# Tree-Miner: Mining Sequential Patterns from SP-Tree

Redwan Ahmed Rizvee(✉), Mohammad Fahim Arefin, and Chowdhury Farhan Ahmed

Department of Computer Science and Engineering, University of Dhaka, Dhaka, Bangladesh
rizveeredwan.csedu@gmail.com, f.arefin8@gmail.com, farhan@du.ac.bd

**Abstract.** Data mining is used to extract actionable knowledge from huge amount of raw data. In numerous real life applications, data are stored in sequential form, hence mining sequential patterns has been one of the most popular fields in data mining. Due to its various applications, across the past decades, a significant number of literature have addressed this problem and provided elegant solutions. In this paper we propose a novel tree data structure, **SP-Tree**, to store the sequence database in a new and efficient manner. Additionally, we propose a new mining algorithm **Tree-miner** to mine sequential patterns from SP-Tree. To further enhance the performance of our algorithm, we incorporate multiple pruning techniques and optimizations. As our SP-Tree stores the complete database, it can also be used for incremental and dynamic databases, tree-structure is particularly advantageous for interactive mining. We demonstrate how our SP-Tree based Tree-miner algorithm significantly outperforms all of the existing state-of-the-art algorithms, across 6 real life datasets. We conclude by discussing the possible extensions of our approach to other related fields of sequential data.

**Keywords:** Pattern mining · Sequential pattern mining · Tree based mining approach

## 1 Introduction

Pattern mining is a branch of data mining which encloses the tasks of discovering inherent, useful and interesting patterns in databases. **Sequential pattern mining** was proposed [1] to apply the pattern mining techniques on sequential or ordered data, where the interestingness of a pattern can be measured in terms of various criteria such as its occurrence frequency, length, profit etc. An example of a sequential pattern is "Customers who buy a digital camera are likely to buy a color printer within a month." If a data-sequence is comprised of a set of events, the problem is to find all sequential patterns with a user-specified minimum support, where the support of a sequential pattern is the percentage of data-sequences that contain the pattern [9]. For example, in the database


H. W. Lauw et al. (Eds.): PAKDD 2020, LNAI 12085, pp. 44–56, 2020.
https://doi.org/10.1007/978-3-030-47436-2_4

of a retail superstore, each data sequence may correspond to the purchase history of a customer and each event represents the items bought in one purchase. A sequential pattern may be 10% customers bought 'Smartphone', followed by 'Screen Protector' and 'Powerbank'. Hence, sequential pattern mining methods are popularly used to identify patterns which are generally used in making recommendation systems, text predictions, improving system usability or making informative product choice decisions.

Due to its wide range of applications, numerous algorithms have been proposed to mine sequential patterns efficiently; most notably of two major classes- apriori based and pattern growth based. A typical apriori-like sequential pattern mining method, such as GSP [9], adopts a multiple-pass, candidate generation-and-test approach. But it is computationally expensive due to generation of huge set of candidates and multiple scan of the database which significantly reduces the performance in large and dense databases, specially in lower minimum support thresholds. On the other hand, pattern-growth based algorithms, which follow a divide and conquer approach are several times faster than the apriori algorithms. But there is still room for major improvement as these algorithms work by generating projected databases. Moreover, an efficient tree-based structure to store complete sequential databases is yet to be proposed, which could be useful in numerous cases like interactive pattern mining, sequential pattern mining in dynamic databases and applications with sliding window. Due to its numerous applications, mining sequential patterns in a parallel or distributed computing environment has also emerged as an important issue with many applications where tree alike structure could be useful.

Being motivated by this, we propose a tree based data structure **SP-Tree** and an algorithm, **Tree-Miner** to mine sequential patterns from it. Consequently, we demonstrate our algorithm's superiority compared to existing algorithms and highlight its versatility. Our main contributions in this paper are:

1. A tree-structure, **SP-Tree** to store the database in an efficient manner with build once, mine many property.
2. An efficient mining algorithm **Tree-Miner** to mine sequential patterns from SP-Tree.
3. Multiple Pruning techniques and optimizations to reduce runtime along with the scope of extensibility and scalability.

In this paper, we provide a brief discussion regarding the existing literature in Sect. 2. In Sect. 3, we propose our **SP-Tree** data structure and our mining algorithm, **Tree-Miner** along with pruning mechanisms and optimization techniques. In Sect. 4 we demonstrate our algorithm's performance across various real life datasets and we draw conclusions in Sect. 5.

## 2 Terminologies and Background Study

In this section, we explore the preliminary terminologies and concepts related to our problem domain and a brief discussion regarding the existing literature.

Let there be a set of **items** $I = i_1, i_2, ...i_m$. An **itemset** or event $X$ is a set of items such that $X \subseteq I$. A **sequence** $S$ is a collection or list of itemsets with a certain order [1] and can be written as $<e_1\ e_2\ e_3\ ...\ e_l>$, where each event $e_i$ happens before event $e_j$ if $i < j$ and each event $e_i$ is a set of items. **A sequence database** $SDB$ is a list of sequences. The **support** of a sequence $s_a$ in $SDB$ is defined as the number of sequences that contain $s_a$ and is denoted by $\mathbf{sup}(s_a)$. A sequence $s$ is said to be a **frequent sequence** or a sequential pattern if $sup(s) \geq minsup$, for a threshold $minsup$ set by the user. So, given a $SDB$ and a $minsup$, the problem of **mining sequential patterns** is, to generate all subsequences where each subsequence $s_a$ has $sup(s_a) \geq minsup$. If $\alpha = <$(ab)b$>$ and $\beta = <$(abc)(be)(de)c$>$, where a, b, c, d and e are items, then $\alpha$ is a subsequence of $\beta$.

As **mining sequential patterns** is a very popular problem, numerous research works have addressed this. Different algorithms follow different strategies and data structures to search for sequential patterns efficiently. As a result, some algorithms are more efficient than others. **GSP** [9] and **SPADE** [11] are two prominent works which have addressed this problem. Both solutions are based on candidate generation and testing paradigm. Their main bottleneck is that they generate a huge amount of redundant candidates while performing multiple database scans.

**PrefixSpan** [7] is one of the benchmark algorithms for frequent sequence mining which adopts a divide and conquer technique. It expands a pattern by recursively creating smaller projected databases on each iteration. Main computation cost of prefixspan is basically the generation of projected databases. Another renowned algorithm to solve the problem of frequent sequential pattern mining is **SPAM** [2] which introduced the idea of depth first search based technique to generate patterns in the search space along with efficient pruning mechanisms. These four literature were the benchmark works which provided completely new techniques to address the problem. After these, several novel techniques were introduced which provided some tweaking over them to improve the basic algorithm's performance. **FAST** [8] improved the support count technique of SPAM [2] using sparse id list which was a modification of SPADE's [11] idea. **Lapin** [10] was another improvement over SPAM [2] which showed the importance of last event's items that how it can reduce the search space and improve performance. A very efficient structure co-occurrence map was proposed in [5] which provided new technique to prune search space in both SPADE [11] and SPAM [2]. In this paper, we propose a complete tree-based structure to represent the sequential database and a mining algorithm along with efficient pruning mechanisms and improvisations to efficiently mine sequential patterns. Main motivation behind this work is, a complete and compact structure provides huge assistance to handle both dynamic and stream database along with interactive mining. Our technique also provides a new dimension to approach the problem.

## 3 Proposed Approach

In this section, we will discuss our proposed tree structure, **SP-Tree** and the mining algorithm **Tree-Miner** along with the pruning mechanisms and improvisations.

**Table 1.** Sequential database

| ID | Sequence |
|---|---|
| 1 | $\langle\{a\}\{abc\}\{ac\}\{d\}\{cf\}\rangle$ |
| 2 | $\langle\{ad\}\{c\}\{bc\}\{ae\}\rangle$ |
| 3 | $\langle\{ef\}\{ab\}\{df\}\{c\}\{b\}\rangle$ |
| 4 | $\langle\{e\}\{g\}\{af\}\{c\}\{b\}\{c\}\rangle$ |

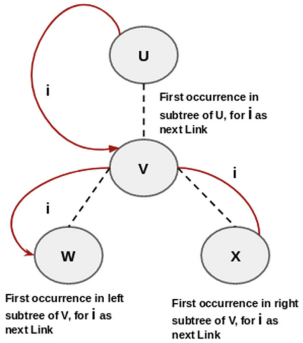



**Fig. 1.** Recursive next link move

### 3.1 SP-Tree

Our proposed Sequential Pattern Tree or *SP-Tree* is a tree which will represent the sequential database(*SDB*) in an efficient manner. We will consider the *SDB* of Table 1 in this section for discussion. In each row of the Table 1 we have sequences with their IDs. Here, item set domain $I = \{\mathrm{a}, \mathrm{b}, \mathrm{c}, \mathrm{d}, \mathrm{e}, \mathrm{f}, \mathrm{g}\}$ and for each sequence, items within curly braces form events. Now, we will explain the node structure of SP-Tree and the representative SP-tree of Table 1.

**Node Structure of SP-Tree:** Before diving into discussion we want to point out some important points. Each sequence's each event(set of items) should be lexicographically sorted and each item of each sequence have an event number which denotes the number of event in which this item appeared in the sequence. For example, in the first sequence of Table 1, first $a$'s event number is 1, because it belongs to event number 1 of that sequence. Similarly, second $a$'s event number is 2 because it appeared in event 2. So, embedded with event number first sequence can be seen as $\langle\{a:1\}\{a:2, b:2, c:2\}\{a:3, c:3\}\{d:4\}\{c:5, f:5\}\rangle$.

1. **Label:** Each node will represent an item and that item is the node's label.
2. **Event Number:** Each node will also have an event number which represents the node's label/item's event number.
3. **Count:** This number denotes how many times this node had been traversed during construction of the tree from sequences. This also denotes how many times this path(root up to this node) or prefix has been shared among the sequences. This attribute's value is important to calculate generated pattern's support.

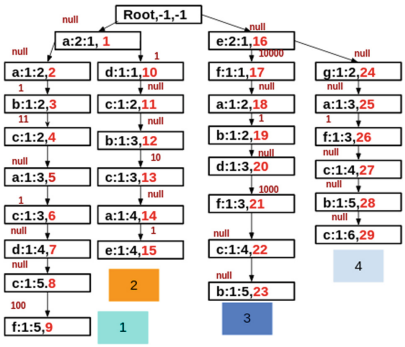
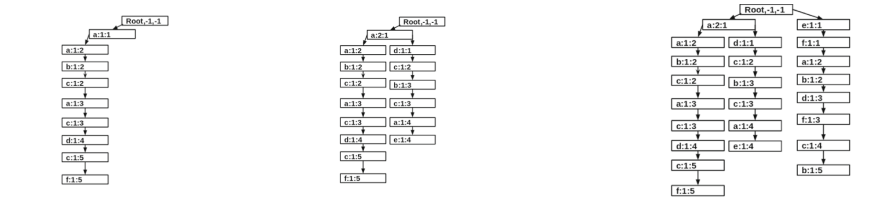



**Fig. 2.** SP-Tree of Table 1



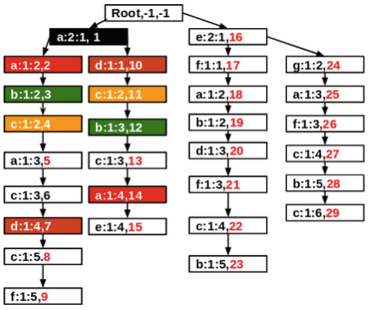
**(a)** Sequence 1 Inserted **(b)** Sequence 2 Inserted **(c)** Sequence 3 Inserted

**Fig. 3.** Intermediate processes of constructing SP-Tree

We have shown the complete SP-Tree of sample database of Table 1 in Fig. 2. In each node we have provided its label, count and event number. Red color values in each node is node number which we have used here for discussion purpose. At first we will have only root node. Then we will insert each sequence into the tree. For each sequence we put each item of the sequence of each event in the tree sequentially with their event number and label. 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 item's label and event with count 1 and 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 transaction/sequence into the tree. The intermediate processes of inserting first three sequences/transactions are shown in Fig. 3 and the complete tree after inserting last sequence is shown in Fig. 2.

Besides these three attributes we have two additional attributes.

4. **Next Link:** Next links are essential to traverse in the tree faster and efficiently. For a node $v$, next links for an item $it$ denotes the first node occurrences $(n_1, n_2, ...n_k)$ in $v$'s subtree in different branches for $it$. By moving through **next links** we can reach different nodes faster (Fig. 1) and generate patterns by con-




| item | Linked nodes |
|---|---|
| a | 2, 14 |
| b | 3, 12 |
| c | 4, 11 |
| d | 7, 10 |

**Fig. 4.** Next links for node 1

necting the node's labels. An example of next links for different items of node 1 is given in Fig. 4. Here to show next links for item $a, b, c, d$ we used different colors for better visualization.

5. **Parent Info:** Each node will store its parent nodes labels which are in the same event as it in the path from root to this node. This information is useful during mining to efficiently reduce search space. Now the best and compact way to store this information is using bitset. If we have domain knowledge, then we can number the items as 0,1,2,.. etc and make a bitset to store parent items. Like, in the SP-Tree of Fig. 4, node 4 needs to store label 'a'(node 2's label) and 'b' (node 3's label), because they are in the same event as it(event 2), so if we number 'a' as 0, 'b' as 1, 'c' as 2 and 'd','e','f','g' respectively then node 4 will store "11" as its parent information (setting the bits of position 0 and 1 only). This bit based representation will give ease to perform bitwise operations which will improve runtime. The respective parent info for each node is shown in Fig. 2. "null" means it does not have any parent item in same event. But our mining algorithm is also capable of handling other representations as well.

### 3.2 Co-existing Item Table

Suppose we have a pattern $P = \langle\{\alpha\}\{\beta\}\rangle$ where $\alpha$ and $\beta$ can be a single item or a set of items. During mining we extend a pattern in two ways, ***Sequence Extension*** ($SE$) and ***Itemset Extension*** ($IE$). $SE$ is if we add an item $A$ at the end of $P$ as new event resulting in $\langle\{\alpha\}\{\beta\}\{A\}\rangle$ and $IE$ is if we add an item $A$ in the last event of $P$ resulting in $\langle\{\alpha\}\{\beta A\}\rangle$. **Co-Existing Item Table** is helpful to understand which items co-exist in the database either in different event or in same event. By definition of pattern extension items existing in different event perform $SE$ and items existing in same event perform $IE$. This table is helpful to reduce search space by giving idea regarding the actual possible symbols to extend a sequence. We provide the Co-Existing Item Table of our database in Table 2.

In Table 2 we have shown the co-existing items for our database Table 1 along with their frequency. Each transaction contributes only once for each combination ($SE$ or $IE$). For $SE$ part this table can be efficiently calculated using $next-links$ and for $IE$ part this table can be calculated during insertion of the sequences in the tree. This idea was adopted in our methodology from [3,5].

**Table 2.** Co-existing item table of sample database

| Items | Sequence extending items | Itemset extending items |
|---|---|---|
| a | a:2, b:4, c:4, d:2, e:1, f:2 | b:2, c:1, d:1, e:1, f:1 |
| b | a:2, b:1, c:3, d:2, e:1, f:2 | c:2 |
| c | a:2, b:3, c:3, d:1, e:1, f:1 | |
| d | a:1, b:2, c:3, e:1, f:1 | f:1 |
| e | a:2, b:2, c:2, d:1, f:2 | f:1 |
| f | a:1, b:2, c:2, d:1 | |
| g | a:1, b:1, c:1, f:1 | |

### 3.3 Tree-Miner: Mining Sequential Patterns from SP-Tree

In this paper, we propose an efficient mining algorithm, **Tree-Miner** to generate patterns from SP-Tree. Tree-Miner is a recursive algorithm which concatenates the nodes of the SP-Tree and generates the sequential patterns by adding the node's labels using **next links**. This algorithm follows pattern expanding approach which means it starts with an empty sequence and gradually by traversing in the tree using **next links** it adds new symbols/items at the end of the sequence as $SE$ or $IE$. The node's count attribute resolves the issue of pattern's support calculation. Now, we will talk about the important concepts of Tree-Miner about how patterns are explored.

**Patterns Formation Rules:** Node combinations from SP-Tree makes a pattern and from different subtrees the first node combinations are always chosen. For example from our SP-Tree of Fig. 2 pattern $\langle\{a\}\rangle$ can be found in node 1, 18 and 25. These three nodes make pattern $\langle\{a\}\rangle$ in 3 different subtrees. Node combination $\{1, 3\}$ forms pattern $\langle\{a\}\{b\}\rangle$ in leftmost subtree, similarly node combination $\{1, 12\}$, $\{18, 23\}$ and $\{25, 28\}$ forms pattern $\langle\{a\}\{b\}\rangle$ in other three different subtrees and always the count attribute value of last node in each combination( here 3, 12, 23, 28) contributes to the pattern's frequency and here is 4. As, the first combination in different subtrees are always chosen it can be said that pattern $\langle\{a\}\{b\}\rangle$ can be found by reaching node 3, 12, 23 and 28 and this will make the $nodeList$ of pattern $\langle\{a\}\{b\}\rangle$ from where next iteration of pattern expansion will begin for $\langle\{a\}\{b\}\rangle$.

**Node Concatenation by Sequential Extension:** Suppose, we have a pattern $P$ and the $nodeList$ of $P$ is $N = \{n_i, n_j, n_k\}$ which denotes where $P$ ends in different subtrees(first occurrence). Suppose, we want to sequentially extend $P$ as

$P\{\alpha\}$ where $\alpha \in I$, then for each node $n \in N$ we need to search in its subtree, the first nodes which have different event number with $n$(**SE-Rule**). That node will sequentially extend node $n$ and for each $n$, the resultant nodes will make *nodelist* for $P\{\alpha\}$. Using next links we can perform recursive moves to find the desired nodes in the subtree. Like, in Fig. 2, for $\langle\{a\}\{b\}\rangle$ $nodeList = (3, 12, 23, 28)$. We want to make $\langle\{a\}\{b\}\{c\}\rangle$. Node 3 using next link for $c$ will reach first node 4 but it has same event number, so it will again move from node 4 using next link for $c$ and will eventually reach node 6. Node 6 is the valid extension for node 3. The resultant *nodeList* for $\langle\{a\}\{b\}\{c\}\rangle$ is (6, 29) and support is 2.

**Node Concatenation by Itemset Extension:** Suppose we have a pattern $\langle P\{Q\}\rangle$ where $P$ can be a set of events or empty and $Q$ a lexicographically sorted set of items and here the *nodeList* for $\langle P\{Q\}\rangle$ is $N$=$\{n_i, n_j, n_k\}$. Now if we want to extend the pattern as IE to $\langle P\{Q\beta\}\rangle$ where $\beta$ is an item and for any item $q \in Q$ lexicographically $< \beta$ then for each node $n \in N$ we need to find the nodes in the subtree of $n$ which will extend $n$ as $IE$ and will comprise *nodeList* of $\langle\{P\}\{Q\beta\}\rangle$. A node $v_i$ is extended by node $v_j$ as $IE$ if $v_j$ in subtree of $v_i$ and both have same event number. There can be two cases for each $n$ to find such node.

1. **Direct Node:** Using next link of $n$ for $\beta$ we reach a node which have same event number as $n$. This node will directly expand $n$ as $IE$. For example in Fig. 2, from node 18 (belonging to *nodeList* of $\langle\{a\}\rangle$) using next link for $b$ we can directly reach node 19 which have same event number as 18. So it will extend node 18 as $IE$.
2. **Indirect Node:** Using next link of $n$ for $\beta$ if we reach a node which does not have same event number as $n$. In this case, we have to find the node $k$ in the subtree of $n$ which have all the items of $Q$ as ancestor in the same event. For example, in Fig. 2, from node 1 (which belongs to *nodeList* of $\langle\{a\}\rangle$) suppose we want to extend it as IE with $b$ making a pattern $\langle\{ab\}\rangle$. Then first using next link $b$ from 1, we will reach node 3, but node 3's event number is different from node 1. So, Direct Node connection is not possible meaning $\{1, 3\}$ does not make $\langle\{ab\}\rangle$. So, we search in the subtree of node 1 using recursive next link for $b$ so that we can find such a node with label $b$ which have $a$ in same itemset. Interestingly in our case node 3 does the work having node 2 as same itemset with label $a$. So, ultimately node 3 belongs to the *nodeList* of $\langle\{ab\}\rangle$. In this purpose bitmask representation really becomes handy. By bitmasking with parent attribute value we will be able to get if this node has desired parent labels in same event.

For each pattern $P$ we always have a *nodeList* which denotes where the pattern ends and we always search for nodes in each subtree of each $n$ in *nodeList* to extend a pattern through recursive next link moves. Besides *nodeList* for each pattern $P$ there exists two lists *sList* and *iList* which says regarding the valid symbols which can perform $SE$ and $IE$ on $P$ respectively. Initially this will be made from Co-Existing Item Table with symbols which will satisfy *minsup*. In each iteration this two lists will get pruned. There can be three types of pruning during pattern extension. They are -

**Table 3.** Dataset description

| Dataset | Sequence | Distinct item | Avg. seq length (items) | Type |
|---|---|---|---|---|
| Snake [6] | 163 | 20 | 60 | Protein sequences |
| FIFA [4] | 20450 | 2990 | 34.74 | Web click stream |
| Leviathan [4] | 5834 | 9025 | 33.81 | Book |
| BMS [4] | 59601 | 497 | 2.51 | Web click stream |
| Sign [4] | 730 | 267 | 51.99 | Language utterances |
| Bible [4] | 36369 | 13905 | 17.84 | Book |

1. **Co-existing Item Table Based Pruning:** Suppose, we have a pattern $P$. Then we can add a symbol $\alpha$ with $P$ as $SE$ iff $\alpha$ occurs with each and every item of the last event of $P$'s at least $minsup$ times as sequence extending item. Similarly to extend $P$ by adding $\alpha$ as $IE$, it must occur with each and every item of the last event of $P$ as itemset extending symbols at least $minsup$ times. If this condition satisfies then and only then we will perform node concatenation and measure actual candidacy by support counting.
2. **sList and iList Pruning:** Suppose during pattern extension we have a pattern $P$ and the corresponding $nodeList$, $sList$ and $iList$. Suppose after node concatenation and measuring support we found that only $sList'(\subseteq sList)$ and $iList'(\subseteq iList)$ can extend $P$ as $SE$ and $IE$ respectively based on $minsup$. Then during recursive pattern expansion for each item $A$ in $sList'$ we can extend $P$ as $P\{A\}$ with $sList'$ as new $sList$ and new $iList$ as $sList'$-the items in the last event of $p\{A\}$. Now for each item $A$ in $iList'$ we can extend $P$ as $\{PA\}$ with $sList'$ as new $sList$ and with $iList'$-$A$ as new $iList$. This is a very popular pruning mechanism which we have adopted in our system.
3. **Heuristic iList Pruning:** Suppose for a pattern $P$ we have a $nodeList$, $sList$ and $iList$ and an item $A$ where $A$ is in both $sList$ and $iList$. After node concatenation and support counting we found that $A$ does not extend $P$ as $SE$. Now during node concatenation the nodes which were first visited through next link for $A$ from each node $n$ in $nodeList$, if their count attribute's summation does not satisfy $minsup$ then $A$ can be pruned from $iList$. It works because count attribute value of any parent node is always $\geq$ child node's count attribute. Due to having a tree like structure we could introduce this heuristic pruning technique.

## 4 Experimental Results

To evaluate the performance of *Tree-Miner* based on *SP-Tree*, we conducted several experiments on a 64 bit machine having intel Core i7-3770 CPU @ 3.40 GHz $\times$ 8, 8 GB RAM and Linux 16.04 Operating System. We analyze the performance with respect to runtime, memory consumption and structure construction time. To compare in run time and memory we will evaluate our performance against three state-of-the-art algorithms $PrefixSpan$, $CM\text{-}SPADE$ and $CM\text{-}SPAM$.

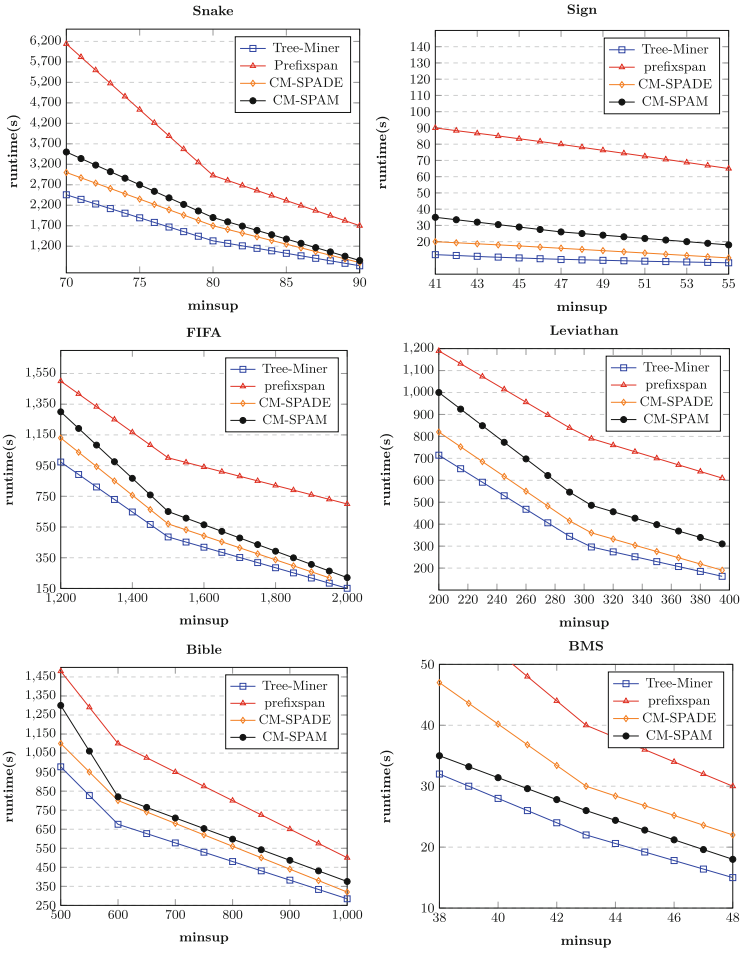



**Fig. 5.** Runtime comparison with various *minsup*

We have conducted our performance on various real life and synthetic datasets and among them we will show the results in the datasets of Table 3. In other datasets our performance were quite similar. We have conducted our approach's performance in both sparse and dense datasets and observed comparatively better results. If for a dataset $\frac{\text{avg seq. length}}{\text{number of unique items}} \geq 19\%$, we considered it as dense.

From the runtime analysis of Fig. 5, we can see that, our approach performs comparatively better than other state-of-the-art algorithms while it outperforms Prefixspan to a huge extent while CM-SPADE and CM-SPAM with comparatively closer but significant amount. Main superiority of our approach is, through next link it reduces the search space faster and efficiently and it does not need to generate any projected database and it also does not need any other structure to calculate the support of a pattern rather than only SP-Tree nodes.

**Table 4.** Construction time vs mining time

| Dataset | Threshold(%) | Mining(S) | Construction(S) |
|---|---|---|---|
| Snake | 42.94 | 2456 | 0.29 |
| Sign | 5.62 | 15.12 | 0.25 |
| FIFA | 5.87 | 973.44 | 1.19 |
| Leviathan | 3.43 | 714.4 | 0.89 |
| Bible | 1.37 | 977 | 2.71 |
| BMS | 0.064 | 27.1 | 0.1 |

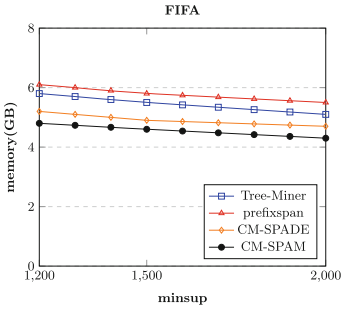



**Fig. 6.** Memory comparison

Our SP-Tree has two important characteristics, one is prefix sharing and another one is next link which we have already mentioned. Through prefix sharing it improves the performance in dense datasets specially while through next link it can move in the tree efficiently specially improving performance in sparse datasets. Besides bit based representation during itemset extension as parent item info also improves performance. In lower thresholds, performance gap is better compared to higher thresholds. Because in lower thresholds we have a significant search space and our approach can traverse in them with better efficiency while in higher thresholds search space gets reduced for each algorithm and so, though ours better but no so differentiable due to time reduction.

As, we provide a structure SP-Tree to represent the sequential database, definitely it will need a memory for that. Besides we use a Co-Existing Item Table to prune search space. These two are the most vital factors which consume the memory usage in our approach. From experiments, we found that our approach takes slightly more memory compared to CM-SPADE and CM-SPAM but less memory compared to Prefixspan, mainly because we do not need to generate any projected databases and pattern's support count measure also does not need any other structure except SP-Tree nodes. But considering runtime improvement this should be considerable. We have shown the comparison of memory usage in FIFA dataset in Fig. 6, in other datasets performance were quite similar. Another important point to note that these structures can be built only once on the complete database and can be used for mining at various *minsup*. So, our solution can be very useful for interactive mining. For the sake of comparison, in Fig. 5 we constructed the tree and table each time from scratch considering the *minsup* and then compared with other algorithms (because they were not interactive algorithms) and found comparatively better results. So, if we had saved the complete structure and mined then definitely performance improvements would have been even more significant. Besides from Table 4 we can see that construction time(tree and table) is insignificant compared to mining time. So, if we need to mine the same database in various thresholds or in lower thresholds our solution is quite impressive. Main challenges behind a tree based structure was how to represent the items within same event in an efficient manner and distinguish during mining. Our SP-Tree and Tree-Miner algorithm provides a novel solution in this regard.

## 5 Conclusion

In this paper, we presented a tree based data structure, *SP-Tree* to store sequential databases and a new mining algorithm *Tree-Miner* to mine sequential patterns efficiently from the tree. We have also utilized the idea of *Co-Existing Item Table* to reduce search space and various pruning mechanisms for pattern expansion phase to improve runtime along with improvisations. We have also demonstrated our mining algorithm's superior performance against various state-of-the-art approaches along with other important metrics performance in experimental analysis section. As our solution is a tree based approach and maintains the *build once mine many* property, it has significant advantage to approach problems regarding interactive mining along with dynamic databases and sliding window based problems. In this paper, we proposed the tree structure and basic mining technique to discover sequential patterns and we plan to extend this solution to solve challenges of dynamic sequential databases and problems regarding sliding window. Besides, another advantage of tree based solution is having a structured way to handle all the data by its *branches*, *subtrees* etc. which can be used in numerous branches of pattern mining including multilevel, multidimensional and parallel or distributed sequential pattern mining.

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

## References

1. Agrawal, R., Srikant, R., et al.: Mining sequential patterns. In: ICDE vol. 95, pp. 3–14 (1995)
2. Ayres, J., Flannick, J., Gehrke, J., Yiu, T.: Sequential pattern mining using a bitmap representation. In: Proceedings of the Eighth ACM SIGKDD International Conference on Knowledge Discovery and Data Mining, pp. 429–435. ACM (2002)
3. Fournier-Viger, P., Gomariz, A., Campos, M., Thomas, R.: Fast vertical mining of sequential patterns using co-occurrence information. In: Tseng, V.S., Ho, T.B., Zhou, Z.-H., Chen, A.L.P., Kao, H.-Y. (eds.) PAKDD 2014. LNCS (LNAI), vol. 8443, pp. 40–52. Springer, Cham (2014). https://doi.org/10.1007/978-3-319-06608-0_4
4. Fournier-Viger, P., et al.: The SPMF open-source data mining library version 2. In: Berendt, B., et al. (eds.) ECML PKDD 2016. LNCS (LNAI), vol. 9853, pp. 36–40. Springer, Cham (2016). https://doi.org/10.1007/978-3-319-46131-1_8
5. Fournier-Viger, P., Lin, J.C.W., Kiran, R.U., Koh, Y.S., Thomas, R.: A survey of sequential pattern mining. Data Sci. Pattern Recognit. **1**(1), 54–77 (2017)
6. Jonassen, I., Collins, J.F., Higgins, D.G.: Finding flexible patterns in unaligned protein sequences. Protein Sci. **4**(8), 1587–1595 (1995)
7. Pei, J., et al.: PrefixSpan: mining sequential patterns efficiently by prefix-projected pattern growth. In: ICDE, pp. 215–224. IEEE (2001)
8. Salvemini, E., Fumarola, F., Malerba, D., Han, J.: FAST sequence mining based on sparse id-lists. In: Kryszkiewicz, M., Rybinski, H., Skowron, A., Raś, Z.W. (eds.) ISMIS 2011. LNCS (LNAI), vol. 6804, pp. 316–325. Springer, Heidelberg (2011). https://doi.org/10.1007/978-3-642-21916-0_35

9. Srikant, R., Agrawal, R.: Mining sequential patterns: generalizations and performance improvements. In: Apers, P., Bouzeghoub, M., Gardarin, G. (eds.) EDBT 1996. LNCS, vol. 1057, pp. 1–17. Springer, Heidelberg (1996). https://doi.org/10.1007/BFb0014140
10. Yang, Z., Wang, Y., Kitsuregawa, M.: LAPIN: effective sequential pattern mining algorithms by last position induction for dense databases. In: Kotagiri, R., Krishna, P.R., Mohania, M., Nantajeewarawat, E. (eds.) DASFAA 2007. LNCS, vol. 4443, pp. 1020–1023. Springer, Heidelberg (2007). https://doi.org/10.1007/978-3-540-71703-4_95
11. Zaki, M.J.: Spade: an efficient algorithm for mining frequent sequences. Mach. Learn. **42**(1–2), 31–60 (2001)