for mining frequent sequences. *Machine learning*, 42(1-2), 31–60.

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

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

List of Notations

- s, s_i, sid_i - A sequence s, i^{th} sequence s, sid of the i^{th} sequence respectively.
- S_P - Support of a pattern P.
- S_P^D - Support of a pattern P in old database D.
- S_P^{db} - Additional support of a pattern P occurred due to addition of the incremental database db.
- $S_P^{D'}$ - Support of a pattern P in updated database D'.
- T_P^M - Total support calculate from the modified nodes that project P.
- T_P^U - Total support calculate from the unmodified nodes that project P.
- $(S_P^{db})^+$ - Intermediary variable to calculate S_P^{db}, starts from upper bound.
- $(T_P^U)^+$ -Intermediary variable to calculate T_P^U, starts from upper bound.
- H_P - Heuristic support of a pattern P.
- H_P^M - Heuristic support of a pattern P from the modified subtrees.
- H_P^U - Heuristic support of a pattern P from the unmodified subtrees.
- δ - $\lceil min\_sup \times |D| \rceil$, minimum support threshold of P in database D.
- δ' - $\lceil min\_sup \times |D'| \rceil$, minimum support threshold of P in database D'.
- N_P - Complete set of projection nodes for a pattern P.

- N_P^M - Complete set of modified projection nodes for a pattern P.
- N_P^U - Complete set of unmodified projection nodes for a pattern P.
- R_{new},R_{com}, R_{prev} - New sequence Ratio, Common Sequence Ratio, Previous Appearing Ratio.
- D_{Append} - A database consisted of the old sequences ($s_{sid} \in D$) which perform *Append* operation over the old database.
- D_{Insert} - A database consisted of the new sequences ($s_{sid} \notin D$) which perform $Insert$ operation over the old database.
- D_{raw} - Raw Database.
- $P\gamma, P\{\gamma\}, \{P\gamma\}$ - Generalized extension, SE, IE of P for γ respectively.