Mining Web Access Patterns using Root-set of Suffix Trees

Manira Akhter
Sr. Software Engineer
Samsung R & D Institute
Bangladesh

Ashin Ara Bithi
Lecturer
Dept of Computer Science and Engineering
Asian University of Bangladesh

Abu Ahmed Ferdaus
Assistant Professor
Dept of Computer Science and Engineering
University of Dhaka, Bangladesh

ABSTRACT

With the rapidly growing uses of World Wide Web for various important and sensitive purposes it becomes a sensible necessity to find out the interesting web access patterns from the web access sequences tracked by users frequently. Web access sequential patterns can be used to achieve business intelligence for e-commerce sites and also can be used to analyze system performance. This paper proposes a more efficient web mining algorithm which mines all the sequential patterns from the web access sequences and totally eliminates the concept of linking between nodes. The algorithm uses the aggregate tree structure for mining and then mines from the tree using RST (Root-set of Suffix Trees) for same prefix items. The algorithm finds the frequent sequential patterns by recursively traversing the tree from root-nodes to child-nodes for the length-1 frequent items. The proposed approach doesn’t need to generate any projected tree; it needs only the root-set for each prefix that got in previous step. Experimental results show huge performance gain over the FOF and WAP-tree mining techniques by considerably reducing the mining time.

Keywords

Frequent sequential pattern, Web access sequence, Web log mining, WAP-tree, First-Occurrence Forest (FOF)**,** and Root-set of Suffix Tree (RST).

1. INTRODUCTION

Web access mining, also known as web log mining, aims to discover interesting and frequent user access patterns from the web log data gained from the interactions of users while surfing the web. For mining purposes, the web browsing data can be stored in web server logs, proxy server logs or browser logs. The mined knowledge can then be used in many practical applications, such as improving the design of web sites, analyzing user behaviors for personalized services, and developing adaptive web sites according to different usage scenarios. Web log mining is a part of sequential pattern mining which is based on association rule mining concepts. In sequential pattern mining sequences of data are mined from the sequence database. In case of web log database each sequence consists URLs accessed by the same user and the database contains multiple sequences of different customers. Due to the importance and various applications of web log mining, now-a-days it becomes an important research field. In recent years many web access pattern mining algorithms are proposed to extract significant information from web log database. The most important and popular web mining algorithms are GSP [1] which is an apriori based algorithm and WAP-tree [2], PLWAP-tree [3], FLWAP-tree [4], and FOF [5] mining algorithm etc which are pattern-growth approach. We know that, apriori based algorithms create lots of candidate sets during mining which degrades the performance of those algorithms. To overcome this problem, in recent years various pattern growth approaches were proposed which can generate the frequent patterns without creating any candidate set. We know WAP [2] is a pattern growth approach used to find frequent sequential patterns from web access database without generating any candidate set but the main drawback of it is creation of intermediate WAP-trees during mining which needs more execution time and also needs large memory spaces to store those trees. There are also some approaches those avoid generating intermediate trees during mining. PLWAP [3] and FOF [5] are such algorithms which do not generate any intermediate tree during mining instead they use the concept of suffix tree. But PLWAP and FOF also have some drawback that reduces their performance. We know that, PLWAP uses position codes for uniquely identifying the tree nodes and also it has an extra burden of maintain queues for storing the nodes with same labels. Though FOF algorithm avoids storing any node link and position code information but its traversing process is very time consuming i.e. to generate a single sequential pattern it needs a traverse through the tree. So, to generate n frequent sequential patterns it needs at least n number of tree traversing. For example, if maximum sequence size is m for any database and let n is the number of length-1 frequent items so to generate maximum number of patterns(where pattern size >= 2) it needs number of traversing = $n^2+n^3+n^4+\dots+n^m$.

In this paper, we have proposed an efficient web log mining algorithm based on root-nodes of suffix trees. The proposed mining approach finds all the frequent sequential patterns from the web log database using Root-set of Suffix Trees (RST). As a tree structure it uses simple original aggregate tree which has no links between nodes and also it does not regenerate any intermediate tree during mining that WAP-tree mining [2] does. The main effectiveness of the proposed algorithm is that, it needs few numbers of tree traverses than others. It can generate all the patterns with same prefix by a single traverse through the tree while FOF mining [5] traverses the tree for each single pattern. Traversing for only a single pattern is an overhead of FOF mining [5] and our proposed approach overcomes this by maintaining a table which stores all the items and their corresponding counts that found during traversing for a specific prefix. After traversing, it generates the frequent patterns from the table and start traversing with new prefixes recursively. So to generate maximum number of patterns (where pattern size >= 2) the proposed approach needs number of traversing = $n+n^2+n^3+\dots+n^{m-1}$. The experimental results show that the proposed approach is faster than the WAP [2] and also FOF [5] because of its less number of tree traversing than others.

In the rest of the paper, section 2 contains related work; the proposed RST-based approach for web access database with an example has been presented in section 3. Section 4 concentrates on the results of the findings and comparisons of our approach with two existing approaches using two datasets and finally, section 5 draws conclusion that points out the potentiality of our work

2. RELATED WORKS

We have compared our proposed approach with two existing algorithms. These algorithms are presented sequentially in this section.

2.1 Web Access Pattern tree (WAP-tree)

Web Access Pattern tree mining [2] is a sequential pattern mining approach for Web access database proposed by Peijian, Han, Mortajav, and Hzsua in April, 2000. This paper proposed a tree structure WAP-tree almost similar to FP-tree [6] to store the Web access subsequences with only frequent items in a compressed form .While FP-tree [6] links only the first occurrences of items in the tree branches, the WAP-tree [2] links together all the items in the tree with same symbol. WAP-tree algorithm [2] then mines the frequent sequences from the WAP-tree by recursively reconstructing the intermediate tree using suffix sequences. The main contributions of WAP-mining [2] are as follows: **First**, a concise, highly compressed WAP-tree structure is designed and implemented which handles the sequences elegantly. **Second**, an efficient mining algorithm (Like FP-growth), WAP-mine, is developed for mining the complete (but non redundant) Web access patterns from large set of pieces of Web log. **Third**, a performance study has been conducted which demonstrates that the WAP-mine algorithm is an order of magnitude faster than its apriori-based counterpart for mining Web access sequences. Although it is better than apriori-based algorithms but it has some drawbacks. Firstly, it has to maintain lots of links information among the tree nodes and secondly, it generates lots of intermediate trees during mining. So it needs large memory space to store the links information and intermediate trees. Because of intermediate tree generation, the performance of this algorithm is lower than other newly proposed techniques.

2.2 First-Occurrence Forests (FOF) mining

First-Occurrence Forests [5] is a pattern growth mining technique used to mine Web log sequential patterns proposed by Erich A. Peterson and Peiyi Tang in March 2008. They use the simple Aggregate tree structure for mining and totally removed the concept of linked tree. Figure 1 shows the tree structure for FOF mining using the database sequences <a b a c>, <a b c a c>, <b a b a>, <a b a c c>, and <a b>. They use forests of first-occurrence subtrees as the basic data structure for the database representation, and employ a simple list of tree node pointers to first-occurrences in the aggregate tree. There is no need to rebuild aggregate trees for projection databases. The first-occurrences of a symbol are found using a depth-first search of the aggregate tree on-the-fly. Given a symbol 'a', each subtree rooted at a first-occurrence of ‘a’ is called first-occurrence subtree of ‘a’. The forest of first-occurrence subtrees of ‘a’ symbol is simply a list of pointers to the first-occurrences of 'a' in the aggregate tree. Figure 2 shows the forest of first-occurrence subtrees of 'a' using the same database sequences. The root-nodes of the first-occurrence subtrees form part (i). The subtrees rooted at the children of the nodes in part (i) form part (ii) which represents the projection database D_a. The algorithm recursively creates the pointers for the first-occurrence forest trees and explores the frequent patterns as pattern-growth approach using the depth-first search. Since, every time FOF mining [5] traverses the tree for generating a single pattern i.e. it creates one frequent sequential pattern by one traversing. For each prefix pattern it needs to traverse as same as the number of length-1 frequent patterns i.e. if the number of length-1 frequent patterns is three for the given database, it will traverse the tree three times for same prefix pattern. This is the drawback of this technique and may degrade its performance.

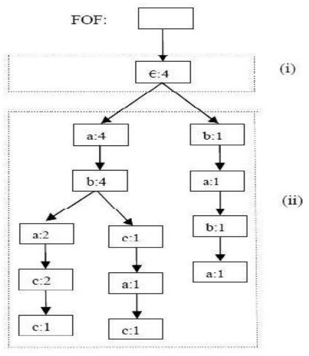


Fig 1: Initial Tree for FOF Mining

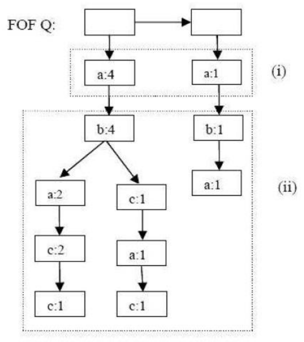


Fig 2: FOF for D_a

3. PROPOSED RST-BASED MINING APPROACH

In this paper, we have proposed a new efficient web pattern mining algorithm based on RST. The algorithm searches frequent patterns using the Root-set of Suffix Trees (RST) for each prefix pattern. In every state it stores the root-nodes of suffix trees for each frequent item created on-the-fly and uses those root-nodes pointers in the next step. In this way it explores the patterns as pattern growth approach. It traverses the original aggregate tree [7] for every prefix sequences using the depth-first search manner. Although it uses the suffix trees' root-nodes but it does not generate the suffix trees physically instead; it uses the root-nodes as pointers and starts traversing from those pointers.

3.1 The Proposed Web Mining Algorithm

3.1.1 Notation

Here,

α = Individual frequent item

Σ = Length-1 Frequent itemset

μ = Minimum support

3.1.2 The Proposed Algorithm

The proposed RST based Web log mining algorithm is given below:

Input: A Web access sequence database with user-id and web access sequence and a minimum support threshold.

Output: The complete set of frequent sequential patterns for Web log database.

Algorithm: (RST based Mining)

Tree Creation:

```
1. Scan the whole database to find the frequent items and frequent subsequences;
2. Create the Root-node of the aggregate tree with zero count value;
3. For each frequent subsequence S'
4.     Scan S' and insert it into the tree;
```

RST Mining:

```
1. For each item α in Σ
2.     Create Root-set of suffix trees rooted at α;
3.     Pattern = α;
4.     Call RST Mine (Root-set, Pattern);
```

```
Function RST Mine (Root-set, Pattern)
1. For each Root-node of Root-set
2.     Call Create Table (Root-node, Table);
3. For each item of Table
4.     If (item. support-count >= μ)
5.         Print (Pattern U item. name);
6.         Call RST Mine (Root-list, Pattern');
```

```
Function Create Table (Root-node, Table)
1. For each child-node of Root-node
2.     If (The child-node not exists in Table)
3.         Insert the child-node into its
           corresponding Root-list on the Table;
4.     Call Create Table (child-node, Table);
```

3.2 An Example: Mining from Aggregate Tree Based on RST

For proper understanding of the proposed approach, the algorithm is briefly described step by step with the help of an example in this section. As for input, the algorithm just takes a web access database and minimum support threshold. The example database is shown in Table 1 which has five user sequences and eight items a, b, c, d, e, f, g and h. Let, the minimum support is 50% i.e. to be frequent; a pattern should appear at least three sequences. From the given web access database we can get three frequent items a: 5, b: 5, and c: 3. Since the support counts for d, e, f, g and h are less than the minimum support count so they are infrequent for the given database. The first column of the table shows the Sequence id of the users, second column presents the web access sequences and finally, the third column contains the frequent subsequences by removing the infrequent items from the sequences. The proposed algorithm given in section 3.1.2 first creates the aggregate tree structure and then mines the tree recursively based on RST (Root-set of Suffix Trees). The steps of algorithm are described below:

Step 1: Scan the given database shown in Table 1 to find the frequent items and frequent subsequences. Frequent items are those, whose support count is greater than or equal to minimum support count, 3. Table 2 shows the frequent items and their corresponding counts. Frequent subsequences are shown in column 3 of Table 1.

Step 2: Scan the frequent subsequences one by one to construct the aggregate tree. At first a Root-node is created with null value of count. Note, each node of the tree contains the item name, support count and the child list. Then scan the first frequent subsequences of Table 1 and insert that subsequence into the tree. Then scan the next subsequences one by one and insert them into the tree. After insertion of all frequent subsequences, Figure 3 shows the final aggregate tree for the given database shown in Table 1.

Table 1. Web Log Sequences

SID	Web Access Sequence	Frequent Subsequence
100	a b d a c	a b a c
200	a e b c a c e	a b c a c
300	b a b a	b a b a
400	a f b a c f c	a b a c c
500	a b e g f h	a b

Table 2. Frequent Items and Their Counts

Frequent Item	Support Count
a	5
b	5
c	3

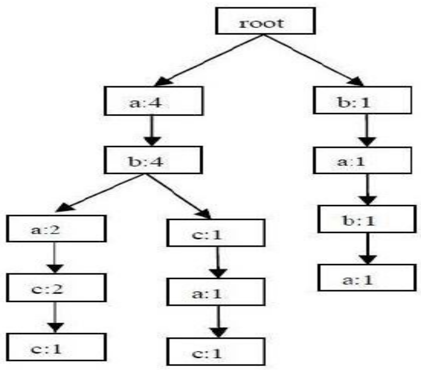


Fig 3: Final Aggregate Tree for the Given Web log sequences

Step 3: Start the mining process from the tree of Figure 3. First, find the Root-sets of suffix trees rooted at ‘a’, 'b' and 'c' and then traverse the tree from the root-nodes of the 'a' root-set, then 'b' and finally 'c'. The initial root-set for 'a' is, [a: 4, a: 1], for 'b' is [b: 4, b: 1] and for 'c' is [c: 2, c: 1] i.e. the first occurrence of nodes for 'a', 'b' and 'c' respectively. After getting the initial root-sets we have to start traversing from the nodes of the root-set for each item. The traversing is started from the root-set with 'a'. During tree traversing we have to store all the first occurrences of frequent items after the root-nodes found during traversing which will be used as new root-sets for the next steps. So, to store the new root-sets found after each traversing we have to maintain a table of root-sets which will contain the nodes information.

Traversing Process: We know that, the root-set nodes for 'a' are, a: 4 and a: 1 which are shown in Figure 4(a) by using color on nodes. We have to start our traversing from the colored nodes labeled with 'a'. We search for the nodes, using the child nodes information of each node. So from Figure 4 (b) we can see that, node a: 4 has only one child b: 4 and after finding the child nodes we also have to search in the table that the found child node already exist or not. If the child node does not exist in the table then we store it into the table otherwise ignore it and go down through the tree for other child nodes. Since, b: 4 is not exists in table so we have to store it and search the child nodes of b: 4. In this case b: 4 has two child a: 2 and c: 1. We will first traverse a: 2 and scan in the table if not exists, then store a: 2 in another row Figure 4 (c) indicates that. Then we will traverse c: 2 child of a: 2 and since c: 2 not present in table so insert it into the table in another row and finally traverse c: 1 which is child of c: 2. Do not need to store c: 1 since we already store a 'c' node for the same branch. Figures 4(d)-(e) show these steps. Since c: 1 has no child node we have to go backward for remaining child nodes and we can see that, c: 2 and a: 2 has no another child but b: 4 has one child which is c: 1. Then we will traverse the c: 1 node and since c: 1 doesn't exist in table so store it into the table in the same row for 'c' nodes which is shown in Figure 4(f). The child node for c: 1 is a: 1 and it is not exist in the table so insert it in the row for 'a' nodes. Since c: 1 child of a: 1 is not the first occurrence of 'c ' for this branch so we will not store c: 1 and c: 1 has no child node so we have to backtrack. Figures 4(g)-(h) show these consequences. There is no another child node for root-node a: 4 so now we have to traverse the tree node from another root-node a: 1. The only one child node for a: 1 is b: 1 and since we did not store any 'b' node for this branch so we have to store b: 1 in the table and finally we store a: 1 which is a child node of b: 1. Figures 4 (i)-(j) show the tables and traversed trees after traversing b: 1 and a: 1. After these we can stop the tree traversing with root-set of suffix trees rooted at 'a'.

From the table which is got after traversing, we can generate the frequent sequential patterns < ab >: 5, < aa >: 4 and < ac >: 3 since the support counts for 'a', 'b' and 'c' are greater than the minimum support count and we also get new root-sets for 'b', 'a' and 'c' prefixed with < a >.

Then start the same process by using the root-set of suffix trees rooted at 'b'. After traversing form 'b' we get two frequent sequential patterns < aba >: 4 and < abc >: 3 prefixed with < ab >. The table for new root-sets got after traversing from 'b' root-set, shown in Table 3.

Then we start traversing using the root-set of 'a' prefix with < ab > and get another frequent sequential pattern < abac >: 3. So the next root-set is [c:2, c:1] prefix with < aba > and after traversing with [c:2, c:1] we don't get any frequent sequential pattern. So after this step we have to backtrack for remaining root-sets and start traversing using those root-sets. In this case, we get root-set for 'c' prefix with < ab > and we don't get any frequent sequential pattern from this step. So in the next step, we have to traverse the tree using the 'a' root-set prefix with < a > and get a frequent sequential pattern < aac >: 3. We then start the traversing from 'c' root-set prefix with < aa > and from this step we cannot get any frequent sequential pattern. Again in the next step we will start traversing using 'c' root-set prefix with <a> and we cannot get any frequent sequential pattern from this step. So we have to backtrack for root-set which will be used for next step and since there is no remaining root-set, we will start traversing using the 'b' root-set [b: 4, b: 1].

After traversing in the same way discussed above, we get two frequent sequential patterns < ba >: 4, < bc >: 3 prefix with < b > and < bac >: 3 prefix with < ba >. Finally, we have to traverse the tree from the nodes of 'c' root-set and in this step we don't get any frequent sequential pattern.

So, the mining process ends after this step and we get full set of frequent sequential patterns for the given database. From the example database, we get total thirteen frequent sequential patterns with 50% minimum support threshold and the patterns are: < a >: 5, < ab >: 5, < aa >: 4, < ac >: 3, < aba >: 4, < abc >: 3, < abac >: 3, <aac >: 3, < b >: 5, < ba >: 4, < bc >: 3, < bac >: 3, and < c >: 3.

Note that the above approach doesn't generate any suffix tree physically instead; it only needs to remember the root-nodes for the current suffix trees which will be used for the next step. Since the algorithm generates multiple root-sets at a single traverse so it needs low execution time than FOF [5], which generates single root-set at a single traverse.

Table 3. Root-sets After Traversing from 'b' Root-set Prefix with 'a'

a:2, a:1,a:1
c:2,c:1

4. PERFORMANCE ANALYSIS

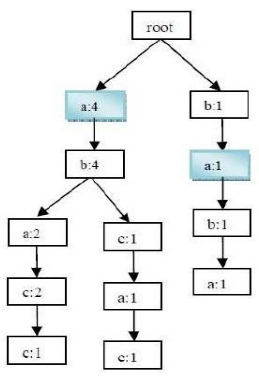
We have compared the performance of our proposed RST-based approach with other two existing approaches WAP-mining [2] and FOF-mining [5] for two datasets. All the experiments were conducted on a 2.40-GHz Intel Pentium(R) 4 processor with 1GB main memory, running on Microsoft Windows XP. All the three algorithms were implemented in NetBeans IDE 6.9 ML with JDK 6. The algorithms for WAP [2] and FOF [5] are also implemented to the best of our knowledge according to the algorithms described in [2] and [5].

4.1 Experimental Datasets

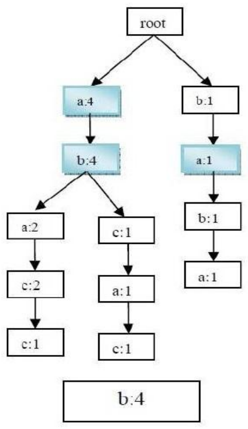
For the performance analysis, we have used two existing datasets Mushroom [8] and Chess [8] which are actually used for frequent pattern mining but in our research we have customized those datasets for web access mining. The properties of these datasets are shown in Table 4. Table 4 characterizes the datasets in terms of the number of distinct items, the number of sequences, the maximum sequence size and the average sequence size.

Table 4. Properties of Experimental Datasets

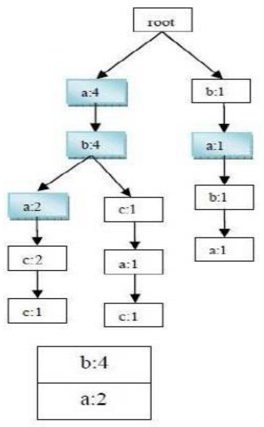
Datasets	Items	No. of Sequences	Max Size	Avg Size
Mushroom	119	8124	23	23.0
Chess	75	3196	37	37.0



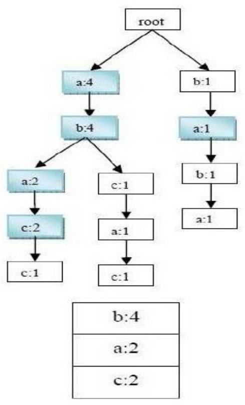
(a) Traversing Start from Root-set [a: 4, a: 1]



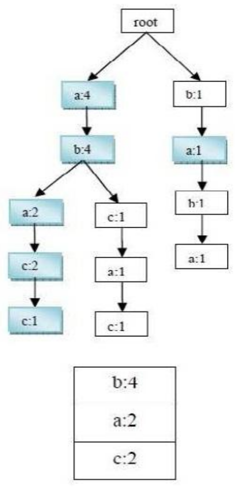
(b) Tree After Traversing b: 4



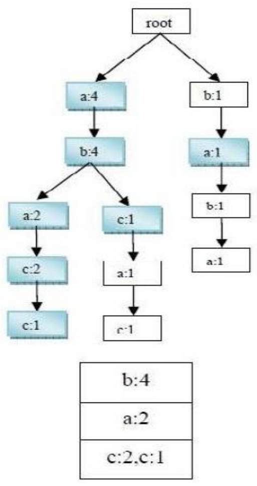
(c) Tree After Traversing a: 2



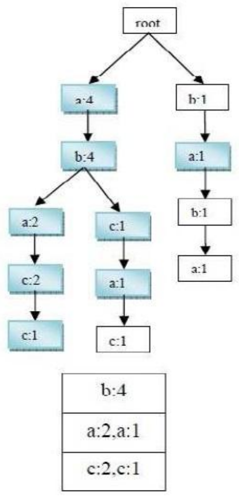
(d) Tree After Traversing c: 2



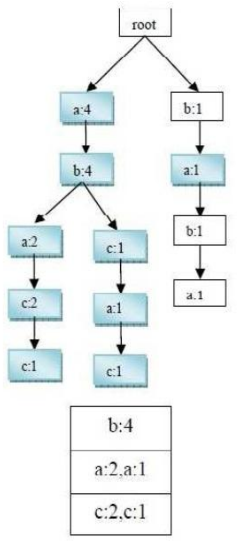
(e) Tree After Traversing c: 1



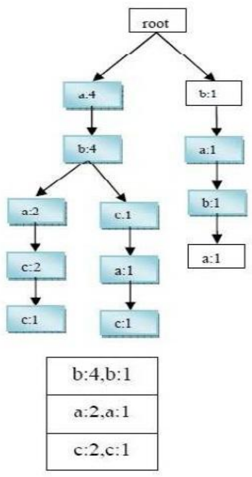
(f) Tree After Traversing c: 1



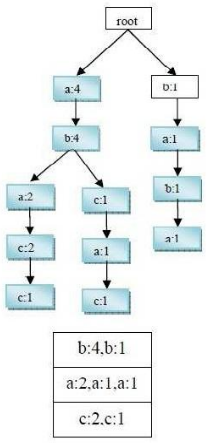
(g) Tree After Traversing a: 1



(h) Tree After Traversing c: 1



(i) Tree After Traversing b: 1



(j) Tree After Traversing a: 1

Fig 4 (a-j): Tree Traversing Process Starting with 'a' Root-set

4.2 Experimental Result

To find out the execution times from the real-life datasets we have used two datasets: Chess and Mushroom .The Chess dataset is derived from its games steps and is obtained from the UC Irvine machine learning repository. Mushroom records drawn from The Audubon Society Field Guide to North American Mushrooms. This dataset includes descriptions of 8124 hypothetical samples corresponding to 23 species of gilled mushrooms in the Agaricus and Lepiota Family. These two datasets produce many long frequent sequences even for very high minimum support threshold. So for our execution we have taken high minimum support thresholds to minimize the runtime.

4.2.1 Chess

Execution times after running three algorithms for different minimum support thresholds by using Chess dataset are plotted in a graph shown in Figure 5.

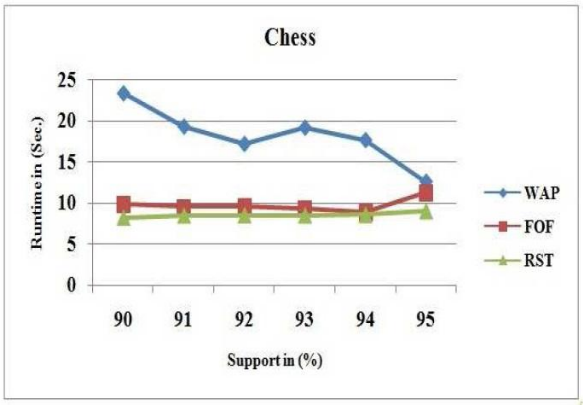


Fig 5: Comparisons between Execution Time and Minimum Support for Chess

4.2.2 Mushroom

Runtime comparisons among WAP [2], FOF [5] and RST for different minimum support thresholds with Mushroom dataset are presenting using a graph shown in Figure 6.

It can be seen from Figure 5 and 6 that, the proposed algorithm outperforms the other two existing algorithms used to find out frequent sequential patterns from the web log databases. It can also be observed that, WAP-mining [2] takes longer times than others two because of creating lots of intermediate trees during mining. FOF-mining [5] takes more time than RST-based mining, because every time it traverses the tree from the roots of forests for each single pattern while RST-traverses the tree from the roots of suffix trees for all length-1 frequent items using a single traverse.

4.3 Correctness and Completeness

The main purpose of all kinds of mining algorithms is to find the complete set of frequent patterns for the specified minimum support threshold. When an algorithm can find those patterns without missing anyone then we can say that the algorithm is complete. On the other hand, the correctness of an algorithm is to find the correct patterns which are interesting for the given minimum support value. When an algorithm has both of these criteria then the algorithm is correct and complete. Like other existing Web log sequential pattern mining algorithms, the proposed algorithm is also correct and complete.

The completeness of the proposed algorithm can be proved by comparing the total numbers of patterns generated for various minimum support thresholds. From Figure 7 and 8 we can see that the proposed algorithm creates the same number of patterns as WAP [2] and FOF [5] generate for datasets Chess and Mushroom. Since for all datasets the proposed algorithm generates complete set of frequent sequential patterns as existing algorithms generate. So it proves the completeness of the algorithm presented in this paper.

Table 5 has presented the sequential patterns obtained from three algorithms for the web log sequences which are shown in Table 1. From this table, we can see that, these three algorithms generate the same frequent sequential patterns with same counts for the same minimum support threshold. In other word, we can say that, the proposed algorithm gives the same result as both WAP [2] and FOF [5] do that proves the correctness of the algorithm presented in this paper.

So, based on the above discussion, it can be said that the proposed algorithm is correct and complete.

Table 5. Sequential Patterns Obtained from Three Algorithms for Database Shown in Table 1

Algorithms	**Total Patterns**	**Sequential Patterns (Pattern : count)**
WAP	13	(a): 5, (ab): 5, (aba): 4, (abac): 3, (abc): 3, (aa): 4, (aac): 3, (ac): 3, (b): 5, (ba): 4, (bac): 3, (bc): 3 and (c): 3.
FOF	13	(a): 5, (ab): 5, (aba): 4, (abac): 3, (abc): 3, (aa): 4, (aac): 3, (ac): 3, (b): 5, (ba): 4, (bac): 3, (bc): 3 and (c): 3.
Our RST-Based Method	13	(a): 5, (ab): 5, (aa): 4, (ac): 3, (aba): 4, (abc): 3, (abac): 3, (aac): 3, (b): 5, (ba): 4, (bc): 3, (bac): 3, and (c): 3.

5. CONCLUSION

This paper presents a new algorithm (RST) for efficiently mining sequential patterns from Web log database. The proposed algorithm uses the aggregate tree structure [7] for storing sequences with frequent items and then starts mining from that tree. For mining, the algorithm uses the last item of common prefix patterns with length k (where k>=1) as Root-set of Suffix Trees rooted with that item. And then traverses the tree from the nodes of root-set and finds the frequent items' counts which are descendants of those root-nodes of the root-set. If the counts of the items found after traversing from the root-set are greater than or equal to the minimum support count then the items will be added after the prefix pattern to generate length k+1 sequential patterns. Then in the next step, each length k+1 pattern will be used as prefix patterns and the nodes label with the last item of the prefix will be used as root-set. This process continues recursively until there exists any frequent sequence found in previous step. Since the algorithm uses the Root-set of suffix trees as pointers; instead of traversing the whole tree every time, it traverses the tree from the nodes of root-set and also it does not need to generate any suffix tree during mining.

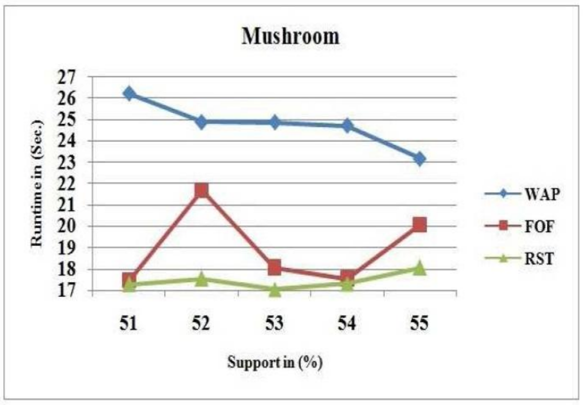


Fig 6: Comparisons between Execution Time and Minimum Support for Mushroom

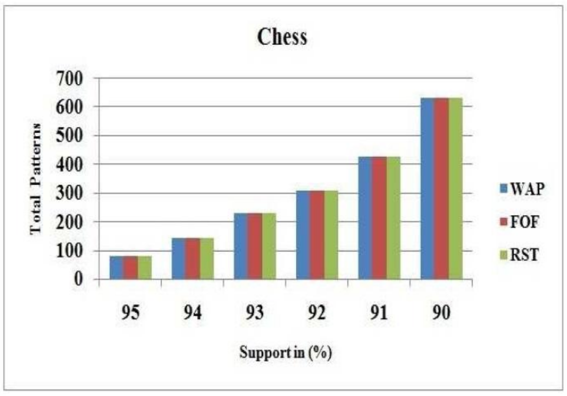


Fig 7: Comparisons between Total Patterns and Minimum Support Values for Chess

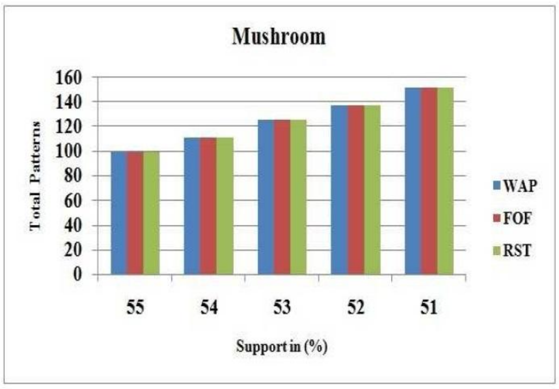


Fig 8: Comparisons between Total Patterns and Minimum Support Values for Mushroom

6. REFERENCES

[1] R. Srikant and R. Agrawal, “Mining sequential patterns: Generalizations and performance improvements," in EDBT, ser. Lecture Notes in Computer Science, P. M. G. Apers, M. Bouzeghoub, and G. Gardarin, Eds., vol. 1057. Springer, 1996, pp. 3-17. [Online]. Available: http://dx.doi.org/10.1007/BFb0014140.

[2] J. Pei, J. Han, B. Mortazavi-Asl, and H. Zhu, “Mining access patterns efficiently from web logs," in PAKDD, ser. Lecture Notes in Computer Science, T. Terano, H. Liu, and A. L. P. Chen, Eds., vol. 1805. Springer, 2000, pp. 396-407. [Online]. Available: http://dx.doi.org/10.1007/3-540-45571-X 47

[3] C. I. Ezeife and Y. Lu, “Mining web log sequential patterns with position coded pre-order linked WAP-tree," Data Min. Knowl. Discov, vol. 10, no. 1, pp. 5-38, 2005. [Online]. Available: http://dx.doi.org/10.1007/s10618-005-0248-3

[4] P. Tang, M. P. Turkia, and K. A. Gallivan, “Mining web access patterns with first-occurrence linked WAP-trees," in SEDE, H. Al-Mubaid and M. Garbey, Eds. ISCA, 2007, pp. 247-252.

[5] E. A. Peterson and P. Tang, “Mining frequent sequential patterns with first-occurrence forests," in ACM Southeast Regional Conference. ACM, 2008, pp. 34-39. [Online]. Available: http://doi.acm.org/10.1145/1593105.1593115

[6] J. Han, J. Pei, and Y. Yin, “Mining frequent patterns without candidate generation," in SIGMOD Conference, W. Chen, J. F. Naughton, and P. A. Bernstein, Eds. ACM, 2000, pp. 1-12. [Online]. Available: http://doi.acm.org/10.1145/342009.335372

[7] M. Spiliopoulou and L. Faulstich, “WUM - A tool for WWW ulitization analysis," in WebDB, ser. Lecture Notes in Computer Science, P. Atzeni, A. O. Mendelzon, and G. Mecca, Eds., vol. 1590. Springer, 1998, pp.184-103.[Online].Available:http://dx.doi.org/10.1007/10704656 12

[8] Z. Zheng, R. Kohavi, and L. Mason, “Real world performance of association rule algorithms," in KDD, 2001, pp. 401-406. [Online]. Available: http://portal.acm.org/citation.cfm?id=502512.502572