Chapter 3

SEQUENTIAL PATTERN MINING – A REVIEW

Sequential Pattern Mining is an important data mining task with broad applications. Apriori and Pattern Growth are the two major heuristics in Sequential Pattern Mining. Sequential Pattern Mining has an important role in web access pattern mining. Efficient mining of web access patterns is crucial in many applications like Web Recommendation, Web Caching, and Web Personalization that are used for making the web more intelligent. WAP-Tree is an efficient tree structure introduced by Pei et al. and WAP-Mine is the corresponding mining algorithm. WAP-Mine outperformed all earlier pattern mining methods. This chapter introduces various approaches in sequential pattern mining, concept of WAP-Tree and WAP-Mine and prominent improvements proposed subsequently.

3.1 Introduction

Sequential pattern mining, which discovers frequent subsequence as patterns in a sequence database, is an important data mining task with broad applications. Customer shopping sequences, medical treatment data, and data related to natural disasters, science and engineering processes data, stocks and markets data, telephone calling patterns, weblog click streams, program execution sequences, DNA sequences and gene expression data are some examples of sequence data.

The sequential pattern mining problem was first introduced by Agrawal and Srikant [Agrawal and Srikant, 1994] based on their study of customer purchase sequences. They stated the problem as follows: Given a set of sequences, where each sequence consists of a list of elements and each element consists of a set of items, and given a user-specified minimum support threshold, sequential pattern mining is to find all frequent subsequences, i.e., the subsequences whose occurrence frequency in the set of sequences is not less than the minimum support.

Let $\boldsymbol{I} = \{\boldsymbol{i_1}, \boldsymbol{i_2}, \boldsymbol{i_3} \ldots \boldsymbol{i_n}\}$ be a set of items. An itemset $\boldsymbol{X}$ is a subset of items i.e. $\boldsymbol{X} \subseteq \boldsymbol{I}$. A *sequence* is an ordered list of itemsets (also called elements or events). A sequence $\boldsymbol{S}$ is

denoted by $\{s_1 s_2 \ldots s_l\}$, where s_j is an itemset (or an element) of the sequence and denoted as $(x_1 x_2 \ldots x_m)$, where x_k is an item. If an element has only one item, the brackets are omitted, i.e., element (x) is written as x. An item can occur at most once in an element of a sequence, but can occur multiple times in different elements of a sequence. The number of instances of items in a sequence is called the *length* of the sequence. A sequence with length l is called an l-sequence. A sequence $\alpha = \{a_1 a_2 \ldots a_n\}$ is called a subsequence of another sequence $\beta = \{b_1 b_2 \ldots b_m\}$ and β a super-sequence of α, denoted as $\alpha \subseteq \beta$, if there exist integers $1 \leq j_1 < j_2 < \ldots < j_n \leq m$ such that $a_1 \subseteq b_{j1}$, $a_2 \subseteq b_{j2} \ldots a_n \subseteq b_{jn}$ [Han, Pei and Yan, 2005].

A sequence database S is a set of tuples <sid, s>, where *sid* is a sequence identifier and s is a sequence. A tuple <*sid*, s> is said to contain a sequence α, if α is a subsequence of s. The support of a sequence α in a sequence database S is the number of tuples in the database containing α, i.e.

$$support_S(\alpha) = \left| \{ < sid, s > | (\langle sid, S \rangle \in S) \wedge (\alpha \in s) \} \right| \quad (3.1)$$

Given a positive integer min-support as the support threshold, a sequence α is called a sequential pattern in sequence database S if $support_S(\alpha) \geq min\text{-}support$. A sequential pattern with length l is called an l-pattern.

Table 3.1: A Sample Database

Seq. Id	Sequence
10	<(bd)cb(ac)>
20	<(bf)(ce)b(fg)>
30	<(ah)(bf)abf>
40	<(be)(ce)d>
50	<a(bd)bcb(ade)>

Example: - Consider the database in Table 3.1. The data base has got 5 sequences. Given min-support=3. Sequential Pattern Mining Problem is to find frequent subsequence with support not less than min-support. First *sequence* <(***bd***)***cb***(***ac***)> has four *elements*: (***bd***), ***c***, ***b***, (***ac***) and ***b*** and ***c*** appear two times. It is a 6-*sequence* since there are 6 instances of items appearing in that sequence.

Table 3.2 shows the 1-length sequences. Among them, five sequence <***a***>, <***b***>, <***c***>, <***d***>, <***e***> have occurrence frequency greater than the minimum support and makes up the 1-length Sequential Patterns.

Table 3.2: 1-Length Sequences of Database in Table: 3.1

1-length sequence	Occurrence
<a>	3
<b>	5
<c>	4
<d>	3
<e>	3
<f>	2
<g>	1
<h>	1

3.2 Various Approaches to Sequential Pattern Mining

3.2.1 Apriori-Based Methods

Since there are usually a large number of distinct single items in a typical transaction database, and their combinations may form a very huge number of itemsets, it is challenging to develop scalable methods for mining frequent itemsets in a large transaction database. Agrawal and Srikant [Agrawal and Srikant, 1994] presented *downward closure* property, called *apriori property* which states that among frequent k-itemsets: *A* k-itemset is frequent only if all of its sub-itemsets are frequent. This implies that frequent itemsets can be mined by first scanning the database to find the frequent 1-itemsets, then using the frequent 1-itemsets to generate candidate frequent 2-itemsets, and check against the database to obtain the frequent 2-itemsets. This process iterates until no more frequent *k*-itemsets can be generated for some *k*. This is the essence of the Apriori algorithm [Agrawal and Srikant, 1994] and its alternatives [Mannila, Toivonen and Verkamo, 1994].

Since its introduction, there have been many studies on efficient mining techniques and extensions of sequential pattern mining method to mine other time-related frequent

patterns [Agarwal and Srikanth, 1995; Bettini et al., 1998; Han, Don and Yin, 1999; Ozden, Ramaswamy and Silberschatz, 1989].

In 1995, Agrawal and Srikant proposed three different versions of apriori algorithm to discover sequential patterns in a transaction database [Agrawal and Srikant, 1995]. The algorithms are AprioriSome, AprioriAll and DynamicSome. Srikant and Agrawal generalized their definition of sequential patterns in [Srikant and Agrawal, 1996a] to include time constraints, sliding time window and user defined taxonomy and developed a generalized sequential pattern mining algorithm, GSP [Srikant and Agrawal, 1996], which outperformed their AprioriAll algorithm [Agrawal and Srikant, 1995]. GSP mines sequential patterns by scanning the sequence database multiple times. In the first scan, it finds all frequent 1-items and forms a set of 1-element frequent sequences. In the subsequent scans, it generates (step-wise longer) candidate sequences from the set of frequent sequences and check their supports. GSP is efficient when the sequences are not long as well as the transactions are not large. However, when the length of sequences increases and/or when the transactions are large, the number of candidate sequences generated may grow exponentially, and GSP will encounter difficulties [Srikant and Agrawal, 1996].

All aforesaid studies on sequential pattern mining adopt an Apriori like paradigm, which promotes a generate-and-test method: first generate a set of candidate patterns and then test whether each candidate has sufficient support in the database (i.e., passing the minimum support threshold test). The Apriori heuristic focus on how a reduced set of candidates can be generated in each of the iteration. However, these algorithms encounter difficulty when the length of the pattern grows long [Pei et al., 2000].

3.2.2 Pattern Growth Approaches

Major difficulty faced in Sequential Pattern Mining is the need for examining combinatorially explosive number of sub sequences for verification of support. Apriori based method substantially reduce the number of sub sequences to be examined. But apriori methods encounter problem when the sequence data base is large and/or when the sequential pattern to be mined are numerous and /or large.

Apriori based methods bear three nontrivial, inherent costs which are independent of implementation procedure [Han et al., 2000].

i. *When the data base is very large, huge number of candidate sequenced are generated*.

All possible permutation of elements in a K-frequent item set are the set of candidate sequences for building k+1 frequent pattern.

ii. *Pattern mining requires multiple scan of the database*.

Length of each candidate sequence grows by one character upon each data base scan.

iii. *Mining of long patterns brings difficulty in Apriori based methods*.

The generation of long patterns from a huge number of small candidate sequences introduces problems in apriori based methods. This is because the number of candidate sequences are exponential to the length of patterns to be mined [Han et al., 2000].

Pattern-growth methods are introduced to deal with the problems of sequential pattern mining. The key idea is to avoid the candidate generation step altogether, and to focus the search on a restricted portion of the initial database. FreeSpan [Han et al., 2000] and PrefixSpan [Pei et al., 2001] are two algorithms adopting this divide and conquer approach. In this approach, the sequence database are recursively projected into a set of smaller projected database based on the current frequent pattern(s), and sequential patterns are grown in each projected databases by exploring only locally frequent fragments [Han et al., 2000].

General idea in FreeSpan (Frequent Pattern Projected Sequential Pattern Mining) [Han et al., 2000] is to use frequent items to recursively project sequence databases into a set of smaller projected databases and grow subsequence fragments in each projected database. This process partitions both the data and the set of frequent patterns to be tested, and confines each test being conducted to the corresponding smaller projected database. Performance study claims that FreeSpan mines the complete set of patterns and is efficient and runs considerably faster than the apriori based GSP algorithm. However, since a subsequence may be generated by any substring combination in a sequence, projection in FreeSpan has to keep the whole sequence in the original data base without length reduction. Moreover, since the growth of a subsequence is explored at any split point in a candidate sequence, it is costly [Pei et al., 2001].

Prefix Span [Pei et al., 2001] examines only the prefix subsequences and project only their corresponding postfix sub-sequences into projected databases. In each projected database, sequential patterns are grown by exploring only local frequent patterns. To further improve mining efficiency, two kinds of database projections are explored:

level-by-level projection and bi-level projection. PrefixSpan mines the complete set of patterns and is efficient and runs considerably faster than both apriori based GSP algorithm [Srikant and Agrawal, 1996] and FreeSpan [Han et al., 2000].

Pei et al. improved the performance of PrefixSpan in [Pei et al., 2004] by introducing the concept of Pseudo-Projection. In Pseudo-Projection instead of performing physical projection, the index (or identifier) of the corresponding sequence and the starting position of the projected suffix in the sequence is registered. Then, a physical projection of a sequence is replaced by registering a sequence identifier and the projected position index point. Pseudo projection reduces the cost of projection substantially when the projected database can fit in main memory. Through a comprehensive performance study the authors [Pei et al., 2004] showed that PrefixSpan outperforms the apriori-based GSP algorithm, FreeSpan, and SPADE in most cases and PrefixSpan integrated with pseudo projection is the fastest among all. PrefixSpan (Prefix-projected Sequential pattern mining) is the most promising of the pattern-growth methods and is based on recursively constructing the patterns, and simultaneously, restricting the search to projected databases [Antunes and Oliveira, 2004].

3.2.3 Mining Frequent Itemsets using Vertical Data Format

Both the Apriori and Pattern growth methods mine frequent patterns from a set of transactions in horizontal data format (i.e., {TID: itemset}), where TID is a transaction-id and itemset is the set of items bought in transaction TID. Alternatively, mining can also be performed with data presented in vertical data format (i.e., {item: TID set}).

SPADE proposed by Mohammed Zaki [Zaki, 2001] follows this approach. It scans database three times. The transformation into vertical format requires both time and more memory space. Moreover, the basic search methodology of SPADE and GSP is breadth-first search and Apriori pruning. Both algorithms have to generate large sets of candidates in order to grow longer sequences.

3.3 Web Access Pattern Mining

3.3.1 Introduction

Each and every user’s interaction with web is recorded in a Server log. Web logs can be considered as a sequence of web pages accessed by users. Web Usage Mining, also

known as Web log mining, discover interesting and frequent user access patterns from the web browsing details stored in server web logs, proxy server logs or browser logs [Pei et al., 2000]. Web Access Pattern information is critical in improving designs of web sites, analyzing system performance as well as network communications, understanding user reaction and motivation, and building Adaptive Web sites [Srivastava et al., 2000].

3.3.2 Web Access Sequence

A Web Log is a sequence of pairs: user-id and access information. Each entry in Web Access Sequence Database (WASD) is a sequence of events from one user or session in time stamp order. For the purpose of study of sequential pattern mining, preprocessing [Pei et al., 2000] is applied to the original log file and WASD is generated.

Let *E* be a set of access events, which represents web pages accessed by users. A Web Access Sequence (WAS) is an ordered sequence that gives the order in which a particular user accesses web pages in a session.

A Web Access Sequence *S* may be defines as

$$S = e_1 e_2 \dots e_n \mid e_i \in E \wedge 1 \leq \mathrm{i} \leq n \tag{3.2}$$

e_i and e_j are not necessarily different for $i \neq j$, that is, repetition of items is allowed. For example, ***aab*** and ***ab*** are two different access sequences, in which ***a*** and ***b*** are two events.

Length of access sequence $S = e_1 e_2 \dots e_n$ is defined as

$$\mathrm{len}(S) = |S| = n \tag{3.3}$$

An access sequence *S* with length *n* is called an n-sequence [Pei et al., 2000]. The empty sequence ε is a special web access sequence of length 0 and $\boldsymbol{\varepsilon}.S = S.\boldsymbol{\varepsilon} = S$ for any sequence *S* where '.' is the concatenation operator [Tang, Turkia and Gallivan, 2007].

3.3.3 Web Access Pattern

A *Web Access Sequence Database* (WASD) is a multi-set of web access sequences including the possible empty sequence.

That is, WASD = $\{S_1, S_2 \dots S_m\}$ where S_i is a Web Access Sequence.

A Web Access Sequence $S'=s_1's_2'...s_n'$ is a subsequence of sequence $S=s_1s_2...s_m$, if and only if $n \leq m$ and there exist $i_1, i_2...i_n$ such that $1< i_1<i_2...<i_n \leq i_m$ and $s'_j = s_{ij}$ for all $1 \leq j \leq n$. The empty sequence $\boldsymbol{\varepsilon}$ is a subsequence of any sequence [Pearson and Tang, 2008]. Access sequence S' is a proper subsequence of sequence S, denoted as $S' \subset S$, if and only if S' is a subsequence of S and $S' \neq S$.

In access sequence $S = e_1e_2 \ldots e_ke_{k+1} \ldots e_n$, let the subsequence $S_{suffix} = e_{k+1} \ldots e_n$ is a super sequence of pattern $P = e'_1e'_2 \ldots e'_l$ where $e_{k+1} = e'_1$. Then the subsequence of S, $S_{prefix} = e_1e_2 \ldots e_k$, is called the prefix of S with respect to pattern P and S_{suffix} is called the suffix sequence of S_{prefix}.

A web access sequence S in WASD is said to support pattern p, if p is a subsequence of S. The support of pattern p in WASD, denoted as $\mathrm{Sup}_{\mathrm{WASD}}(p))$, is the number of web access sequences in WASD that support pattern p.

That is,

$$\mathrm{Sup}_{\mathrm{WASD}}(\mathcal{P}) = \frac{|\{S_i \mid p \subseteq S_i, S_i \in \mathrm{WASD}\}|}{|\mathrm{WASD}|} \quad (3.4)$$

Given a Web Access Sequence Database WASD and a support threshold ξ in the interval [0:1], an access sequence S is frequent with respect to ξ, if $\mathrm{Sup}_{\mathrm{WASD}}(p)) \geq \xi \times |\mathrm{WASD}|$, where $|\mathrm{WASD}|$ is the number of web sequences in WASD. $\xi \times |\mathrm{WASD}|$ is called the absolute support threshold and denoted as η. A frequent access sequence of WASD satisfying the support threshold ξ is said to be an *access pattern* of WASD with respect to ξ or simply an *ξ-pattern* of WASD [Pei et al., 2000].

3.3.4 Web Access Pattern Mining

Given a Web Access Sequence Database, WASD and a support threshold ξ, Web Access Pattern Mining mines the complete set of ξ-patterns of WASD [Pei et al., 2000].

As a Web access sequence contains page view information of a user in a session in a time stamp order, sequential pattern mining can be adapted for mining the frequent access patterns in the set of web access sequences, WASD.

As discussed in *Section 3.2*, there are mainly two characteristics for sequential pattern mining techniques, namely (i) the mode of candidate sequences generation and its storage and (ii) the mode of support counting. Based on these characteristics there are

two major approaches for sequential pattern mining: - apriori based methods and pattern growth based methods.

In sequential pattern mining, Apriori property substantially reduces the size of candidate sets. However, because of the combinatorial nature of the sequential pattern mining, it may still generate a huge set of candidate patterns, especially when the sequential pattern is long, which is exactly the case of Web Access Pattern Mining [Pei et al., 2000].

This implies that for web access pattern mining, pattern growth sequential pattern mining method is the best suitable one. The key consideration is how to facilitate the tedious support counting and candidate generating operations in the mining procedure.

Pei et al. proposed an efficient tree structure WAP-Tree to hold the access sequence compactly and a corresponding access pattern mining algorithm WAP-Mine [Pei et al., 2000]. The authors proved experimentally that WAP-Mine algorithm outperforms the earlier apriori based access pattern mining algorithms [Agrawal and Srikant, 1995; Srikant and Agrawal, 1996]

Subsequent to the introduction of WAP-Tree, several proposals have been made by researchers to improve the performance of mining [Zhou, Hui and Fong, 2004]. Next section presents a detailed discussion of some prominent WAP-Tree based algorithms.

3.4 WAP-Tree Based Access Pattern Mining Algorithms

3.4.1 Introduction

This section presents WAP-Mine algorithm and other four algorithms, PLWAP-Mine, CS-Mine, FLWAP-Mine and FOF-Mine that present significant modifications to WAP-Mine to improve the performance of mining. As WAP-Mine is the base algorithm illustration with a simple example is also provided. Since CS-Mine and FOF-Mine are used subsequently, these algorithms are also dealt in detail.

3.4.2 WAP-Mine Algorithm

Pei et al. [Pei *et a*l., 2000] proposed a compressed data structure known as Web Access Pattern Tree (WAP-Tree) and WAP-Mine (Web Access Pattern Tree Mining) algorithm for mining web access patterns efficiently from web logs.

WAP-Mine algorithm is a pattern growth method based on the suffix heuristic: if ***a*** is a frequent event in the set of prefixes of sequences in Web Access Sequence Database (WASD) with respect to pattern *P*, then sequence ***aP*** is an access pattern of WASD. WAP-Tree is used to register all access sequences and their corresponding count very compactly and maintains linkages for traversing prefixes with respect to the same suffix pattern. As WAP-Tree is much smaller than the original database the processing becomes easier.

WAP-Mine is the recursive mining algorithm used to generate access patterns from the WAP-Tree. WAP-Mine first scans WAS to find all frequent events, events that appear in at least $\xi \times |\text{WASD}|$ access sequence of WASD, where $|\text{WASD}|$ is the number of access sequences in WASD and ξ is the support threshold. WASD is scanned again and WAP-Tree is constructed over the set of frequent events. WAP-Mine algorithm recursively mines the WAP-Tree using conditional search which is based on partition-based Divide-and-Conquer method.

Steps in the WAP-Mine Procedure can be summarized as follows: -

1. Scan WASD once, find all frequent events.
2. Scan WASD again; construct a WAP-Tree over the set of frequent events.
3. Recursively mines the WAP-Tree using conditional search.

Only frequent 1-sequences are considered for constructing WAP-Tree, as only 1-sequences are useful in generating k-sequences, where k>1. Initial Step scans the database and collects the support of each of the element from the access sequences. Common prefixes are shared in the WAP-Tree to save space. Each WAP-Tree node registers two pieces of information, label and count. The root of the tree is a special virtual node with an empty label and count 0.

Construction of WAP-Tree: For each access sequence, frequent subsequence is entered into the tree, starting from the root node. While inserting the first event ***e***, if the current node has already a child e, then the count of the child node with event ***e*** is incremented by 1, otherwise a new child with label ***e*** and count 1 is created. The rest of the subsequence is recursively inserted to the sub tree rooted at the current ***e*** node. All nodes with same label ***e*** are linked by shared label linkages into a queue, called event-queue. Event queue for $\boldsymbol{e}_i$ is called $\boldsymbol{e}_i$-queue. Head of each event-queue is registered in a header table H.

A simple pre-processed web access sequence database, with the set of access events $E = \{a, b, c, d, e, f\}$ is shown in Table 3.3 [Pei et al., 2000].

Table 3.3: Sample Web Access Sequence Database for WAP-Tree

User ID	Web Access Sequence	Frequent Sub- sequence
100	*abdac*	*abac*
200	*eaebcac*	*abcac*
300	*babfaec*	*babac*
400	*afbacfc*	*abacc*

Assume the minimum support threshold to be 75%. So, an access sequence is considered to be frequent if it is present in 3 out of 4 records in the example. Constructing WAP-Tree entails first scanning database once, to obtain events that are frequent. The frequent sub-sequences are formed by discarding the non frequent elements from each sequence. For example, in Table 3.3, the list of all events is *a, b, c, d, e, f* , and the support of ***a*** is 4, ***b*** is 4, ***c*** is 4, ***d*** is 1, ***e*** is 2, and ***f*** is 3.With the minimum support of 3, only *a, b, c* are frequent events. Thus, all non-frequent events, *d, e and f* are deleted from each transaction sequence to obtain the frequent subsequence shown in column three of Table 3.3.

Figure 3.1 shows how the frequent subsequences are entered and linkages are made for the sequences in Table 3.3. The WAP-Tree with linkage header for the given WASD is given in Figure 3.1d. All pattern information related to a frequent event $\boldsymbol{e_i}$ can be accessed by following all the branches in WAP-Tree linked by $\boldsymbol{e_i}$-queue only once.

All nodes in the path from root of the tree to node $\boldsymbol{e_i}$ (excluded) form a prefix sequence of $\boldsymbol{e_i}$ and the count of this node is the count of the prefix sequence. A prefix sequence of $\boldsymbol{e_i}$ may contain another prefix sequence of $\boldsymbol{e_i}$. So, to avoid double counting that can occur, concept of un-subsumed count is introduced. Let G and H be two prefix sequences of $\boldsymbol{e_i}$ and G is also formed by the sub-path from root that H is formed by, H is called a super-prefix sequence of G, and G is a sub-prefix sequence of H. For a prefix

sequence of e_i without any super-prefix sequences, the un-subsumed count is the count of it. For a prefix sequence of e_i with some super-prefix sequences, the un-subsumed count is the count of that sequence minus un-subsumed counts of all its super-prefix sequences. Access patterns with same suffix are now used for searching all web access patterns. As suffix becomes longer, the remaining search space becomes smaller potentially.

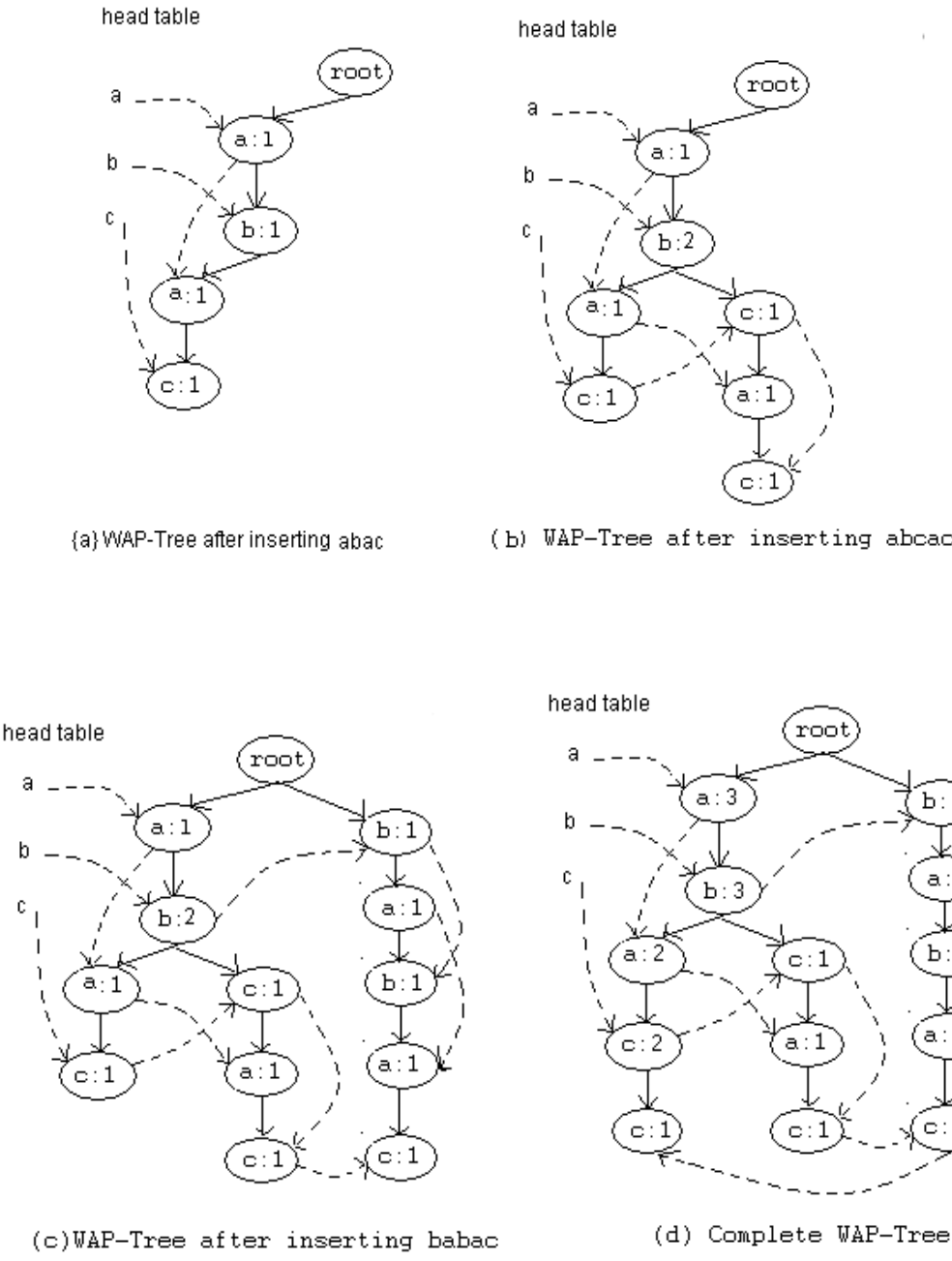

Figure 3.1: Construction of the WAP Tree [Pei et al., 2000]

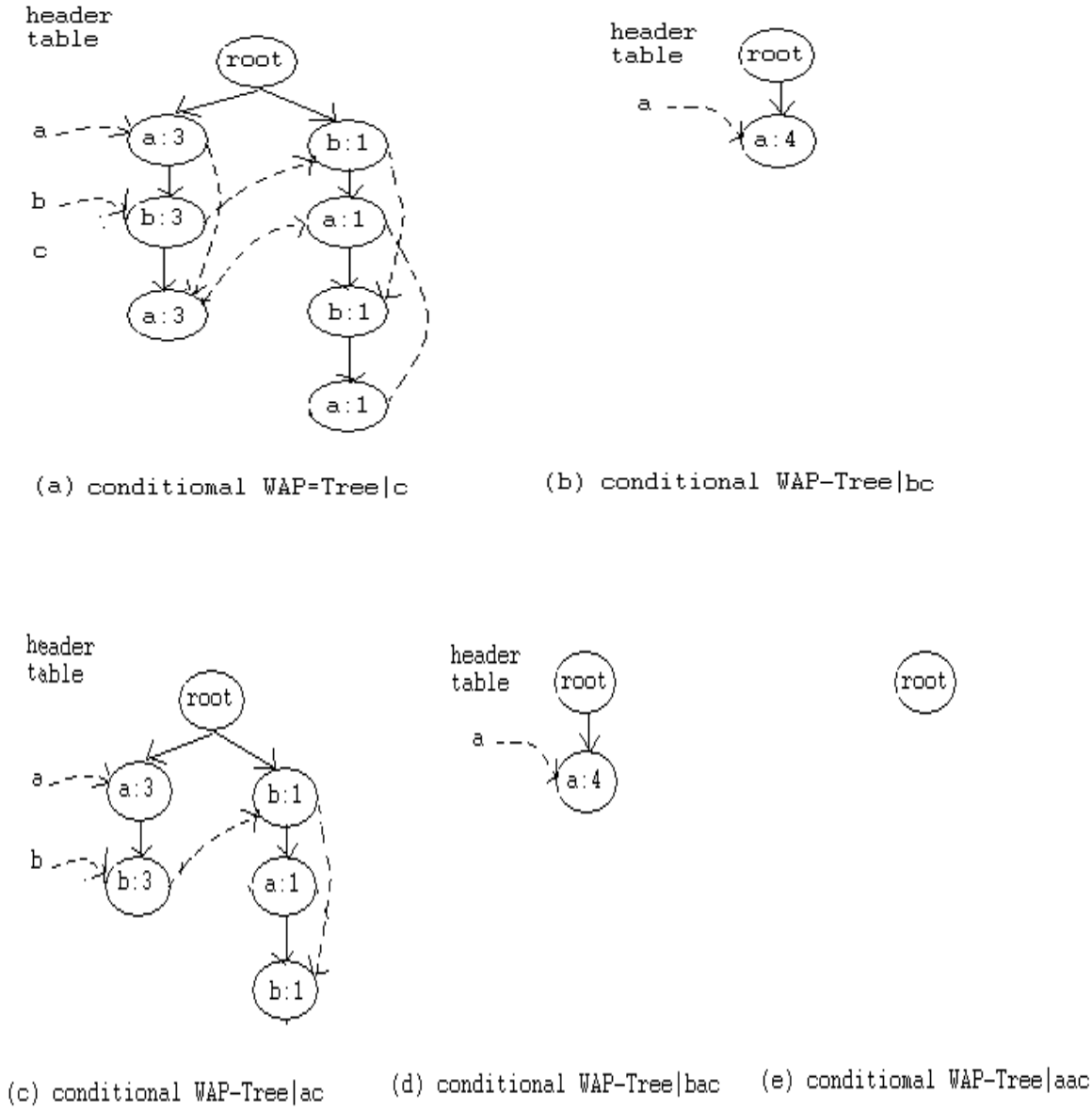

Figure 3.2: Reconstruction of WAP-Trees for Mining Conditional Pattern Base ***c***

WAP-Tree Mining: For each event e_i in the header list conditional sequence base is found. The conditional sequence list of a suffix event is obtained by following the header link of the event and reading the path from the root to each node (excluding the node). The count of the prefix sequence is same as the count of the node. For each prefix sequence inserted into the conditional sequence base with count ***c***, all its sub-prefix sequences are inserted with ***–c*** as count, to get the un-subsumed count. Then the set of conditional frequent events is found. If it is not empty, recursive mining is done

on the conditional WAP-Tree of each frequent event. When there is only one branch in the conditional WAP-Tree, all combinations are generated as patterns.

Once the sequential data is stored on the complete WAP-tree [Figure 1(d)], the tree is mined for frequent patterns starting with the lowest frequent event in the header list. In the example, mining starts with frequent event ***c***. From the WAP-tree of Figure 3.1(d), it first computes the conditional sequence base of ***c*** *as*:***aba***:2; ***ab***:1; ***abca***:1; ***ab***:−1; ***baba***:1; ***abac***:1; ***aba***:−1. The conditional sequence list of a suffix event is obtained by following the header link of the event and reading the path from the root to each node (excluding the node). The count for each conditional base path is the same as the count on the suffix node itself. But to qualify as a conditional frequent event, it must have a count of 3. Therefore, after counting the events in sequences above, the conditional frequent events ***a***(4) and ***b***(4) are frequent. So, the conditional sequences based on ***c*** are:-

aba*:**2; ***ab:1; ***aba***:1; ***ab***:−1; ***baba***:1; ***aba***:1; ***aba***:−1

Using these conditional sequences, a *conditional WAP tree*, WAP-tree|***c***, is built as in Figure 3.2(a). Recursively, based on the WAP-tree in Figure 3.2(a), the next conditional sequence base for the next suffix subsequence, ***bc*** is found as ***a***(3), ϵ, ***ba***(1) with ***a*** as the only frequent pattern. Next conditional WAP tree is constructed as in 2(b). This ends the re-construction of WAP trees that progressed as suffix sequences |***c***, |***bc*** and the frequent patterns found along this line are ***c***, ***bc*** and ***abc***.

The recursion continues with the suffix path |***c***, |***ac***, |***bac*** and intermediate trees are generated using Figure 3.2(a) as in Figure 3.2(c) and Figure 3.2(d). To complete the mining of frequent patterns with suffix ***ac***, Figure 3.2(c) is mined for suffix ***aac*** to find ***b***: 1. But it is discarded due to less support. The empty conditional WAP-tree with the suffix ***aac*** is shown in Figure 3.2(e).

The search for frequent patterns that have the suffix of other header frequent events (starting with suffix base |***b*** and then |***a***) are also mined the same way the mining for patterns with suffix ***c*** is done above. After mining the whole tree, discovered frequent pattern set is: {***c***, ***aac***, ***bac***, ***abac***, ***ac***, ***abc***, ***bc***, ***b***, ***ab***, ***a***, ***aa***, ***ba***, ***aba***}.

3.4.3 PLWAP-Mine Algorithm

Y. Lu, and C. I. Ezeife propose Pre-Order Linked WAP-Tree Mining (PLWAP-Mine) and a method of assigning binary position code to tree nodes [Lu and Ezeife, 2003]. PLWAP-Mine algorithm based on WAP-Tree avoids recursive re-construction of

intermediate WAP-Trees. The method attaches binary position codes to the nodes of the tree to determine the suffix trees of any frequent pattern prefix under consideration by comparison. That is, PLWAP-Mine uses the suffix trees of the last frequent event in an *m*-prefix sequence to recursively extend the subsequence to ***m***+1 sequence by adding a frequent event that occurred in the suffix trees.

Position code is a binary code used to indicate the position of the nodes in the WAP-Tree. The position code is assigned to the nodes on the binary tree equivalent of the tree using the Huffman coding idea. Given a WAP-Tree with some nodes, the position code of each node is assigned following the rule that the root has null position code, and the leftmost child of the root has a code of 1, but the code of any other node is derived by appending 1 to the position code of its parent, if this node is the leftmost child, or appending 10 to the position code of the parent if this node is the second leftmost child, the third leftmost child has 100 appended, etc. In general, for the n^{th} leftmost child, the position code is obtained by appending the binary number for $2n-1$ to the parent's code. A node **α** is an ancestor of another node **β** if and only if the position code of **α** with "1" appended to its end, equals the first ***x*** number of bits in the position code of **β**, where ***x*** is the number of bits in the position code of **α** + 1. Figure 3.3 shows the position code assignments to the nodes of PLWAP-Tree.

PLWAP Algorithm mainly includes functions to build PLWAP-Tree and to mine patterns from PLWAP-Tree.

Building of PLWAP-Tree: Root node is created with position code NULL and *count* 0. For each access sequence *S* in the access sequence database WASD, frequent subsequence is extracted and entered into the tree just like the creation of WAP-Tree [Pei *et.al*, 2000]. A header table with all frequent events is constructed next. Now the constructed tree is traversed in pre-order and each node e_i is attached to the corresponding e_i-queue and head of each queue is registered in corresponding header table entry. Major difference in the construction of a PLWAP-Tree and a WAP-Tree is in the generation of binary code for nodes along with tree creation and in the generation of the pre-order linkages.

PLWAP-Mine: Mining starts by finding the frequent 1-sequence {***a***, ***b***, ***c***}. Then for every frequent event and the suffix trees of current conditional PLWAP-Tree being mined, it finds the first occurrence of this frequent event in every suffix tree being

mined following the pre-order linkage of this event, and adds the support count of all first occurrences of this frequent event. If the count is greater than the minimum support threshold, then this event is concatenated to the previous list of frequent sequence, *F*. Now, the suffix trees of these first occurrence events in the previously mined conditional suffix PLWAP-Trees are used for mining the next event.

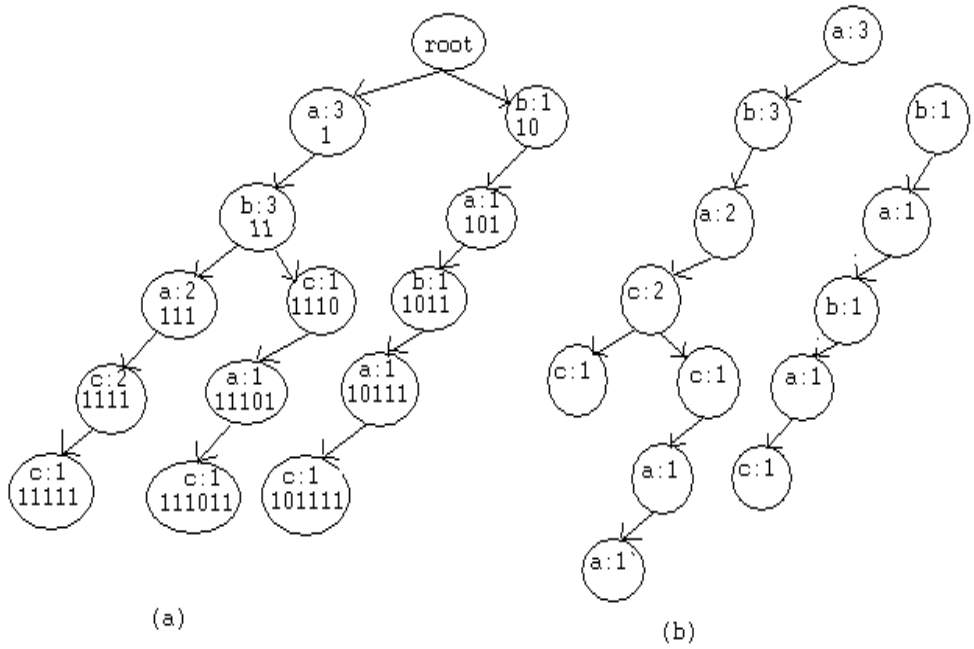

Figure 3.3: Position Code Assignments with Node Positions in its Binary Tree [Lu and Ezeife, 2003]

3.4.4 CS-Mine

In [Zhou, Hui and Fong, 2004], the authors proposed a method CS-Mine (Conditional Sequence Mining) based on WAP-Tree structure [Pei et.al, 2000]. CS-Mine consists of five processes: Constructing Initial Conditional Sequence Base (CSB), Constructing Event Queues (EQ) for CSB [Figure 3.4], Constructing Sub-CSB [Figure 3.5], Single Sequence Testing for CSB [Figure 3.6] and Recursive Mining for Sub-CSB [Figure 3.7].

Algorithm: ConstructEQ

Input:

1: *MinSup* - support threshold

2: *CSB*(*Sc*) - conditional sequence base of conditional suffix *Sc*

3: $E = \{ei | 1 \leq i \leq n\}$ – all events in *CSB*(*Sc*)

Output: the *CSB*(*Sc*) with Header Table *HT* and Event Queues

Method:

Create an empty Header Table for *CSB*(*Sc*).

For each event *ei*∈*E*, if sup(*ei*)≥ *MinSup*, insert *ei* into *HT*.

For each conditional sequence ∈ *CSB*(*Sc*) do

(a) For each event *ei*∈*HT* of *CSB*(*Sc*), insert the last item labelled *ei* of this sequence into *ei*-queue.

(b) Delete all items of events ∉ *HT* from this sequence.

Return *CSB*(*Sc*) with *HT* and Event Queues.

Figure 3.4: Algorithm for Constructing Event Queues for CSB

The conditional sequence base of event $\boldsymbol{e_i}$ based on suffix sequence *Ssuffix*, is the set of all long prefix sequences of $\boldsymbol{e_i}$ in sequences of a database. If *Ssuffix* is empty, the conditional sequence base is the frequent sub-sequence set of the given database. Otherwise, it is the conditional sequence base CSB(*Ssuffix*).

Algorithm: ConstructSubCSB

Input:

1: *CSB*(*Sc*) - conditional sequence base of conditional suffix *Sc*

2: $\boldsymbol{e_i}$ - a given event in Header Table of *CSB*(*Sc*)

Output: *CSB*($\boldsymbol{e_i}$ +*Sc*) - conditional sequence base of $\boldsymbol{e_i}$ of *CSB*(*Sc*)

Method:

Initialize sub-conditional sequence base *CSB*($\boldsymbol{e_i}$ *i*+*Sc*) = ∅.

For each item in $\boldsymbol{e_i}$-queue of *CSB*(*Sc*), insert its prefix sequence into *CSB*($\boldsymbol{e_i}$ +*Sc*).

Return *CSB*($\boldsymbol{e_i}$ +*Sc*).

Figure 3.5: Algorithm for Constructing Sub-CSB

CS-Mine starts by constructing the initial conditional sequence base by mining WAP-Tree. For each frequent event $\boldsymbol{e_i}$ in the given WAP-Tree, all the prefix sequences of all the nodes in the $\boldsymbol{e_i}$ -queue form the Initial Conditional Sequence Base of event $\boldsymbol{e_i}$, Init-CSB(e_i). The count of related suffix node in the e_i-queue is the count of each prefix

sequence. To avoid duplicate counting, for each prefix sequence of e_i with count $\boldsymbol{n}$, all of its sub-prefix sequences are also inserted into Init-CSB($\boldsymbol{e_i}$) with minus count, $\boldsymbol{-n}$.

Algorithm: TestCSB

Input:

1: *CSB*(*Sc*) – conditional sequence base of conditional suffix *Sc*

2: *HT* – Header Table of *CSB*(*Sc*)

Output:

1: test result - successful or failed

2: *SingleSeq* - single sequence of *CSB*(*Sc*)

Method:

Initialize the single sequence of *CSB*(*Sc*) *SingleSeq* = ∅.

If *CSB*(*Sc*) = ∅, test is successful and return *SingleSeq* = ∅.

For i = 1 to the maximum length of sequences ∈ *CSB*(*Sc*) do

(a) If all the i^{th} items in each sequence ∈ *CSB*(*Sc*) are the same event ***e***, create a new item ***e*** with the total count of these items and insert it into *SingleSeq*.

(b) Otherwise, test is failed and returns *SingleSeq* = ∅.

Test is successful and returns *SingleSeq*.

Figure 3.6: Algorithm for Testing of Conditional Sequence Base

To construct event queues for each conditional sequence base CSB(*Sc*) (Sc= $\boldsymbol{e_i}$ for Init-CSB), conditional frequent events from CSB(*Sc*) are identified first. Conditional frequent events are events in CSB(*Sc*) with support of not less than Minimum Support. Header Table is created with all the conditional frequent events. Then, for each conditional frequent event $\boldsymbol{e_i}$, a linked-list structure called $\boldsymbol{e_i}$–queue, connecting the last item labeled $\boldsymbol{e_i}$ in the sequences of CSB (*Sc*), is created. The head pointer of each event queue is recorded in the Header Table. Finally, all non-frequent events in sequences in CSB (*Sc*) are discarded.

Algorithm: CS-Mine

Input:

1: *MinSup* - support threshold

2: *T* - WAP-Tree of a web access sequence database with *MinSup*

3: $E = \{ e_i \mid 1 \le i \le n\}$ – all events in Header Table of WAP-Tree *T*

Output: *WAP* - the set of web access patterns

Method:

Initialize the set of web access patterns $WAP = \emptyset$.

For each event $e_i \in E$ do

(a) Set Conditional Suffix $Sc = e_i$, and construct *Init-CSB*(*Sc*) by following e_i-queue in WAP-Tree *T*.

(b) Use ConstructEQ to construct event queues for CSB(Sc).

(c) Use **TestCSB** to test single sequence for *CSB*(*Sc*).

(d) If test is successful, insert all ordered combinations of items in frequent sequence *FS* = *SingleSeq*+*Sc* into*WAP*.

(e) Otherwise, for each e_j in Header Table of *CSB*(*Sc*), use **ConstructSubCSB** to construct *CSB*(e_j+*Sc*), set $Sc = e_j$+*Sc*.

Recursively mine *CSB*(*Sc*) from step 2(b).

Return *WAP*.

Figure 3.7: CS-Mine, Algorithm for Mining Web Access Patterns

CSB(e_i +*Ssuffix*) is called the sub-conditional sequence base of CSB(*Ssuffix*), if e_i is not null. The *Construct-SubCSB* algorithm is used for constructing CSB(e_i+*Sc*) based on CSB(Sc) for each event e_i in the Header Table of CSB(*Sc*). Single Sequence Test checks whether all sequences in CSB(*Sc*) can be combined into a single sequence. If so, the mining of *CSB(Sc)* will be stopped. Otherwise, we construct Sub-CSB for each event in the Header Table based on *CSB(Sc)* and perform recursive mining.

For example, assume *CSB*(***a***) is {***cab***: 2, ***cacb***:1, ***ccab***:1}. First item of each sequence is ***c***. So, they can be combined into one item *c* with the total count 4, denoted as (***c***:4). But the second items of sequences are not same. So, Single sequence test fails. But *CSB* (***aa***) is {***c***: 3, cc: 1}. The sequences can be combined into one single sequence (***c*:** 4) (***c***:1) and the test is successful. Single sequence (***c***: 3) form a part of the final web access patterns and ***cc:***1 is neglected as it fails to satisfy support requirement. In the WAP-mine algorithm [Pei et.al, 2000], recursive construction of conditional WAP-Tree is continued until the tree has only one branch. Instead of this, CS-Mine uses Single Sequence Testing method efficiently.

3.4.5 FLWAP-Mine

P. Tang *et al*. introduced First Occurrence Linked WAP Tree (*FLWAP* Tree) and presented a pattern mining algorithm *FLWAP-Mine* to mine *FLWAP* Tree [Tang, Turkia and Gallivan, 2007]. They also describe the theory of conditional searching on which their pattern-growth mining algorithms are based.

Given a web access sequence *S* and a symbol ***a*** from the set of symbols ∑, such that ***a*** is in *S*, the ***a***-prefix of ***S*** is the prefix of ***S*** from the first symbol to the first occurrence of ***a*** inclusive. Thus ***a***-prefix of sequence ***bcabad*** is ***bca***. The ***a***-projection of ***S*** is what is left after the ***a***-prefix is deleted. That is, the ***a***-projection of sequence ***bcabad*** is ***bad***. If ***a*** occurs only once as the last symbol in *S*, the ***a***-prefix is *S* and the ***a***-projection is the empty sequence **ε** [Tang, Turkia and Gallivan, 2007].

Given the database *D* and a symbol ***a*** in ∑, the ***a***-projection database of *D*, denoted as *Da*, is the multi-set of ***a***-projections of the web access sequences in *D* that support ***a***.

$$Da=\{\boldsymbol{a}\text{\_projections of S}/\boldsymbol{a} \subseteq \text{S} \wedge \text{S} \in \text{D}\} \tag{3.5}$$

Da, the ***a***-projection database is used to grow frequent patterns by using symbols from ∑. It is sufficient to use the sub-trees rooted at the children of the first-Occurrences of ***a***, to represent projection database *Da*, ignoring possible empty sequences. Since first-Occurrences play a central role in finding the support of a symbol and its projection database, all pattern-growth mining algorithms based on aggregate trees [Tang, Turkia and Gallivan, 2007] try to find them efficiently.

Given a symbol ***a*** from ∑ and database *D*, the support of pattern *p* in the ***a***-projection database *Da* of *D* is equal to the support of pattern ***a***.*p* in the original database *D*.

$$Sup_D(\boldsymbol{a} \cdot p) = Sup_{Da}(p) \tag{3.6}$$

And support of empty pattern is $|\boldsymbol{a}|$. That is,

$$Sup_D(\boldsymbol{a}) = |Da| \tag{3.7}$$

Let $F(D, \eta)$ be the set of frequent patterns in *D* with respect to η.

$$F(D,n) = \{p | Sup_D(p) \geq \eta\} \tag{3.8}$$

The set of frequent pattern is empty if $|D|$ is less than the absolute threshold. Otherwise it is the union of all sequences that are prefixed by ***a*** for each ***a*** ∈ E having support ≥η

That is, if $|D| < \eta$, then *F(D,n)* is empty. If $|D| > \eta$, then *F(D,n)* contains at least ϵ, the null pattern.

Let $F'(\boldsymbol{a}.D, \eta)$ be the set of non-empty frequent patterns in *D* that start with symbol ***a***, then $F'(\boldsymbol{a}.D, \eta) = F(Da, \eta)$. Thus

$$\mathcal{F}(\mathcal{D}.n) = \phi \qquad if\ |D| < \eta \qquad \text{and}$$

$$\mathcal{F}(\mathcal{D}, n) = \{\varepsilon\} \cup \cup_{a\epsilon\Sigma}\, \boldsymbol{a}\mathcal{F}(\mathcal{D}a.n) \quad if\ |D| \geq \eta \qquad (3.9)$$

Equations (3.7) and (3.9) are the base of the FLWAP-Mining algorithm [Tang, Turkia and Gallivan, 2007].

Mining with FLWAP-Tree

A node is a first-Occurrence, if none of its ancestors has the same label. The count of first-Occurrence of a node with label ***a*** is the number of sequences in *D* that share the common ***a***-prefix represented by the path from the root node to this. The sum of the counts of all the first-Occurrences of a symbol, $\boldsymbol{a} \in \sum$ is the number of sequences in *D* that contain at least one occurrence of ***a***, i.e. the support of *a* in D, $Sup_D(a)$.

To build FLWAP-Tree, sequences are entered into the tree as in base WAP-Tree. Then the first-Occurrences of each symbol are linked to form the First-Occurrence Linked WAP-Tree (FLWAP-Tree). The first occurrences can be found by the pre-order traversal of a portion of the base WAP-Tree. Once the FLWAP-Tree is constructed, it is mined to generate all patterns using the FLWAP-mine algorithm. The algorithm works as follows. For each frequent event a, follow the first-Occurrence link to find its first-Occurrences. If the sum of the counts of first-Occurrences is greater than absolute threshold, ***a*** is concatenated to the previous pattern and added to the set of patterns. Now the set of the sub-trees rooted at the children of the first-Occurrences is found, FLWAP-Tree for D*a* is built and recursively mined. The main algorithm of FLWAP-Mine is given in Figure 3.8.

Function FLWAP-Mine(database D, int η)

if $|D| < \eta$ then

    return

else

    $F = \{\boldsymbol{\epsilon}\}$;

    for each $\boldsymbol{a} \in \sum$ do

```
        if (Sup_D(a) ≥ η) then
            F′ =  Mine(Da, η);
            F =  F ∪ aF′ ;
        endif
    endfor
    return F;
endif
```

Figure 3.8: FLWAP-Mine Algorithm for Mining Web Access Patterns

Difference of FLWAP-Mine algorithm from PLWAP-Mine lies in finding first-Occurrences and in building intermediate tree for the projection database. To find the first-Occurrences of a symbol, PLWAP attach a position code to each node and link all the nodes of the same label in a pre-order traversal of the tree. In FLWAP-Mine, first occurrences are found by traversing a portion in pre-order and the detailed algorithms are available in [Tang, Turkia and Gallivan, 2007].

3.4.6 FOF-Mine

In [Pearson and Tang, 2008], the authors proposed a modification to FLWAP mining method in [Tang, Turkia and Gallivan, 2007]. Instead of the concept of linked trees, FOF-Mine uses Forest of First-Occurrence sub-trees (FOF) as the basic data structure for representing projection database. Given a symbol ***a***, each sub tree rooted at a first-Occurrence of ***a***, is the first-Occurrence sub-tree of ***a***. A list of pointers to the first-Occurrences of ***a*** in the aggregate tree is the forest of first-Occurrence sub-trees of a symbol. FOF structure of the projection database D*a* is provided in Figure 3.9.

In First Occurrence Forest-Mine (FOF-Mine), the aggregate tree is extended to make the root node represent the empty symbol ***ε***. The count of the root node is the total number of sequences in the database.

The count of first-occurrence of a node with label ***a***, is the number of sequences that share the common ***a***-prefix. The sub-trees rooted at the children of first-occurrence of symbol ***a***, represent all the non-empty a-projections of the sequences sharing this common ***a***-prefix. The sum of the counts of the root nodes of the first-occurrence sub-trees of ***a*** gives $Supp_D(\boldsymbol{a})$. As all the nodes already exist in the original aggregate tree of

the database to be mined, the memory cost of the forest of first-occurrences sub-trees is only the list of pointers.

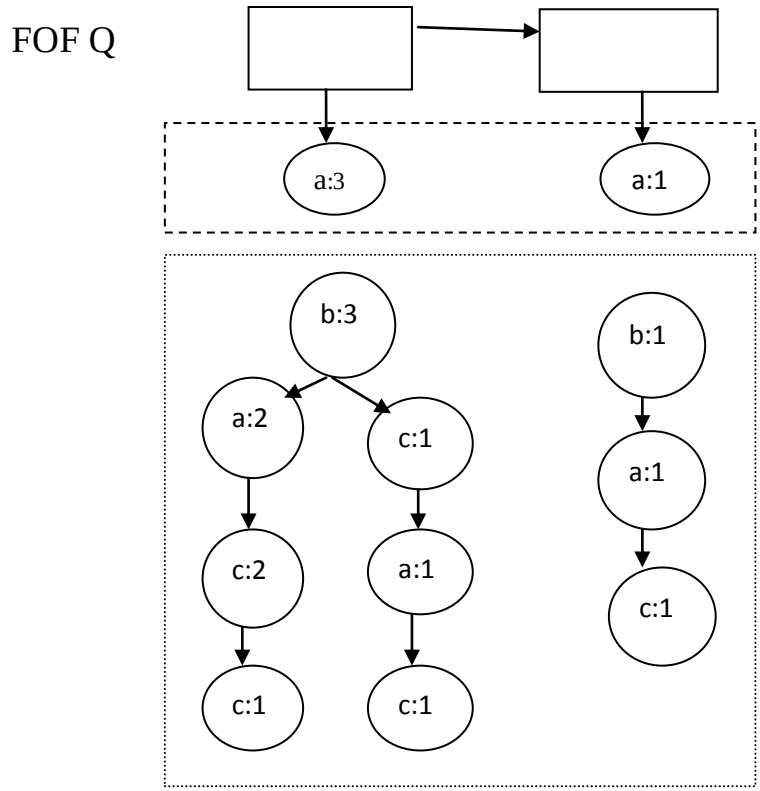

Figure 3.9: FOF for the Projection Database, *Da*

FOF-Mine is the recursive algorithm used for mining FOF structure. Figure 3.10 depicts the algorithm for FOF-Mine. The initial FOF structure based on aggregate tree is passed on to the mining function. The sub-trees rooted at the children of the first occurrence nodes represent the current database to be mined. The FOF structure for the projection database is established, by repeatedly calling the function Find-First-Occurrences. The overview of the algorithm for mining is as follows: In the main function, initial aggregate tree is constructed, starting from root and incorporated into the FOF structure. Then it is passed to the recursive FOF mining algorithm.

The global variable Q of type FOF in Figure 3.9, is used to pass a new FOF structure every time the FOFMine() function is called. Function FOFMine() is the pattern-growth mining function. Argument *T* of type FOF represents the current database to be mined. Part (ii) of the FOF structure represents the subtrees rooted at the children of the nodes of part (i). Part (i) of T represents the first occurrences of the previous symbol in the previous call and it is not used in the current call.

```
---------------------------------------------------------------------------------------- ----------
FOF Q;
function main ()
{
    FOF initialFOF ← newFOF();
    node root;
    Construct initial aggregate tree starting from root;
    initialFOF.append(root);
    F ← FOFMine(ϙ, initialFOF, η);
}
function FOFMine(pattern q, FOF T, int η)
{
     F ← ∅;
    for each symbol a in ∑ do
    {
        Q ← newFOF();
        for each root node N of the subtrees in T do
            FindFirstOccurrences(a,N.firstChild);
        endfor
        if (the sum of the count of the root nodes of all the subtrees in Q) ≥ η) then
             F ← F ∪ {q . a};
            F ← F ∪ FOFMine(q. a, Q, η);
        endif
        delete Q;
    }//endfor
return F;
}
procedure FindFirstOccurrences(symbol a, node N)
{
    if (N.label = a) then
        Q.append( N);
    else
        if (N.firstChild = NULL) then
```

```
            Find First Occurrences(a,N.firstChild);
        endif
    endif
    if (N.nextSibling= NULL) then
    Find First Occurrences (a , N. nextSibling);
    endif
}
```

Figure 3.10: FOF Algorithm

In FOF-Mine, for each frequent event **a**, the projection database of the current database is build. If the sum of the counts of the root nodes of all the sub-trees is greater than the absolute threshold, ***a*** is appended to the previous pattern and added to the set of pattern. Now, the new database is recursively mined to generate all pattern starting with the current event.

3.4.7 Comparison of the Methods

WAP-Mine introduced in [Pei et al., 2000] is quite different from the apriori methods like Apriori, Apriori All, Apriori Some [Agrawal and Srikant, 1995], GSP (Generalized Sequential Pattern mining) [Srikant and Agrawal, 1996], PSP (Prefix Tree for Sequential Patterns) [Chen et al., 2007] that are based on generate and test methods. WAP-Mine uses a very compact WAP-Tree structure to store access sequences. In a WAP-Tree, common prefixes are shared by the branches. Support counting also is easier. As the size of the tree is much smaller, mining is easier. Moreover, the conditional searching improves the efficiency. Experimental studies proved that WAP-Mine performs better than GSP. Both GSP and WAP-Mine exhibit linear scalability, but WAP-Mine outperforms GSP [Srikant and Agrawal, 1996]. Though WAP-Mine avoids explosive generation of candidate sequences, recursive reconstruction of intermediate WAP-Trees during mining in order to compute frequent prefix subsequences of every suffix sequences is very time consuming.

PLWAP-Mine [Lu and Ezeife, 2003] modified the base WAP-Tree to Pre Order Linked WAP-Tree. In WAP-Tree similar nodes are connected in the order of arrival, but in PLWAP-Tree nodes are linked in a pre-order fashion. PLWAP algorithm avoids the recursive reconstruction of intermediate WAP-Trees by identifying the suffix trees or

forest of any frequent pattern prefix under consideration by comparing the binary codes of nodes. Experimental evaluations done by the authors of [Lu and Ezeife, 2005] revealed that PLWAP outperforms both GSP and WAP-Mine when the number of frequent patterns increases and the minimum support threshold is low. Performance of PLWAP degrades when the length of sequence is more than 20 because of the increase in the size of position codes as the depth of tree increases [Lu and Ezeife, 2005].

Though the memory requirement is reduced in PLWAP tree method, use of the original PLWAP-Tree for the entire mining forces it to go through the nodes that are not in the projection databases for finding out first-Occurrences. The FLWAP-Tree algorithm [Tang, Turkia and Gallivan, 2007] links the first occurrences of each symbol. Since all the non-empty sequences in the *a*-projection database *Da* of database *D* are included in the sub-trees rooted at the children of the first occurrences of a symbol ***a***, the FLWAP-Tree for *Da* can be built from these sub-trees. Building new trees for projection databases use more memory. However, reduction in the sizes of projection databases reduces the processing complexity. That is, FLWAP-Tree mining incurs a trade-off between memory and high performance. During experiments FLWAP-Tree mining outperforms the PLWAP-Tree mining consistently as the average length of sequence increases. The speed up of FLWAP also increases linearly as the average length of sequence increases [Pearson and Tang, 2008]

Both the PLWAP-tree and FLWAP-Tree algorithms are based on the concept of the linked tree. FOF algorithm [Pearson and Tang, 2008] outperforms both PLWAP and FLWAP. The FLWAP-Tree algorithm rebuilds every projection database, and thus, uses a lot of memory. Due to the increase in the size of the tree nodes in PLWAP-Tree, the FOF algorithm outperforms PLWAP-Tree algorithm in memory usage, even though PLWAP-Mine does not create additional projection databases [Pearson and Tang, 2008].

CS-Mine is also based on WAP-Tree, but it uses WAP-Tree only for generating conditional sequence base. Experimental results have shown that the CS-mine algorithm performs much more efficient than the WAP-mine algorithm, especially when the support threshold becomes small and the number of web access sequences gets larger [Zhou, Hui and Fong, 2004].

3.5 Summary

Sequential pattern mining finds the relationships between occurrences of sequential events. Sequential patterns indicate the correlation between transactions while association rule represents intra transaction relationships.

In this chapter, a detailed discussion of various approaches to sequential pattern mining is presented. Basic ideas of Web Access Sequence, Web Access Pattern and Web Access Pattern Mining and how sequential pattern mining is used for mining web access pattern are illustrated. It also provides a detailed study of the WAP-Tree structure and WAP-Mine algorithm. Four prominent improvements over WAP-Mine algorithm Viz. PLWAP-Mine, CS-Mine, FLWAP-Mine and FOF-Mine are also discussed. Finally a comparison of the aforesaid algorithms based on the reviewed literature is carried out. FOF-Mine is found to be the best approach for Web Access Pattern Mining.