Open access • Journal Article • DOI:10.1007/S10618-005-0248-3

Mining Web Log Sequential Patterns with Position Coded Pre-Order Linked WAP-Tree — Source link

Christie I. Ezeife, Yi Lu

Institutions: University of Windsor

Published on: 01 Jan 2005 - Data Mining and Knowledge Discovery (Kluwer Academic Publishers)

Topics: Web mining, Trie, Web page, Sequence database and Tree (data structure)

Data Mining and Knowledge Discovery, 10, 5–38, 2005

Mining Web Log Sequential Patterns with Position Coded Pre-Order Linked WAP-Tree*

C.I. EZEIFE cezeife@uwindsor.ca;(http://www.cs.uwindsor.ca/~cezeife)
YI LU
School of Computer Science, University of Windsor, Windsor, Ontario, Canada, N9B 3P4

Editor: Fayyad

Received July 2002; Revised June 2004

Abstract. Sequential mining is the process of applying data mining techniques to a sequential database for the purposes of discovering the correlation relationships that exist among an ordered list of events. An important application of sequential mining techniques is web usage mining, for mining web log accesses, where the sequences of web page accesses made by different web users over a period of time, through a server, are recorded. Web access pattern tree (WAP-tree) mining is a sequential pattern mining technique for web log access sequences, which first stores the original web access sequence database on a prefix tree, similar to the frequent pattern tree (FP-tree) for storing non-sequential data. WAP-tree algorithm then, mines the frequent sequences from the WAP-tree by recursively re-constructing intermediate trees, starting with suffix sequences and ending with prefix sequences.

This paper proposes a more efficient approach for using the WAP-tree to mine frequent sequences, which totally eliminates the need to engage in numerous re-construction of intermediate WAP-trees during mining. The proposed algorithm builds the frequent header node links of the original WAP-tree in a pre-order fashion and uses the position code of each node to identify the ancestor/descendant relationships between nodes of the tree. It then, finds each frequent sequential pattern, through progressive prefix sequence search, starting with its first prefix subsequence event. Experiments show huge performance gain over the WAP-tree technique.

Keywords: sequential patterns, Web usage mining, WAP-tree mining, pre-order linkage, position codes, apriori techniques

1. Introduction

Association rule mining is a data mining technique which discovers strong associations or correlation relationships among data. Given a set of transactions (similar to database records in this context), where each transaction consists of items (or attributes), an association rule is an implication of the form $X \rightarrow Y$, where X and Y are sets of items and $X \cap Y = \emptyset$. The support of this rule is defined as the percentage of transactions that contain the set $X \cup Y$, while its confidence is the percentage of these "X" transactions that also contain items in "Y". In association rule mining, all items with support higher than a specified

*This research was supported by the Natural Science and Engineering Research Council (NSERC) of Canada under an operating grant (OGP-0194134) and a University of Windsor grant.

Table 1. The example database transaction table with items.

TID	Items bought
100	f, a, c, d, g, i, m, p
200	a, b, c, f, l, m, o
300	b, f, h, j, o
400	b, c, k, s, p
500	a, f, c, e, l, p, m, n

minimum support are called large or frequent itemsets. An itemset X is called an i-itemset if it contains i items. Agrawal and Srikant (1994) presents the concept of association rule mining and an example of a simple rule is "80% of customers who purchase milk and bread also buy eggs." Table 1 shows an example database transaction table, which can be mined. This table has the set of items $I = \{a, b, c, d, e, f, g, h, i, j, k, l, m, n, o, p\}$, where these items could stand for retail store items like bread, butter, cheese, egg, milk, sugar or web pages if the domain of interest is web mining. Using this database, which has only five transactions for simplicity, the support of the itemset, "a, f, c" is 3 or 3/5 (60%) since the itemset "a, f, c" is present in only three of the five transactions (100, 200 and 500). Thus, if the minimum support given by the user for deciding frequent itemsets is 60% or lower, then, the itemset "a, f, c" is a frequent itemset. However, if the minimum support is 70%, then, itemset "a, f, c" is not a frequent itemset since it has a support that is lower than the minimum support threshold. The confidence of a rule, $a, f \rightarrow c$, that can be generated from the frequent itemset, "a, f, c" is 100% because all three transactions that contain the antecedent items, "a, f" also contain the consequent item "c".

Since discovering all such rules may help market baskets or cross-sales analysis, decision making, and business management, algorithms presented in this research area include (Agrawal and Srikant, 1994; Park et al., 1997; Mannila et al., 1995; Han and Kamber, 2000). These algorithms mainly focus on how to efficiently generate frequent patterns from a non-sequential (non-ordered) lists of items, and how to discover the most interesting rules from the generated frequent patterns.

While traditional association rule mining finds intra-transaction patterns, sequential pattern mining finds inter-transaction patterns, to detect the presence of a set of items in a time-ordered sequence of transactions. In basic association rule mining, the items occurring in one transaction have no order, but in sequential pattern mining, an order exists between the items (events) and an item may re-occur in the same sequence. Example of a sequential pattern is: in a video rental store, 80% of customers typically rent "star wars", then "Empire strikes back", and then "Return of Jedi". The measures of support and confidence, used in association rule mining for deciding frequent itemsets, are still used in sequential pattern mining to determine frequent sequences and strong rules that can be generated from them.

Web usage mining is used for automatic discovery of user access patterns from web servers. Interesting user access patterns can be extracted from web access logs, recorded on the web server. Analysis of these access data can provide useful information for server performance enhancements, restructuring a web site, and direct marketing in e-commerce (Etzioni, 1996). Madria et al. (1999), Borges and Leven (1999), and Srivastava et al. (2000), all propose to categorize web mining into three areas based on which part of the web information to mine: web content mining, web structure mining and web usage mining. Web content mining requests the discovery of useful information from the real data on web pages, such as the data that web page was designed to convey to the users. This usually consists of several types of data such as textual, image, audio, video, metadata, as well as hyperlinks. Web content data include free texts, semi-structured data like HTML documents, and more structured data like data in tables, as well as database generated HTML pages and XML pages. Web structure mining discovers the model underlying the link structures of the web. The model is based on the topology of the hyperlinks with or without the description of the links. This model can be used to categorize web pages and is useful for generating information such as similarity and relationships between different web sites. Web structure mining could be used to discover authority sites which are sites organized for particular subjects and have many links to other related web sites based on the subject. Moreover, it can be used to find the hubs which point to many authorities regarding one particular subject. Web usage mining makes sense of data generated by observing web surf sessions or behaviors. Web usage mining, finds the relationship among different users' accesses. For example, it may be found that: 90% of clients who accessed the page with URL /products/product.html, also accessed the page /contact/contact.html. This information reveals that these two pages are closely related and can be organized together to provide users with an easier browsing route. All users' behaviors on each web server can be extracted from the web log. An Example of a line of data in a web log is given below in the format:

```
host/ip user [date:time] “request url” status bytes
137.207.76.120–[30/Aug/2001:12:03:24-0500] “GET/jdk1.3/docs/relnotes/deprecatedlist.
  html HTTP/1.0” 200 2781
```

This recorded information represents from left to right, the host ip address of the computer accessing the web page (137.207.76.120), the user identification number (–) (this dash stands for anonymous), the time of access (12:03:24 p.m. on Aug 30, 2001 at a location 5 hours behind Greenwich Mean Time (GMT)), request (GET /jdk1.3/docs/relnotes/deprecatedlist. html)(other types of request are POST and HEAD for http header information), the unified reference locator (url) of the web page being accessed (HTTP/1.0), status of the request (which can be either 200 series for success, 300 series for re-direct, 400 series for failure and 500 series for server error), the number of bytes of data being requested (2781). Although most projects extract the information from the common log, the common log can not reflect every user's visit behavior one hundred percent. The cached page views can not be recorded in server log, and the information passed through the POST method will not be available in server log. Thus, in order to gain proper web usage data source, some extra techniques, such as packet sniffer and cookies may be needed. In most cases, researchers are assuming that user web visit information is completely recorded in the web server log, which is pre-processed to obtain the transaction database to be mined for sequences.

Basic association rule discovery using the Apriori algorithm (Agrawal and Srikant, 1994) can relate pages that are most often referenced together in a single user session. Thus, the presence or absence of such rules can help web designers to restructure their web site. The association rule method used in web usage mining is mostly the same process as that used in traditional databases or data warehouses. With sequential pattern mining, web marketers can predict future visit patterns, which will be helpful in placing advertisements aimed at certain user groups. For example, starting from Yahoo's home page, users can locate information regarding universities in Canada by following either *Home* → *Education* → *Higher Education* → *Colleges and Universities* → *By Region* → *Countries* → *Canada* or *Home* → *Regional* → *Countries* → *Canada* → *Education* → *Higher Education*. Thus, universities that want to attract prospective students can place their advertisements on any of the pages along the path. Sequential pattern technique is useful for finding frequent web access patterns and has been applied in the following recent work (Spiliopoulou, 1999; Berendt and Spiliopoulou, 2000; Nanopoulos and Manolopoulos, 2000).

A web log is a sequence of events (or items), each, with a pair of attribute values—user identifier, access information. The information can be any combination of values in the original web log format provided earlier on. For example, the access information here, stands for access content. For simplicity, the web log access content is represented as items {a, b, c, d, e, f}. For example, a fragment of a web log is shown below in the format, <User ID, Access content>):

<100, *a*><100, *b*><200, *e*><200, *a*><300, *b*><200, *e*><100, *d*><200, *b*><400, *a*><400, *f*> <100, *a*><400, *b*><300, *a*><100, *c*><200, *c*><400, *a*><200, *a*> <300, *b*><200, *c*> <300, *f*><400, *c*><400, *f*><400, *c*><300, *a*><300, *e*> <300, *c*>.

These web log events are pre-processed, to group them into sets of access sequences for each user identifier and to create web access sequences in the form of a transaction database. The web log sequences in a transaction database, obtained after pre-processing the web log has each tuple consisting of a transaction ID and the sequence of this transaction's web accesses. Thus, for example, user ID 100, from the given web log above, has accessed contents, a then b, d, a, and c. The transaction web access sequences from these given web log data is shown as Table 2. The problem of mining sequential patterns from web logs are now based on the database of Table 2. Given a set of events E, the access sequence S can be represented as $e_1e_2 \ldots e_n$, where $e_i \in E$ $(1 \le i \le n)$. That means the access sequence

Table 2. Sample Web access sequence database.

TID	Web access sequences
100	*abdac*
200	*eaebcac*
300	*babfaec*
400	*babfaec*

is composed of a series of events, which are members of event set E. It is not necessary that $e_i \neq e_j$ for $(i \neq j)$ in an access sequence S, meaning that repetition of events are allowed in a sequence. For example, in Table 2, $E = \{a, b, c, d, e, f\}$ and an S is abdac. The length of an access sequence, $|S|$ is the number of events in the sequence and an access sequence with length m is called an m-sequence. For example, abdac is a 5-sequence. Web Access Sequence Database (*WASD*) is the set $S_1, S_2, \ldots, S_m$, where S_i, $(1 \leq j \leq m)$ are access sequences. Also, $|WASD|$ is used to indicate the number of access sequences in the database. For example, the web database above is a WASD with 4 access sequences *abdac*, *eaebcac*, *babfaec* and *babfaec* in the database. Access sequence $S' = e'_1 e'_2 \ldots e'_l$ is called a subsequence of an access sequence, $S = e_1 e_2 \ldots e_n$, and S is a super-sequence of S', denoted as $S' \subseteq S$, if and only if for every event e'_j in S', there is an equal event e_k in S, while the order that events occurred in S should follow the order of events in S'. For example, with $S' = ab$, $S =$ babcd, we can say that S' is a subsequence of S. We can also say that ac is a subsequence of S, although there is b occurring between a and c in S. A frequent pattern is an access sequence to be discovered during the mining process and it should have a support that is higher than minimum support. In access sequence $S = e_1 e_2 \ldots e_k e_{k+1} \ldots e_n$, if subsequence $S_{\text{suffix}} = e_{k+1} \ldots e_n$ is a super sequence of pattern $P = e'_1 e'_2 \ldots e'_l$, where $e_{k+1} = e'_1$, $S_{\text{prefix}} = e_1 e_2 \ldots e_k$, is called the prefix of S with respect to pattern P, while S_{suffix} is the suffix sequence of S_{prefix}. For example, in sequence *eaebcac*, *eae* is a prefix of *bcac*, while the bcac is a suffix of eae. The support of pattern S in WASD is defined as the number of sequences S_i, which contain the subsequence S, divided by number of transactions in the database WASD. Although events can be repeated in an access sequence, a pattern can get at most one support count contribution from one access sequence. For example, from Table 2, fc is a pattern, which gets 50% support from user ids 300 and 400, fc appears twice in user id 400's sequence, but only one can contribute to the support count of fc. Similar to association rule mining, the minimum support for sequential pattern mining is the percentage value between 0 and 1 that is set by the user to identify the frequent sequence. The problem of web usage mining is that of finding all patterns which have supports greater than λ, given web access sequence database WASD and a minimum support threshold λ.

Techniques for mining sequential patterns from web logs fall into Apriori or non-Apriori. The Apriori-like algorithms generate substantially huge sets of candidate patterns, especially when the sequential pattern is long (Agrawal and Srikant, 1995; Srikant and Agrawal, 1996; Masseglia et al., 1999; Nanopoulos and Manolopoulos, 2000, 2001). WAP-tree mining (Pei et al., 2000) is a non-Apriori method which stores the web access patterns in a compact prefix tree, called WAP-tree, and avoids generating long lists of candidate sets to scan support for. However, WAP-tree algorithm has the drawback of recursively reconstructing numerous intermediate WAP-trees during mining in order to compute the frequent prefix subsequences of every suffix subsequence in a frequent sequence. This process is very time-consuming. This paper first proposes a novel technique for assigning a position code to each node of any general tree, derived from a collection of the single binary position codes of all of this node's ancestors in the binary tree equivalent. Using the position code to label each node of the WAP-tree, this paper then, further proposes a technique that builds the WAP-tree head links in a pre-order fashion rather than in the order the nodes arrive as done by the WAP-tree algorithm. With the pre-order linked, position coded WAP-tree, the algorithm proposed in this paper is now able to mine frequent sequences, starting with the prefix sequence without the need to recursively re-construct any intermediate WAP-trees. This approach results in tangible performance gain.

1.1. Related work

Sequential mining was proposed (Agrawal and Srikant, 1995), using the main idea of association rule mining presented in Apriori algorithm of Agrawal and Srikant (1994). Later work on mining sequential patterns in web log include the GSP (Srikant and Agrawal, 1996), the PSP (Masseglia et al., 1999), the G sequence (Spiliopoulou, 1999) and the graph traversal (Nanopoulos and Manolopoulos, 2001) algorithms. Agrawal and Srikant proposed three algorithms (Apriori, AprioriAll, AprioriSome) to handle sequential mining problem (Agrawal and Srikant, 1995). Following this, the GSP (Generalized Sequential Patterns) (Srikant and Agrawal, 1996) algorithm, which is 20 times faster than the Apriori algorithm in Agrawal and Srikant (1995) was proposed. The GSP Algorithm makes multiple passes over data. The first pass determines the frequent 1-item patterns (L_1). Each subsequent pass starts with a seed set: the frequent sequences found in the previous pass (L_{k-1}). The seed set is used to generate new potentially frequent sequences, called candidate sequences (C_k). Each candidate sequence has one more item than a seed sequence. In order to obtain k-sequence candidate C_k, the frequent sequence L_{k-1} joins with itself Apriori-gen way. This requires that every sequence s in L_{k-1} joins with other sequences s' in L_{k-1} if the last *k-2* elements of s are the same as the first *k-2* elements of s'. For example, if frequent 3-sequence set L_3 has 6 sequences as follows: $\{((1, 2)\,(3)), ((1, 2)\,(4)), ((1)\,(3, 4)), ((1, 3)\,(5)), ((2)\,(3, 4)), ((2)\,(3)\,(5))\}$. In order to obtain frequent 4-sequences, every frequent 3-sequence should join with the other 3-sequences that have the same first two elements as its last two elements. Sequence $s = ((1, 2)\,(3))$ can join with $s' = ((2)\,(3, 4))$ to generate a candidate 4-sequence because the last 2 elements of s, $(2)\,(3)$, are the same as the first 2 elements of s'. Then, element (4) can be added to the sequence $((1, 2)\,(3))$. Since element (4) is part of the last element $(3, 4)$ of s', $((2)\,(3, 4))$, the new sequence is $((1, 2)\,(3, 4))$. Also, $((1, 2)\,(3))$ can join with $((2)\,(3)\,(5))$ to form $((1, 2)\,(3)\,(5))$. The remaining sequences can not join with any sequence in L_3. Following the join phase is the pruning phase, when the candidate sequences that have any of their contiguous $(k - 1)$-subsequences having a support count less than the minimum support, are dropped. The supports for the remaining candidate sequences are found next to determine which of the candidate sequences are actually frequent (L_k). These frequent candidates become the seed for the next pass. The algorithm terminates when there are no frequent sequences at the end of a pass, or when there are no candidate sequences generated. The GSP algorithm uses a hash tree to reduce the number of candidates that are checked for support in the database.

The PSP (Prefix Tree For Sequential Patterns) (Masseglia et al., 1999) approach is much similar to the GSP algorithm (Srikant and Agrawal, 1996). At each step k, the database is browsed for counting the support of current candidates. Then, the frequent sequence set, L_k

is built. The only difference between the PSP algorithm and the GSP is that it introduces the prefix-tree to handle the procedure. Any branch, from the root to a leaf stands for a candidate sequence, and a terminal node provides the support of the sequence from the root to the considered leaf inclusive.

The main idea of Graph Traversal mining which is proposed by Nanopoulos and Manolopoulos (2000, 2001), is using a simple unweighted graph to reflect the relationship between the pages of web sites. Then, a graph traversal algorithm similar to Apriori algorithm, is used to traverse the graph in order to compute the k-candidate set from the $(k-1)$-candidate sequences without performing the apriori-gen join. From the graph, if a candidate node is large, the adjacency list of the node is retrieved. The database still has to be scanned several times to compute the support of each candidate sequence although the number of computed candidate sequences is drastically reduced from that of the GSP algorithm. Other tree based approaches include (Spiliopoulou, 1999) called G sequence mining. This algorithm uses wildcards, templates and construction of Aggregate tree used for mining.

The FP-tree structure (Han et al., 2004) first reorders and stores the frequent non-sequential database transaction items on a prefix tree, in descending order of their supports such that database transactions share common frequent prefix paths on the tree. Then, mining the tree is accomplished by recursive construction of conditional pattern bases for each frequent 1-item (in ordered list called f-list), starting with the lowest in the tree. Conditional FP-tree is constructed for each frequent conditional pattern having more than one path, while maximal mined frequent patterns consist of a concatenation of items on each single path with their suffix f-list item. FreeSpan (Han et al., 2000) like the FP-tree method, lists the f-list in descending order of support, but it is developed for sequential pattern mining. FreeSpan mines frequent sequential patterns starting with each of its f-list items α, through recursive construction of projected databases of this f-list item α. A projected database of an ordered f-list item α from the database D, consists of all sequences in D containing this f-list item α but removing all items after α in the ordered f-list. PrefixSpan (Pei et al., 2001) is a pattern-growth method like FreeSpan, which reduces the search space for extending already discovered prefix pattern p by projecting a portion of the original database that contains all necessary data for mining sequential patterns grown from p. While FreeSpan supports frequent pattern guided projection, PrefixSpan supports prefix guided projection. Thus, projected database for each f-list prefix pattern α consists of all sequences in the original database D, containing the pattern α and only the subsequences prefixed with the first occurrence of α are included. Although PrefixSpan projects smaller sized databases than FreeSpan, they both still incur non-trivial costs for constructing and storing these projected databases for every sequential pattern in the worst case. Optimization techniques include (1) bi-level projecting for reducing the number and sizes of projected databases, and (2) Pseudo-projection for projecting memory-only databases, where each projection consists of the pointer to the sequence and offset of the postfix to the sequence.

Pei et al. (2000) proposed an algorithm using WAP-tree, which stands for web access pattern tree. This approach is quite different from the Apriori-like algorithms. The main steps involved in this technique are summarized next. The WAP-tree stores the web log data

Table 3. Sample Web access sequence database for WAP-tree.

TID	Web access sequence	Frequent subsequence
100	*abdac*	*abac*
200	*eaebcac*	*abcac*
300	*babfaec*	*babac*
400	*afbacfc*	*abacc*

in a prefix tree format similar to the frequent pattern tree (Han et al., 2004) (FP-tree) for non-sequential data. The algorithm first scans the web log once to find all frequent individual events. Secondly, it scans the web log again to construct a WAP-tree over the set of frequent individual events of each transaction. Thirdly, it finds the conditional suffix patterns. In the fourth step, it constructs the intermediate conditional WAP-tree using the pattern found in previous step. Finally, it goes back to repeat Steps 3 and 4 until the constructed conditional WAP-tree has only one branch or is empty.

Thus, with the WAP-tree algorithm, finding all frequent events in the web log entails constructing the WAP-tree and mining the access patterns from the WAP tree. The web log access sequence database in Table 3 is used to show how to construct the WAP-tree and do WAP-tree mining. Suppose the minimum support threshold is set at 75%, which means an access sequence, s should have a count of 3 out of 4 records in our example, to be considered frequent. Constructing WAP-tree, entails first scanning database once, to obtain events that are frequent. When constructing the WAP-tree, the non-frequent part of every sequence is discarded. Only the frequent sub-sequences are used as input. For example, in Table 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 (like d, e, f) are deleted from each transaction sequence to obtain the frequent subsequence shown in column three of Table 3.

With the frequent sequence in each transaction, the WAP-tree algorithm first stores the frequent items as header nodes so that these header nodes will be used to link all nodes of their type in the WAP-tree in the order the nodes are inserted. When constructing the WAP-tree, a virtual root (Root) is first inserted. Then, each frequent sequence in the transaction is used to construct a branch from the Root to a leaf node of the tree. Each event in a sequence is inserted as a node with count 1 from Root if that node type does not yet exist, but the count of the node is increased by 1 if the node type already exists. Also, the head link for the inserted event is connected (in broken lines) to the newly inserted node from the last node of its type that was inserted or from the header node of its type if it is the very first node of that event type inserted. For example, as shown in figure 1(a), to insert the first frequent sequence *abac* of transaction ID 100 of the example database, since there is no node labeled a yet, which is a direct child of the Root, a left child of Root is created, with label a and count 1. Then, the header link node for frequent event a is connected (in broken lines) to this inserted a node from the *a header node*. The next event b is inserted as the left child of node a with a count of 1 and linked to header node b, the third event a is inserted as the left child of the node b having a count of 1, and the a link is connected to

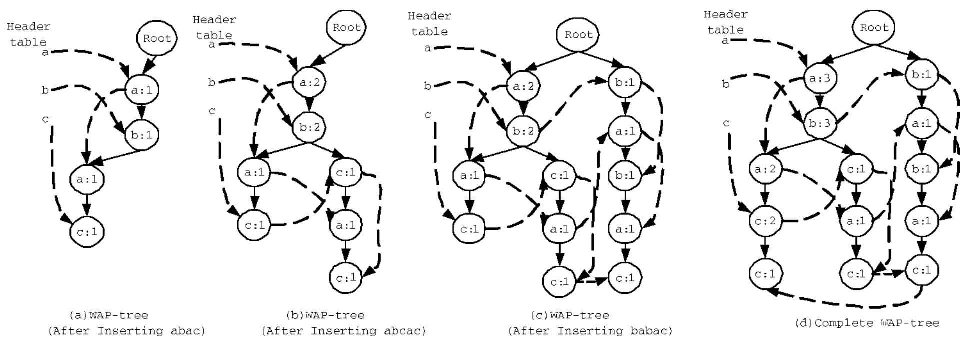
Header table
a
b
c
Root
(a)WAP-tree (After Inserting abac)
(b)WAP-tree (After Inserting abcac)
(c)WAP-tree (After Inserting babac)
(d)Complete WAP-tree

Figure 1. Construction of the original WAP tree.

this node from the last inserted *a*. The fourth and last event of this sequence is *c* and it is inserted as the left child of the second *a* on this branch with a count of 1 and a connection to *c* header node. Secondly, insert the sequence *abcac* of the next transaction with ID 200, starting from the virtual Root (figure 1(b)). Since the root has a child labeled *a*, the node *a*'s count is increased by 1 to obtain (*a*:2) now. Similarly, (*b*:2) is also in the tree. The next event, *c*, does not match the next existing node *a*, and new node *c*:1 is created and inserted as another child of *b* node. The third sequence *babac* of ID 300 and the fourth sequence *abacc* are inserted next to obtain figure 1(c) and (d) respectively.

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 our example, starting from frequent event *c* as the following discussion shows. From the WAP-tree of figure 1(c), it first computes prefix sequence of the base *c* or 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. The first sequence in the list above, *aba* represents the path to the first *c* node in the WAP tree. When a conditional sequence in a branch of a WAP-tree, has a prefix subsequence that is also a conditional sequence of a node of the same base, the count of this new subsequence is subtracted because it has contributed before. Thus, the conditional sequence list above has two sequences *ab* and *aba* with counts of −1. This is because, when the subsequence, *abca* is added to the list, its subsequence *ab* was already in the list, from the first *c* on the same branch. Thus, the count of *ab* with −1 has to be added to prevent it from contributing twice. To qualify as a conditional frequent event, one event must have a count of 3. Therefore, after counting the events in sequences above, the conditional frequent events are *a*(4) and *b*(4) and *c* with a count of 2, which is less than the minimum support, is discarded. After discarding the non-frequent part *c* in the above sequences, the conditional sequences based on *c* are listed below:

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

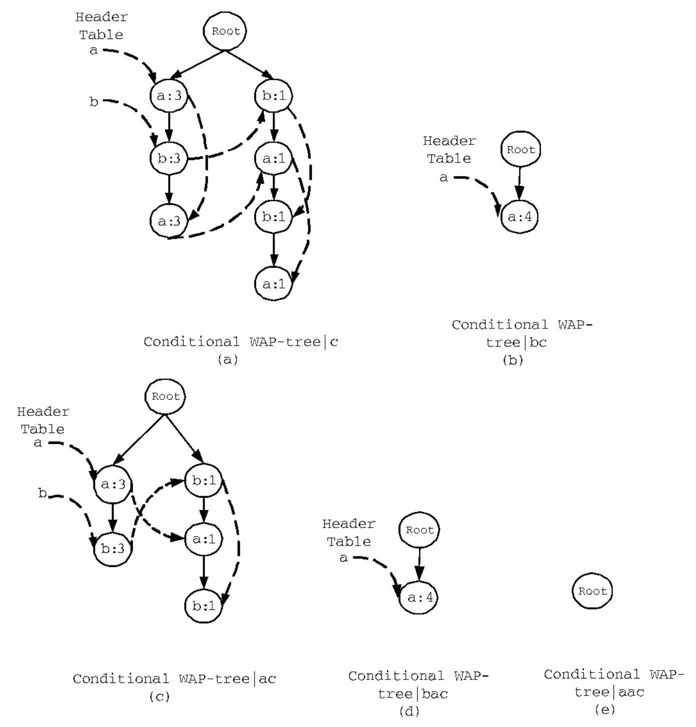

Figure 2. Reconstruction of WAP trees for mining conditional pattern base *c*.

Using these conditional sequences, a conditional WAP tree, WAP-tree|c, is built using the same method as shown in figure 1. The new conditional WAP-tree is shown in figure 2(a). Recursively, based on the WAP-tree in figure 2(a), the next conditional sequence base for the next suffix subsequence, *bc* is found as $a(3)$, $\emptyset$, $ba(1)$. With *a* as the only frequent pattern in this base, the frequent sequence base of *bc* used to construct the next WAP tree shown in figure 2(b) is $a(4)$. 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 $|c$, $|ac$. Thus, the conditional sequence base for suffix *ac* is computed from figure 2(a) as $\emptyset$, ab:3; b:1; bab:1, b:-1. This list is used to construct the WAP tree of figure 2(c). The algorithm keeps running, finding the conditional sequence bases of *bac* as a:3; ba:1. From the list, the conditional frequent events of *bac* is only a:4. Then, the conditional WAP-tree|bac is built as shown in figure 2(d). Now back to completing the mining of frequent patterns with suffix *ac*, figure 2(c) is mined for conditional sequence bases for suffix *aac* to find b:1. Since the count of b:1 is less than the minimum support threshold, it is discarded. The empty conditional WAP-tree with the suffix *aac* is shown in figure 2(e).

The conditional search of c is now finished. 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*}.

WAP-tree algorithm scans the original database only twice and avoids the problem of generating explosive candidate sets as in Apriori-like algorithms. Mining efficiency is improved sharply, but the main drawback of WAP-tree mining is that it recursively constructs large numbers of intermediate WAP-trees during mining and this entails storing intermediate patterns, which are still time consuming operations.

1.2. Motivations and contributions

The use of World Wide Web as a means for marketing and selling has increased dramatically in recent years. As e-commerce activities become more important, organizations must spend more time to provide the right level of information to their customers. Web usage mining is the application of established data mining techniques for analyzing web site usage. For an e-commerce company, this means detecting future customers likely to make a large number of purchases, or predicting which online visitors will click on what commercials or banners based on observation of prior visitors who have behaved both positively or negatively to the advertisement banners. The Apriori-like sequential mining algorithms generate huge sets of candidate patterns, especially when the patterns are long. WAP-tree algorithm has the drawback of recursively re-constructing intermediate WAP-trees during mining, which is time-consuming. This paper proposes the Pre-Order WAP tree algorithm, which stores the sequential data in a Pre-order linked WAP tree. Each of this tree's nodes has a binary position code assigned for directly mining the sequential patterns without re-constructing the intermediate WAP trees. This paper also contributes the technique for assigning a binary position code to nodes of any general tree, which, can be used to quickly define the ancestors and descendants of any node. The technique proposed for mining in this paper presents a much better performance than that achieved by the WAP-tree technique.

1.3. Outline of the paper

Section 2 presents the proposed Pre-Order Linked WAP-Tree Mining (PLWAP) algorithm with the Tree Binary Code Assignment (TreBCA) algorithm. Section 3 presents an example sequential mining of a web log database with the PLWAP algorithm. Section 4 discusses experimental performance analysis, while Section 5 presents conclusions and future work.

2. Proposed pre-order linked WAP-tree mining and tree binary code assignment algorithms

Pre-Order Linked WAP-Tree Mining (PLWAP) algorithm is a new sequential pattern mining algorithm for web logs, which is based on WAP-tree (Pei et al., 2000), but avoids recursively re-constructing intermediate WAP-trees during mining of the original WAP tree for frequent patterns. PLWAP algorithm is able to quickly determine the suffix trees or forests of any frequent pattern prefix under consideration by comparing the assigned binary position codes of nodes of the tree. Thus, this section also presents a Tree Binary Code Assignment (TreBCA) algorithm defined for assigning unique binary position codes to nodes of any general tree, by first transforming the tree to its binary tree equivalent and using a rule similar to that used in Huffman coding (Shaffer, 2000), to define a unique code for each node. Section 2.1 presents some definitions necessary for presenting the algorithms, Section 2.2 presents the TreBCA algorithm, while Section 2.3 presents the PLWAP algorithm.

2.1. Common terms and concepts related to PLWAP-tree based mining

A tree is a data structure accessed starting at its root node and each node of a tree is either a leaf or an interior node. A leaf is an item with no child. An interior node has one or more child nodes and is called the parent of its child nodes. All children of the same node are siblings. Like WAP-tree mining, every frequent sequence in the database can be represented on a branch of a tree. Thus, from the root to any node in the tree defines a frequent sequence. For any node labeled e_i in the WAP-tree, all nodes in the path from root of the tree to this node (itself excluded) form a *prefix sequence* of e_i. The count of this node e_i is called the count of the prefix sequence. Any node in the prefix sequence of e_i is an ancestor of e_i. On the other hand, the nodes from e_i (itself excluded) to leaves form the *suffix sequences* of e_i. Any node in the suffix sequence is a descendant of e_i. The suffix sequence of e_i is not unique. Normally, there are several children of e_i in the tree, and each branch from a child to a leaf node will represent a suffix sequence and these suffix branches of e_i are called the suffix trees (forest) of e_i. The suffix trees of e_i are rooted at several nodes that are children of e_i, called the suffix root set of e_i. The suffix root sets are used to virtually represent the suffix forests without the need to physically store each forest. Figure 3(b) presents the suffix trees (forest) of root “A” in figure 3(a). Since there are a number of suffix trees for a node e_i, the relationship between the position of the suffix trees of a node is described using left and right tree of a tree root. For example, in figure 3(a), “B” is left-tree of “C” or “D”, and “D” is the right-tree of “B” or “C”, while node “E” belongs to the left-tree of “C” or “D”.

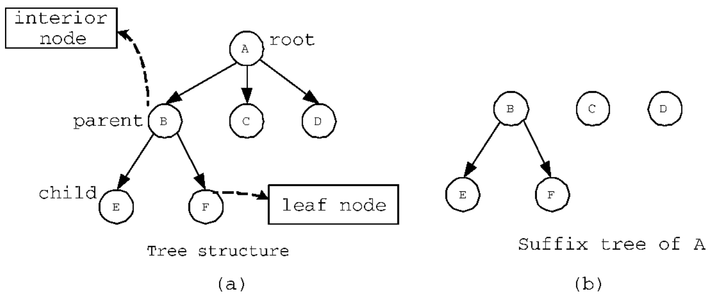

Figure 3. A tree and its suffix forest.

2.1.1. Why suffix trees? The main objective of PLWAP technique is to avoid recursive re-construction of intermediate WAP-trees during mining of the original WAP tree. Unlike the condition search in WAP-tree mining, which is based on finding common suffix sequence first, the PLWAP technique finds the common prefix sequences first. The main idea is to find a frequent pattern by progressively finding its common frequent subsequences starting with the first frequent event in a frequent pattern. For example, if abcd is a frequent pattern to be discovered, the WAP tree technique would start by finding the suffix event *d*, then, it will find the next suffix subsequence *cd*, then *bcd* and finally *abcd* and for each of the these suffix subsequences, an intermediate WAP-tree is reconstructed. The PLWAP tree, on the other hand, would find the the prefix event *a* first, then, using the suffix trees of node *a*, it will find the next prefix subsequence *ab* and continuing with the suffix tree of *b*, it will find the next prefix subsequence *abc* and finally, *abcd*. Thus, the idea of PLWAP is to use 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. For a sequence in the WAP-tree, the events occurring at the beginning of the sequence are presented in the upper part of the WAP-tree, the remaining events of the sequence are only represented by the nodes in the lower part of WAP-tree. So, if we know what part of the WAP-tree can be used to find the next suffix event in the frequent sequence, we need not re-construct the WAP-tree again. The main problem is how to identify which sequences are suffix sequences of the last event. Thus, binary *position codes* are introduced for identifying the position of every node in the WAP tree. Using this method, we can find the suffix tree of a particular event without reconstructing the tree.

2.2. *Tree binary position code assignment algorithm*

Position code is a binary code used to indicate the position of the nodes in the WAP-tree. In data structure (Shaffer, 2000), when implementing a general tree data structure, a tree is usually transformed into its equivalent binary tree, which has a fixed number of child nodes. To convert a given general tree, T, with nodes at n levels, and root at level 0, the leaf nodes at level $(n - 1)$, to a binary tree, the following rule is applied. The root of the binary tree is the leftmost child of the root of the general tree, T. Then, starting from level 1 of the general tree and working down to level $n - 1$ of the tree, for every node: (1) the leftmost child of this node in the general tree is the left child of the node in the binary tree, and (2) the immediate right sibling of this node in the general tree is the right child of this node in the binary tree. For example, given a tree shown as figure 4(a), it can be transformed into its binary tree equivalent shown in figure 4(b), where every node has at most two links, one is its left child, and the other is its sibling.

The position code is assigned to the nodes on the binary tree equivalent of the tree using the Huffman coding idea. Here, the code assignment rule, starts from the leftmost child of the root node of the general tree, which has a binary position code of 1 because this node is the root of the binary tree equivalent of the tree. Thus, given the binary tree equivalent of a tree, with root node having a code of 1, the single temporary position code assignment rule assigns 1 to the left child of each node, and 0 is assigned to the right child of each node. These temporary position codes are used to define the actual binary position code for each

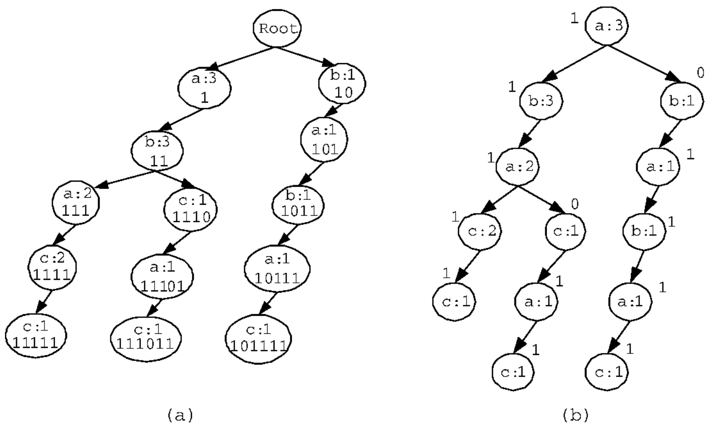

Figure 4. Position code assignment with node position in its binary tree.

node in the original general tree. The position code of a node on the WAP tree is defined as the concatenation of all temporary position codes of its ancestors from the root to the node itself (inclusive) in the transformed binary tree equivalent of the tree. For example, in figure 4(a), the position code of the rightmost leaf node (c:1) is obtained by concatenating all temporary position codes from path (a:3) to (c:1) of the rightmost branch of figure 4(b) to obtain 101111. The transformation to binary tree equivalent is mainly used to come up with a technique for defining and assigning position codes (presented below as Rule 2.1) to nodes of the tree that will accomplish the identification task needed during PLWAP tree mining. It should be stressed that the PLWAP algorithm does not involve physical transformation of a WAP-tree to a binary tree before mining. The tree transformation technique presented in this section is mainly used to define a mechanism for assigning position codes straight to PLWAP tree nodes during their construction. After observing the position code assignment with nodes in figure 4(a), the following properties are defined.

Rule 2.1. *Given a WAP-tree with some nodes, the position code of each node can simply be 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 nth leftmost child, the position code is obtained by appending the binary number for* 2^{n-1} *to the parent's code.*

Property 2.1. *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).

For example, in figure 4(a), (c:1:1110) is an ancestor of (c:1:111011) because the position code of (c:1:1110) is 1110, and after appending 1 at the end of 1110, we get 11101, which is equal to the first 5 (i.e, length of $c + 1$) bits of (c:1:111011). On the other hand, (c:1:1110) is not the ancestor of (c:1:101111), since after appending 1, the code will be 11101 and is not equal to the first 5 bits position code of (c:1:101111). Not only can we use the position code to find the ancestor and descendant relationships between nodes, but we can also find whether one node belongs to the right-tree or left-tree of another node. From figure 4(a), it can be seen that node (c:1:1111) and node (c:1:111011) are two nodes that belong to two subtrees, which are rooted at (a:2:111) and (c:1:1110) respectively. The node (c:1:1111) belongs to a left-tree of (c:1:111011) since the fourth bit of (c:1:111011) is 0, which means the node is extended from the node with position code 1110. The node with position code 1110 is a right sibling of node with 111, which is an ancestor of node (c:1:1111). Thus, (c:1:111011) is a right-tree of (c:1:1111).

2.3. The PLWAP algorithm

The PLWAP algorithm is similar to the WAP-tree algorithm (Pei et al., 2000), introduced in the related work section since both the WAP tree and PLWAP tree are constructed the same way. However, the PLWAP is more efficient since it eliminates recursive re-construction of intermediate WAP-trees in order to mine the original WAP-tree for frequent patterns. The PLWAP algorithm is based on the following Property 2.2 which is derived from the Apriori algorithm (Agrawal and Srikant, 1995). Other applicable properties are also presented.

Property 2.2. *If e is a frequent event within the set of suffixes of a sequential pattern L in the web access sequence database (WASD), then, the sequence Le is a frequent access pattern of WASD. For example, if ac is a frequent pattern, and b is a frequent event within the suffix sets of c, then, acb is a frequent access pattern.*

Property 2.3. *The support count of a node e_i in PLWAP is greater than or equal to the sum of the counts of the suffix subtrees of e_i.*

Using figure 4(a), Property 2.3 is explained with node b:3:11, which has a support count of 3, that is equal to the sum of the support counts of its two suffix tree root set nodes, a:2:111 and c:1:1110. The implication of this property is that in counting the support of a suffix root node e_i of the PLWAP-tree, since the first e_i node from the root, has the higher count than any of its descendants, only the count of this first e_i node contributes to the total count of e_i as presented in Property 2.4.

Property 2.4. *If there is a node in an e_i current suffix subtree, which is also labeled e_i, the support count of the first e_i is the one that contributes to the total support count of e_i, while the count of any other e_i node in this same subtree is ignored.*

From figure 4(a), Property 2.4 allows us to obtain the support count of event a on the left suffix subtree of Root as 3, and that on the right subtree as 1.

Property 2.5. *The support count of a node e_i, in the current e_i suffix trees (also called current conditional PLWAP-trees), ready to be mined is the sum of all first e_i nodes in all suffix trees of e_i, or in the suffix tree of the Root if the first event of a frequent pattern is being mined.*

Applying Property 2.5 to figure 4(a), the support count of node a in the Root suffix trees is 4 (the sum of a:3:1 and a:1:101). The support count of a in subsequent a suffix trees (that is subtrees rooted at b:3:11 and b:1:1011) is 4, from the sum of a:2:1111, a:1:11101 and a:1:10111. The support count of event b in this same a suffix trees rooted at (b:3:11 and b:1:1011) is 4.

Property 2.6. *e_i is the next frequent event in the mined prefix subsequence if the node e_i in the current suffix tree of e_i has a support count greater than or equal to the minimum support threshold.*

Property 2.7. *For any frequent event e_i, all frequent subsequences containing e_i can be visited by following the e_i linkage starting from the last visited e_i record in the PLWAP-tree being mined.*

Property 2.8. *For any access sequence in an access sequence database, WASD, there exists a unique path in the PLWAP-tree starting from the root such that all labels of the nodes in the path in order are exactly the same as the events in the sequence.*

This Property 2.8 ensures that the number of distinct leaf nodes as well as paths in a PLWAP-tree can not be more than the number of distinct frequent sequences in the access sequence database, and the height of the PLWAP-tree is bounded by one (for the root) plus the maximal number of instances of frequent 1-events in an access sequence. The key design points behind the PLWAP algorithm are summarized next.

1. The PLWAP-tree data structure, similar to WAP-tree, is used to store access sequences in the database, and the corresponding counts of frequent events compactly, so that the tedious support counting is avoided during mining.
2. A position code is assigned to each node in PLWAP-tree, using Rule 2.1 defined in Section 2.2. These codes are used during mining for identifying the position of the nodes in the tree. WAP-tree algorithm does not use position codes.
3. The header frequent nodes in the PLWAP-tree are linked pre-order fashion, by visiting the root first, then, the left subtree before the right subtree. The WAP-tree algorithm links the header node in the order that they are inserted. Header node linking is used to keep track of nodes with the same label for traversing prefix sequences and identifying their suffix trees and next prefix pattern efficiently.
4. An efficient recursive algorithm is proposed to enumerate web access frequent patterns from the PLWAP-tree. The philosophy of this mining algorithm is prefix sequence search rather than suffix search done by WAP-tree algorithm. Instead of searching common suffix pattern (that is, obtain the frequent pattern starting with the last event in the sequence

and keep extending the suffix subsequence until the entire sequence is found) as WAP-tree mining does, PLWAP algorithm searches common prefix pattern in the PLWAP-tree (by finding the first event in the sequence and extending this prefix subsequence until the entire sequence is found). Properties 2.1, 2.6 and 2.7, with position codes, are used to efficiently find the next prefix pattern to add to an already found prefix subsequence, while Property 2.2 proves the correctness of this extension. Other properties present features of the PLWAP-tree applied by Property 2.6 to correctly find the next prefix event.

The sequence of steps involved in the PLWAP-tree algorithm are presented next, before the formal presentation of the algorithm.

- *Step 1*: The PLWAP algorithm scans the access sequence database first time to obtain the support of all events in the event set, E. All events that have a support greater than or equal to the minimum support are frequent. Each node in a PLWAP-tree registers three pieces of information: node label, node count and node position code, denoted as label: count: position. The root of the tree is a special virtual node with an empty label and count 0. Every other node is labeled by an event in the event set E.
- *Step 2*: The PLWAP scans the database a second time to obtain the frequent sequences in each transaction. The non-frequent events in each sequence are deleted from the sequence. PLWAP algorithm also builds a prefix tree data structure, called PLWAP tree, by inserting the frequent sequence of each transaction in the tree the same way the WAP-tree algorithm would insert them. The insertion of frequent subsequence is started from the root of the PLWAP-tree. Considering the first event, denoted as e, increment the count of a child node with label e by 1 if there exists this node, otherwise create a child labeled by e and set the count to 1, and set its position code by applying Rule 2.1 above. Once the frequent sequence of the last database transaction is inserted in the PLWAP-tree, the tree is traversed pre-order fashion (by visiting the root first, the left subtree next and the right subtree finally), to build the frequent header node linkages. Auxiliary node linkage structures are constructed to assist node traversal in a PLWAP-tree. All the nodes in the tree with the same label are linked by shared-label linkages into a queue, called event-node queue. The event-node queue with label e_i is also called e_i-queue. There is one header table L for a PLWAP-tree, and the head of each event-node queue is registered.
- *Step 3*: Then, PLWAP algorithm recursively mines the PLWAP-tree using prefix conditional sequence search to find all web frequent access patterns. Starting with an event, e_i on the header list, it finds the next prefix frequent event to be appended to an already computed m-sequence frequent subsequence, by applying Property 2.6, which confirms an e_n node in the root set of e_i, frequent only if the count of all current suffix trees of e_n is frequent. It continues the search for each next prefix event along the path, using subsequent suffix trees of some e_n (a frequent 1-event in the header table), until there are no more suffix trees to search. The reason for storing the auxiliary node links in pre-order traversal mode is that it puts the nodes with the same label in the same suffix tree closely, improving mining efficiency. To mine the PLWAP-tree, the algorithm starts with an empty list of already discovered frequent patterns and the list of frequent events

in the head linkage table. Then, for each event, e_i, in the head table, it follows its linkage to first mine 1-sequences, which are recursively extended until the m-sequences are discovered. From Property 2.6, the algorithm finds the next tree node, e_n, to be appended to the last discovered sequence, by counting the support of e_n in the current suffix tree of e_i(header linkage event). Note that e_i and e_n could be the same events $\{a, b, c\}$. The mining process would start with an e_i event a, and given the PLWAP-tree, it first mines the first a event in the frequent pattern by obtaining the sum of the counts of the first e_n (i.e., a) nodes in the suffix subtrees of the Root. This event is confirmed frequent if this count is greater than or equal to minimum support. Now, the discovered frequent pattern list is $\{a\}$. To find frequent 2-sequences that start with this event a, the next suffix trees of e_i (i.e., a) are mined for events a, b, c in turn to possibly obtain frequent 2-sequences aa, ab, ac respectively if support thresholds are met. Frequent 3-sequences are computed using frequent 2-sequences and the appropriate suffix subtrees. All frequent events in the header list are searched for, in each round of mining in each suffix tree set. Once the mining of the suffix subtrees near the leaves of the tree are completed, it recursively backtracks to the suffix trees towards the root of the tree until the mining of all suffix trees of all patterns starting with all elements in the header link table are completed. An example, showing the construction and mining of the PLWAP-tree is given in Section 3.

The formal presentation of the PLWAP-tree algorithm is given as figure 5. The PLWAP-tree T, the frequent 1-sequence, and minimum support threshold are obtained either from previous call to the algorithm or user defined. These variables are used during every round of the algorithm, and thus, are considered global variables. The other variables needed by the algorithm are suffix subtrees' roots set, and the frequent sequence found from previous round. When the algorithm is called first time, the suffix subtrees roots set consists of the root of the whole PLWAP-tree. As the algorithm runs in a particular round, it is called by the previous round to find if an e_i in the header table, is the next frequent event in the sequence. Thus, it has the suffix subtree roots set of e_i node in the PLWAP-tree. It also has the frequent sequence F, as the sequence, which is ended with e_i in previous round.

Algorithm 2.1 *(PLWAP-Tree:Mines Web Log Sequences with Pre-Order Linked WAP Tree)*

```
Algorithm PLWAP-Tree()
Input:    web access sequence database WASD, minimum support $\lambda$ ($0 < \lambda \leq 1$).
Output: complete set of frequent patterns in WASD $F_i$.
begin
      (1) Scan WASD once, find all frequent individual events.
      (2) Scan WASD again, construct a PLWAP-tree over the set of individual frequent
      events using algorithm PLWAP-Construct (presented in Figure 6).
      (3) Recursively mine the PLWAP-tree using common prefix pattern search
      algorithm, PLWAP-Mine (presented in Figure 7).
end // of PLWAP-Tree //
```

Figure 5. The main PLWAP_TREE algorithm.

Table 4. Sample Web access sequence database for PLWAP-tree.

TID	Web access sequence	Frequent subsequence
100	*abdac*	*abac*
200	*eaebcac*	*abcac*
300	*babfaec*	*babac*
400	*afbacfc*	*abacc*

3. An example—Constructing and mining PLWAP-tree for frequent patterns

The same web access sequence database used in Section 1 for introducing WAP tree mining is used here as Table 4 for showing how the PLWAP algorithm constructs a PLWAP tree and how it mines the tree to obtain frequent sequences. The minimum support threshold is also set at 75%, which is 3 out of 4 records in our example. The construction of WAP-tree process is described next.

3.1. Example construction of PLWAP tree

The first step of the PLWAP algorithm of figure 5 is to scan the WASD once to obtain the supports of the events in the event set $E = \{a, b, c, d, e, f\}$ and store the frequent 1-events in the header List, L as $\{a{:}4, b{:}4, c{:}4\}$. The events $d{:}1$, $e{:}2$, $f{:}2$ have supports less than the minimum support of 3 and thus, are not frequent events. The second instruction in the PLWAP algorithm (figure 5) causes the WASD database to be scanned the second time, using only the frequent events in each transaction, and constructing pre-order linkages of the header nodes for frequent events a, b, and c. The algorithm PLWAP_Construct of figure 6 is called by the main algorithm to construct the PLWAP tree. The first instruction of figure 6 creates a root node for the PLWAP tree, and for each access sequence in the second column of the WASD (Table 4), it removes all non-frequent events d, e, f to create the frequent events in the transactions in their sequential order shown as the third column of Table 4. Thus, for the first sequence $abdac$, the frequent sequence *abac* is obtained and for the second sequence, *eaebcac*, the frequent sequence *abcac* is obtained. These frequent sequences of the WASD transactions are used for constructing the PLWAP tree, one frequent sequence after the other. Thus, the PLWAP_Construct algorithm inserts the first frequent sequence *abac* in the PLWAP tree with only the root by checking if an *a* child exists for the root. Since there is no such node yet, an *a* node is inserted as the leftmost child of the root as shown in figure 8(a). The count of this node is 1 and its position code is 1 since it is the leftmost child of the root. Next, the *b* node is inserted as the leftmost child of the *a* node since no *b* node following an *a* node exists yet. Since it is the first of its kind, its count is 1 and its position code is 11 because Rule 2.1 says to append 1 to the position code of its parent node to obtain the position code of the leftmost child. Then, the third event in this sequence, a, is inserted as a child of node b with count 1 and position code 111. Finally, the last event c is inserted with count 1 and position code 1111. All inserted

Algorithm 2.2 *(PLWAP-Construct - Constructs the Pre-Order Linked WAP Tree Mining)*

```
Algorithm PLWAP-Construct()
Input:      web access sequence database WASD, minimum support
            $\lambda$ $(0 < \lambda \leq 1)$.
            frequent 1-events $L$ (stored in the header linkage table)
Output: a Pre-Order Linked WAP Tree $T$.
Intermediate Variable:    NodeFound, current-node(pointer).
begin
(1) Create a root node of T. The position code of root is set to NULL,
      count set to 0.
(2) For each access sequence, s in the sequence database, WASD, do
      (2.1)Extract frequent subsequence, $s'$ $(s_1 s_2 \ldots s_n)$ from s by
            removing all events in s that are not in L. Set the current-node
            pointing to the leftmost child of root in tree, T.
      (2.2)For k = 1 to n (length of sequence, $s'$) do
            (2.2a) If current-node is NULL, then create a new child node ($s_k$:1).
                  with position code equal to "1" appended to position code of parent
                  of current-node.
                  else if current-node is labeled $s_k$, then set NodeFound to true
                        else let current-node point to current-node.sibling, and keep checking
                        whether current-node is labeled $s_k$, until there is no sibling
                        or $s_k$ is found
            (2.2b) If NodeFound
                  Then, increase count of $s_k$ by 1 and make current-node point to $s_k$,
                  else create a new child node ($s_k$:1) with the position code of
                        current-node.positionCode and '0' appended at the end,
                        make current-node point to the newly created node.
(3) From root, do a pre-order traversal of T, visit root, then left and
      then right subtrees and add all nodes to appropriate linkage queue.
(4) Return (T) with linkage header table L.
end // of PLWAP-Construct //
```

Figure 6. The PLWAP-construction and pre-order linkage algorithm.

nodes on the tree have information recorded as (node label: count:position code). Next, the second frequent sequence, *abcac* is inserted starting from root as shown in figure 8(b). Since the root has a child labeled *a*, thus *a*'s count is increased by 1. Now the node (*a*:1:1) is changed to (*a*:2:1). Similarly, we now have (*b*:2:11). The next event c does not match the existing node *a*. Thus, a new node is created. The new node is not the first child of its parent (*b*:2:11), thus, its position code is the position code of its closest left sibling (*a*:1:111) with 0 appended. Thus, the position code of the new node is 1110. Finally, the new node can be represented as (*c*:1:1110). The next two events are newly created as (*a*:1:11101) and (*c*:1:111011) respectively. This process keeps running until there is no more access sequence in WASD. Figure 8(c) and (d) show the PLWAP-tree after sequences *babac* and *abacc* have been inserted into the tree. Now, that the PLWAP tree is constructed, the third instruction in the algorithm PLWAP_Construct traverses the tree to construct a pre-order linkage of frequent header nodes, *a*, *b* and *c*. Starting from the root using pre-order traversal mechanism to create the queue, we first add (*a*:3:1) into the a-queue. Then, visiting its left child, (*b*:3:11) will be added to the b-queue. After checking (*b*:3:11)'s left child, (*a*:2:111) is

Algorithm 2.3 *(PLWAP-Mine - Mining the Pre-Order Linked WAP Tree)*

```
Algorithm PLWAP-Mine()
Input:     PLWAP tree T, header linkage table L,
           minimum support $\lambda$ ($0 < \lambda \leq 1$), Frequent m-sequence F).
           suffix tree roots set R (R includes root and F is empty first time algorithm is called.
Output: Frequent (m+1)-sequence, $F'$.
Other Variables:    S stores whether node is ancestor of the following nodes
                 in the queue, C stores the total number of events $e_i$ in the suffix trees.
begin
(1) If R is empty, return
(2) For each event, $e_i$ in L, find the suffix tree of $e_i$ in T (i.e, $e_e|suffixtree$), do
    (2a) Save first event in $e_i$-queue to S.
    (2b) Following the $e_i$-queue
         If event $e_i$ is the descendant of any event in R, and is not descendant of S,
              Insert it into suffix-tree-header set $R'$
              Add count of $e_i$ to C.
              Replace the S with $e_i$.
    (2c) If C is greater than $\lambda$
              Append $e_i$ after F to $F'$ and output $F'$
              Call Algorithm PLWAP-Mine of Figure 7 passing $R'$ and $F'$.
end // of PLWAP-Mine //
```

Figure 7. The PLWAP-tree mining for frequent patterns algorithm.

found and labeled as a, and it is added to the a-queue. Then, (c:2:1111) and (c:1:11111) are inserted into the c-queue. Since there is no more left child after (c:1:11111), algorithm goes backward, until it finds the sibling of (a:2:111). Thus, (c:1:1110) is inserted into c-queue. The algorithm keeps running until the whole PLWAP-tree has been traversed. Figure 8(e) shows the completely constructed PLWAP tree with the pre-order linkages.

3.2. *Example mining of constructed PLWAP tree*

The PLWAP-tree structure constructed by algorithm PLWAP_Tree Construct, which is called by the main PLWAP tree algorithm of figure 5 contains all sequences to be mined to obtain frequent patterns. Unlike WAP-tree mining, which searches access patterns with the same suffix, the PLWAP finds the access sequences with the same prefix. The third instruction of the PLWAP algorithm (figure 5) calls the PLWAP_Mine algorithm in figure 7 to recursively mine the pre-order linked WAP-tree for frequent patterns using prefix pattern search. The algorithm starts to find the frequent sequence with the frequent 1-sequence in the set of frequent events(FE) $\{a, b, c\}$. For every frequent event in FE and the suffix trees of current conditional PLWAP-tree being mined, it follows the linkage of this event to find the first occurrence of this frequent event in every current suffix tree being mined, and adds the support count of all first occurrences of this frequent event in all its current suffix trees. If the count is greater than the minimum support threshold, then this event is appended (concatenated) to the last list of frequent sequence, F. The suffix trees of these first occurrence events in the previously mined conditional suffix PLWAP-trees are now in turn, used for mining the next event. Please note that the conditional suffix PLWAP-tree,

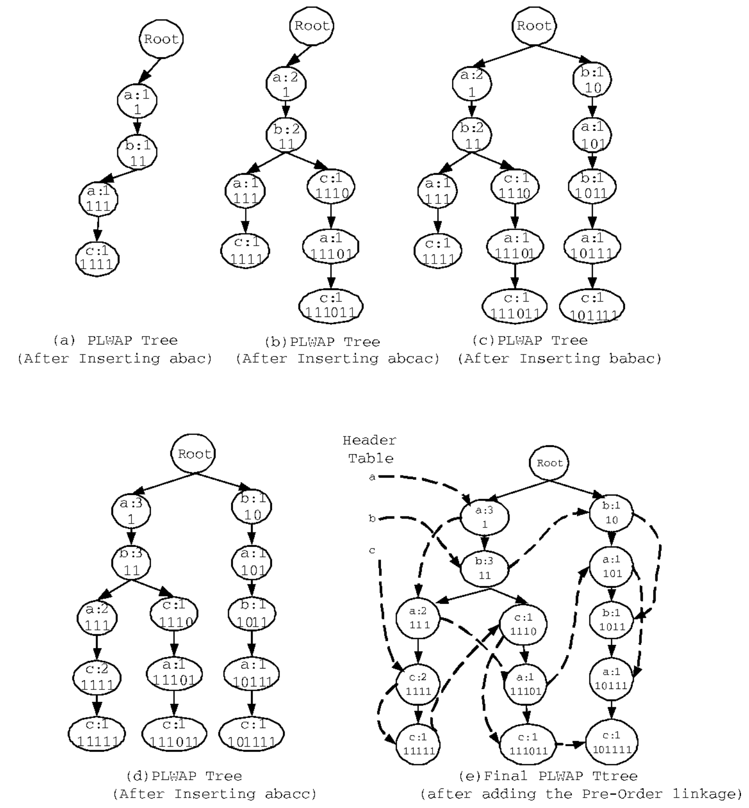
(a) PLWAP Tree (After Inserting abac)
(b)PLWAP Tree (After Inserting abcac)
(c)PLWAP Tree (After Inserting babac)
(d)PLWAP Tree (After Inserting abacc)
Header Table
(e)Final PLWAP Ttree (after adding the Pre-Order linkage)

Figure 8. Construction of PLWAP-tree using pre-order traversal.

which is obtained during the mining process does not actually physically exist. To obtain this conditional PLWAP-tree, we only need to remember the roots of the current suffix trees, which are stored for next round mining. For example, the algorithm starts by mining the tree in figure 9(a) for the first element in the header linkage list, a following the a link to find the first occurrencies of a nodes in a:3:1 and a:1:101 of the suffix trees of the Root since this is the first time the whole tree is passed for mining a frequent 1-sequence. Now the list of mined frequent patterns F is $\{a\}$ since the count of event a in this current suffix trees is 4 (sum of a:3:1 and a:1:101 counts), and more than the minimum support of 3. The mining of frequent 2-sequences that start with event a would continue with the next suffix trees of a rooted at $\{b$:3:11, b:1:1011$\}$ shown in figure 9(b) as unshadowed nodes. The objective here is to find if 2-sequences aa, ab and ac are frequent using these suffix trees. In order to confirm aa frequent, we need to confirm event a frequent in the current suffix tree set, and similarly, to confirm ab frequent, we should again follow the b link to confirm event b frequent using this suffix tree set, same for ac. The position codes of the nodes are used to identify which nodes of each suffix tree are descendants of the first event node e_i in the

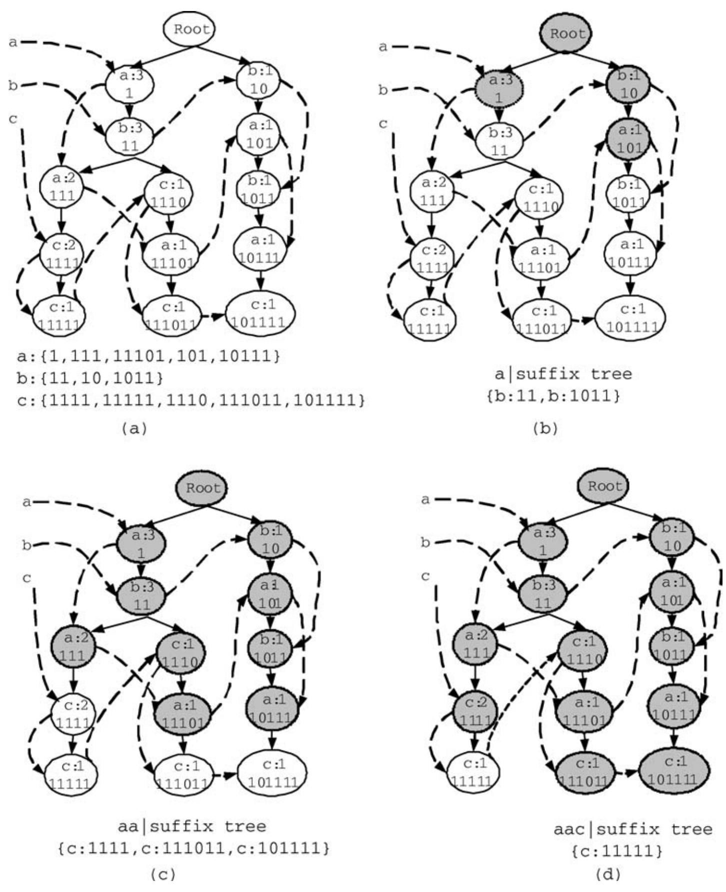

Figure 9. Mining PLWAP-tree to find frequent sequence starting with *aa*.

same subtree so that their counts are ignored. The position codes are also used to quickly identify the e_i nodes on a current suffix trees set that belong to a different subtree so that their counts can contribute to the total support count of e_i. We continue to mine all frequent events in the suffix trees of a:3:1 and a:1:101, which are rooted at b:3:11 and b:1:1011 respectively. From figure 9(b), we find the first occurrence of 'a' on each suffix tree, as a:2:111, a:1:11101 and a:1:10111 giving a total count of 4 to make a the next frequent event in sequence. Thus, a is appended to the last list of frequent sequence 'a', to form the new frequent sequence 'aa'. We continue to mine the conditional suffix PLWAP-tree in figure 9(c). The suffix trees of these 'a' nodes which are rooted at c:2:1111, c:1:111011 and c:1:101111, give another c frequent event in sequence, to obtain the sequence 'aac'. The last suffix tree (figure 9(d)) rooted at c:1:11111 is no longer frequent, terminating this leg of the recursive search. Backtracking in the order of previous conditional suffix PLWAP-tree mined, we search for other frequent events. Since no more frequent events are found in the conditional suffix PLWAP-tree in figure 9(c), we backtrack to figure 9(b), to find that b:3:11, b:1:1011 yield frequent event for b to give the next frequent sequence as *ab*. This leg of mining of patterns that begin with prefix subsequence ab is shown in figure 10(a) to (d). So far, discovered frequent patterns are $\{a, aa, aac, ab\}$. Since figure 9(b) has just been mined for *b*, the just mined *b* nodes are shadowed to give the next suffix trees to continue with, in figure 10(a). The current suffix trees to mine for frequent 3-sequences starting with ab are rooted at a:2:111, c:1:1110 and a:1:10111. Following the last *a* links, we find the first *a* nodes to sum their counts as a:2:111, a:1:11101 and a:1:10111 for a total count of 4 to be confirmed frequent. This yields the frequent pattern *aba* with the leaf end suffix trees to be mined next as shown in figure 10(b). From this figure, the current suffix trees have event *c* frequent to yield the pattern *abac* and the last suffix tree shown in figure 10(c) for the *abac* leg of mining. It backtracks to the suffix tree in figure 10(a) to look for the pattern *c* to be appended to ab using the *c* header links in this suffix tree. The nodes c:2:1111, c:1:1110 and c:1:101111 are used for its support counting as shown in figure 10(d). Since no more frequent event can be obtained from the rest of the suffix trees in this figure, it backtracks to figure 9(b) to mine for the pattern *ac* using the *c* link to get figure 10(e). This completes the mining of frequent patterns starting with event *a* and the patterns obtained so far are $\{a, aa, aac, ab, aba, abac, abc, ac\}$. This process will be repeated in turn for patterns that start with frequent events *b* and *c* respectively shown in figure 11. Finally, we have the frequent sequence set $\{a, aa, aac, ab, aba, abac, abc, ac, b, ba, bac, bc, c\}$. After checking with the final set of frequent sequences, this result is the same as the result extracted by the WAP-tree algorithm.

4. Performance analysis and experimental evaluation

The PLWAP algorithm, has various advantages over the WAP-tree algorithm, which include:

1. The PLWAP algorithm eliminates the need to store numerous intermediate WAP trees during mining. Since only the original PLWAP tree is stored, it drastically cuts off huge memory access costs, which may include disk I/O cost in a virtual memory environment, especially when mining very long sequences with millions of records. Section 4.1.4 presents results of experimental analyses showing contributions of I/O access and CPU (including main main memory) usages for both the WAP and PLWAP algorithms.
2. PLWAP also eliminates the need to store and scan intermediate conditional pattern bases for re-constructing intermediate WAP trees.
3. The PLWAP tree algorithm uses the pre-order linking of header nodes to store all events e_i in the same suffix tree closely together in the linkage, making the search process more efficient.
4. A simple technique for assigning position codes to nodes of any tree has also emerged, which can be used to decide the relationship between tree nodes without repetitive traversals.

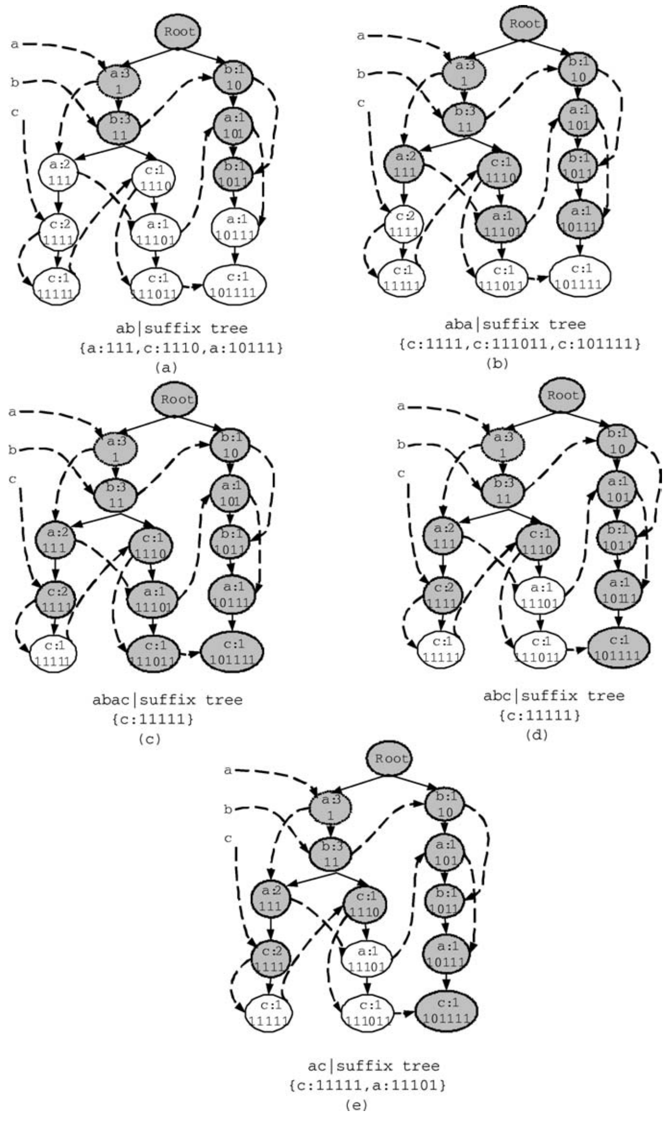

Figure 10. Mining PLWAP-tree to find frequent sequence starting with *ab* or *ac*.

Suppose the number of frequent sequential patterns that can be extracted from the WAP-tree is p and the number of 1-sequences is f, the number of constructed intermediate WAP-trees is $p - f$. The time complexity for constructing PLWAP-tree or WAP tree is the same and is $O(nl)$, where n is the number of sequences in the WASD and l is the length of the longest frequent sequence in the WASD. The time complexity for mining the PLWAP tree is $O(fp)$, where f is the number of frequent 1-events and f is the total number of frequent patterns

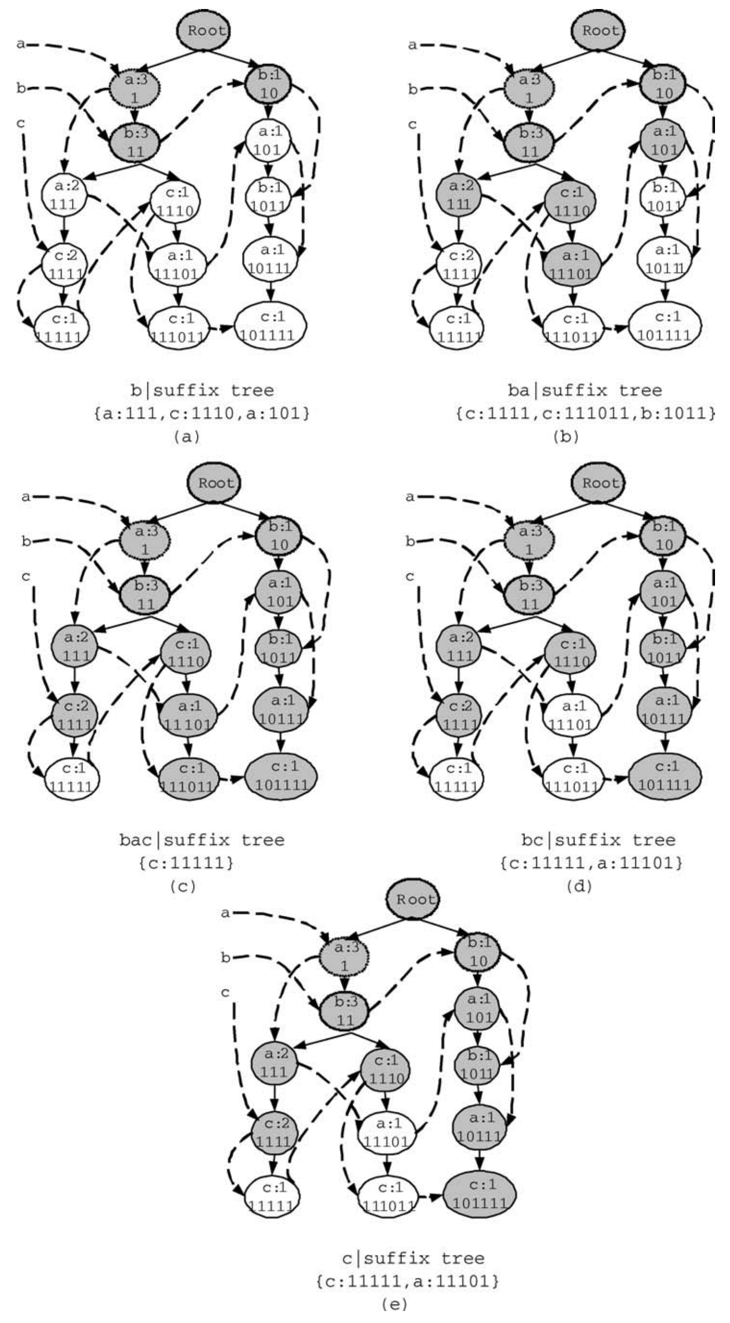

Figure 11. Mining PLWAP-tree to find frequent sequence starting with *b* or *c*.

to be discovered. This means that for WAP algorithm, this time complexity is multiplied by $(p - f)$ times needed for constructing intermediate trees. From the memory view, it can be seen that WAP-tree needs more memory space for intermediate trees and conditional databases. The position code for each node introduced by the PLWAP algorithm does not take much more space as it is based on binary tree transformation of the PLWAP tree. The height of the WAP tree is the length of the longest frequent subsequence in the web access sequence database. The web access sequence is sequence of page-views, which is produced by user visiting the web pages. Normally, the sequence length is shorter than its width. For every node in the WAP-tree, we need 1 byte to store the label, 1 byte to store the count, 2 bytes to store the links which indicate its child and its sibling. The deeper the recursive round the WAP-tree algorithm calls, the more space it needs. In the worst case, it may need the space of the original WAP-tree multiplied several times. When the original WAP-tree is large and the sequence we try to find is long, WAP-tree may face the problem of not being able to store all information in the main memory. Some I/O work may be needed between the main memory and virtual memory. For PLWAP mining, at the beginning of mining process, a small extra space is needed to store the position code. may need some little space to store the pointers of the roots set during mining.

4.1. Experimental evaluation

This section compares the experimental performance of PLWAP with the WAP-tree mining and the Apriori-like GSP algorithms. The three algorithms are implemented with C++ language running under Inprise C++ Builder environment. All experiments are performed on 400 MHz Celeron PC machine with 64 megabytes memory. The operating system is Windows 98. Synthetic datasets are generated using the publicly available synthetic data generation program of the IBM Quest data mining project at http://www.almaden.ibm.com/cs/quest/, which has been used in most sequential pattern mining studies (Srikant and Agrawal, 1996; Masseglia et al., 1998; Pei et al., 2000). The datasets consist of sequences of events, where each event represents an accessed web page. The parameters shown below are used to generate the data sets.

$\|D\|$:	Number of sequences in the database
$\|C\|$:	Average length of the sequences
$\|S\|$:	Average length of maximal potentially frequent sequence
$\|N\|$:	number of events

For example, C10·S5·N2000·D60 K means that $|C| = 10$, $|S| = 5$, $|N| = 2000$, and $|D| =$ 60 K. It represents a group of data with average length of the sequences as 10, the average length of maximal potentially frequent sequence is 5, the number of individual events in the database are 2000, and the total number of sequences in database is 60 thousand. The datasets with different parameters test different aspects of the algorithms. Basically, if the number of these four parameters becomes larger, the execution time becomes longer.

Table 5. Execution times for dataset at different minimum supports.

	Runtime (in secs) at different supports						
Algorithms	0.8	1	2	3	4	5	10
GSP	–	–	–	–	–	1678	852
WAP-tree	2080	1532	805	541	372	335	166
PLWAP-tree	492	395	266	179	127	104	69

4.1.1. Experiment 1: Execution time for different support. This experiment uses fixed size database and different minimum support to compare the performance of PLWAP, WAP and GSP algorithms. The datasets are described as C10 · S5 · N2000 · D60 K, and algorithms are tested with minimum supports between 0.8% and 10% against the 60 thousand (60 K) database.

From Table 5 and figure 12, it can be seen that the execution time of every algorithm decreases as the minimum support increases. This is because when the minimum support increases, the number of candidate sequence decreases. Thus, the algorithms need less time to find the frequent sequences. Among these three algorithms, the GSP algorithm is the most time-consuming algorithm. It is hard to generate and test the candidate sequences when the number of frequent 1-sequence becomes large. Suppose the frequent 1-sequence is 9, the candidate 2-sequences will be 81, the testing work against the 60 K database is huge. As shown in figure 12, the execution time of GSP increases sharply, when the minimum support decreases. The GSP algorithm faces a problem when the minimum support is set to 4% or lower. It took so long that the process had to be aborted. The PLWAP algorithm always uses less runtime than the WAP algorithm. WAP tree mining incurs higher storage

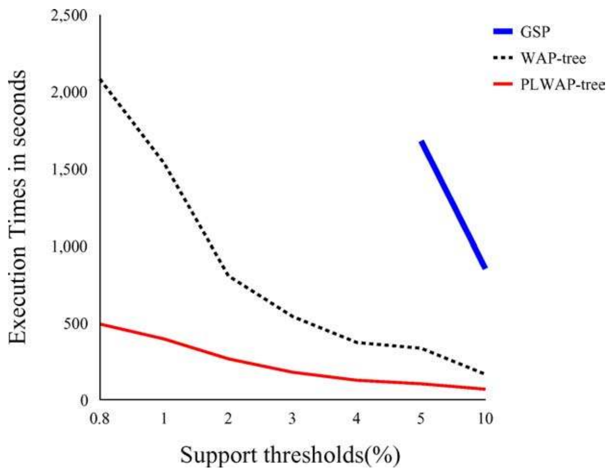

Figure 12. Execution times trend with different minimum supports.

Table 6. Execution times at different data sizes on support 5%.

Algorithms (times in secs)	Different changed transaction size				
	20 K	40 K	60 K	80 K	100 K
WAP-tree	158	272	335	464	566
PLWAP-tree	54	81	104	146	185

cost (memory or I/O). Even in memory only systems, the cost of storing intermediated trees adds appreciably to the overall execution time of the program. It is however, more realistic to assume that such techniques are run in regular systems available in many environments, which are not memory only, but could be multiple processor systems sharing memories and CPU's with virtual memory support. As the minimum support threshold decreases, the number of events that meet minimum support increases. This means that WAP-tree becomes larger and longer, and the algorithm needs much more I/O work during mining of WAP-tree. As minimum support decreases, the execution time difference between WAP-tree and PLWAP increases.

4.1.2. Experiment 2: Execution times for databases with different sizes. In this experiment, databases with different sizes from 20 K to 100 K with the fixed minimum support of 5% were used. The five datasets are C10 · S5 · N2000 · D20 K, C10 · S5 · N2000 · D40 K, C10 · S5 · N2000 · D60 K, C10 · S5 · N2000 · D80 K and C10 · S5 · N2000 · D100 K. Only the execution times of the WAP and PLWAP algorithms were compared and the results of this experiment are presented in Table 6 and figure 13.

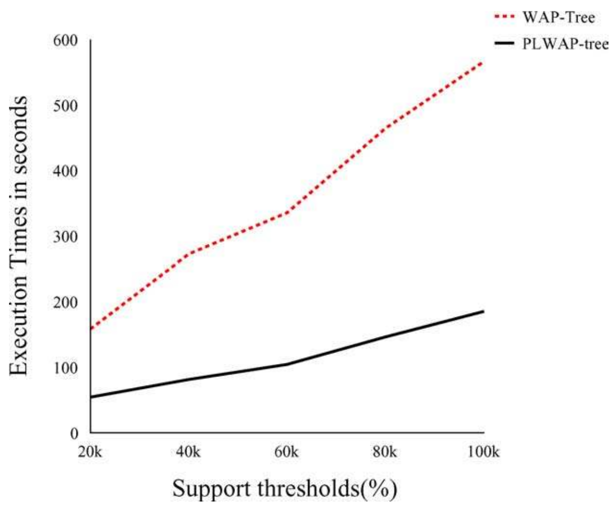

Figure 13. Execution times trend with different data sizes.

Table 7. Execution times with different sequence lengths at support 1%.

Algorithms (times in secs)	Different changed transaction size 10 (5)	20 (8)	30 (10)
WAP-tree	402	1978	5516
PLWAP-tree	217	743	1751

When the data size increases, the cost of scanning original database twice and building original WAP-tree or PLWAP-tree increases. However, much more time is needed by WAP-tree than PLWAP algorithm for mining the tree because of WAP algorithm's need to store intermediate patterns and construct intermediate WAP-trees.

4.1.3. Experiment 3: Different length sequences.

In Experiment 3, the performance of WAP and PLWAP algorithms for the sequences with varying average lengths of 10, 20 and 30 was observed. Three datasets used to do the comparison are: C10 · S5 · N1500 · D10 K, C20 · S8 · N1500 · D10 K and C30 · S10 · N1500 · D10 K. The minimum support is set at 1%.

From the result shown in Table 7 and figure 14, it can be clearly seen that the PLWAP algorithm is much more efficient than the WAP algorithm. When the sequence becomes longer, the WAP algorithm needs more time to construct the intermediate WAP trees.

4.1.4. Experiment 4: Analyses of CPU and IO usages of PLWAP and WAP.

This section discusses results of experiments conducted on a UNIX SUN microsystems with a total of 8192 Mb memory and 12 × 336 MHz processor speed, intended for analyzing portions of

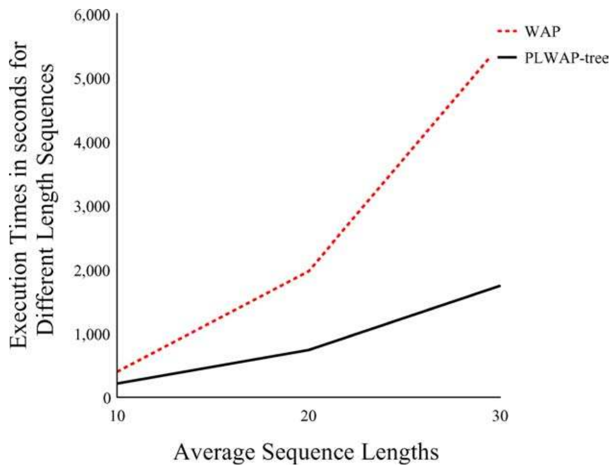

Figure 14. Execution times trend with different length sequences.

Table 8. Analyses of CPU and IO usages of PLWAP and WAP.

MinSupport (%)		5	7	9	15	20	30	50
CPU Time	PLWAP	4.47	3.15	2.24	1.32	1.04	0.77	0.52
(sec)	WAP	13.4	10.16	8.75	6.09	4.47	3.32	1.13
I/O Time	PLWAP	0.22	0.09	0.08	0.07	0.06	0.02	0.01
(sec)	WAP	1.52	1.22	0.86	0.44	0.43	0.2	0.08
Overall	PLWAP	4.73	3.26	2.39	1.39	1.14	0.8	0.54
Response (sec)	WAP	15.27	11.47	9.79	6.63	4.96	3.54	1.24

the overall execution times of the two algorithms, PLWAP and WAP are that devoted to CPU execution of instructions including access to main memory, and the portions that are spent in “kernel” mode processing I/O requests. A number of experiments were run on two databases of one thousand and ten thousand records in order to collect the CPU time, I/O time and overall response time in seconds of both algorithms for minimum supports of 5% to 50%. Since the I/O and CPU usage behavior patterns are regular across different database sizes, results of the experiments on the one thousand record database, are summarized in Table 8 and used to discuss these behaviors next. From Table 8, comparative I/O and CPU usage times of the algorithms are derived as shown in Table 9.

From Table 9 (columns 6 and 8), it can be seen that I/O access accounts for 5% or less of the overall program execution time of the PLWAP algorithm, while its CPU usage, including main memory access accounts for 95% or more of its overall execution time. The WAP algorithm from Table 9 (columns 7 and 9) invests more on I/O usage than the PLWAP algorithm, and the portions of its overall execution time that goes to I/O usage is 6% to 13%, but its CPU usage accounts for 87% to 94% of its overall program execution time. Thus, CPU time dominates the I/O time in both algorithms although the WAP algorithm spends more time on I/O operations than the PLWAP algorithm. Table 9 (columns 3) shows that the WAP algorithm for the same tasks would require 5 to 13 times more I/O time than

Table 9. CPU and IO usages of PLWAP(PW) and WAP derived from Table 8.

Min-Sup %	PW I/O time	Increase in WAP I/O need	PW CPU time	Increase in WAP CPU need	PW time % to I/O	WAP time % to I/O	PW time % to CPU	WAP time % to CPU
5	0.22	5.9 PW’s	4.47	2.0 PW’s	4.7	10	95	90
7	0.09	12.6	3.15	2.23	2.82	13.0	97.2	87.0
9	0.08	9.75	2.24	2.91	3.4	8.9	96.6	91.1
15	0.07	5.3	1.32	3.61	5	6.74	95	93.3
20	0.06	6.2	1.04	3.3	5.4	8.8	94.6	91.2
30	0.02	9.0	0.77	4.2	2.5	5.7	97.5	94.3

PLWAP algorithm. Table 9 (columns 5) shows that the WAP algorithm for the same tasks would require 2 to 5 times more CPU time than PLWAP algorithm. Thus, in a memory-only environment, where the WAP algorithm would not incur direct disk I/O cost, the about 10% of its running time that would have gone to I/O processing, goes back to CPU and main memory access, which are still about 4 times higher than the PLWAP CPU time. This means that the gain in response time achieved by the PLWAP algorithm over the WAP algorithm, remains very significant even in a memory-only environment. This experiment supports the theoretical analysis, which states that the cost of memory access (primary or virtual) incurred by the WAP algorithm is higher than that by the PLWAP algorithm, because the WAP algorithm constructs and accesses intermediate patterns and trees during mining.

4.1.5. Correctness of algorithm implementations. To show that the implementations of the three algorithms are correct, a small sample database (Table 10), similar to that used in Section 3, is used to test the correctness of implementations.

After running the three algorithms against the database in Table 10, with a minimum support of 0.75, the results of frequent sequences are shown in Table 11. By comparing this output with results produced from the theoretic work in Section 3, the results of these three algorithms are the same. The only difference between these results is the order of presentation of the computed frequent sequences. Thus, these results, allow us to confirm that the experiments conducted in Section 4.1 are based on correct and working algorithm implementations.

Table 10. Sample database for testing implementation correctness.

SID	Events
100	10, 20, 40, 10, 30
200	50, 10, 50, 20, 30, 10, 30
300	20, 10, 20, 60, 10, 50, 30
400	10, 60, 20, 10, 30, 60, 30

Table 11. Results to confirm correctness of PLWAP and algorithm implementations.

Algorithm	Events
GSP	((10), (20), (30), (10, 10), (10, 20), (10, 30), (20, 10), (20, 30) (10, 10, 30), (10, 20, 10), (10, 20, 30), (20, 10, 30), (10, 20, 10, 30))
WAP	((10, 20, 30), (20, 30), (10, 20, 10, 30), (20, 10, 30), (10, 10, 30), (10, 30), (30), (10, 20), (20), (10, 20, 10), (20, 10), (10, 10), (10))
PLWAP	((10), (10, 10), (10, 10, 30), (10, 20), (10, 20, 10), (10, 20, 10, 30), (10, 20, 30), (10, 30), (20), (20, 10), (20, 10, 30), (20, 30), (30))

5. Conclusions and future work

This paper presents a new algorithm (PLWAP) for efficiently mining sequential patterns from web log. The PLWAP algorithm adapts the WAP-tree structure for storing frequent sequential patterns to be mined. However, to improve on mining efficiency, the paper proposes to find common prefix patterns instead of suffix patterns as done by WAP-tree mining. Moreover, in order to avoid recursively re-constructing intermediate WAP-trees, pre-order frequent header node linkages and position codes are proposed. While the pre-order linkage provides a way to traverse the event queue without going backwards, position codes are used to identify the position of nodes in the PLWAP tree. With these two methods, the next frequent event in each suffix tree is found without traversing the whole WAP-tree. Thus, it avoids re-constructing WAP-tree recursively. The experiments show that mining web log using PLWAP algorithm is much more efficient than with WAP-tree and GSP algorithms, especially when the average frequent sequence becomes longer and the original database becomes larger. For mining sequential patterns from web logs, the following aspects may be considered for future work. As synthetic datasets were used in our experiment, there is no need to transform the original web log to sequential database. The procedure for transforming the web log to database is still time-consuming and could be improved upon for web log mining. The PLWAP algorithm could be extended to handle sequential pattern mining in large traditional databases other than web log. For mining sequential patterns in transaction databases, there is need to handle concurrency of events. Thirdly, efficient web usage mining, could benefit from relating usage to the content of web pages. Other areas of interest for future work include distributed mining with PLWAP trees and applying these techniques to incremental mining of web logs and sequential patterns.

References

Agrawal, R. and Srikant, R. 1994. Fast algorithms for mining association rules in large databases. In Proceedings of the 20th International Conference on Very Large Databases. Santiago, Chile, pp. 487–499.

Agrawal, R. and Srikant, R. 1995. Mining sequential patterns. In Proceedings of the 11th International Conference on Data Engineering (ICDE '95). Taipei, Taiwan, pp. 3–14.

Borges, J. and Levene, M. 1999. Data mining of user navigation patterns. In Proceedings of the KDD Workshop on Web Mining. San Diego, California, pp. 31–36.

Berendt, B. and Spiliopoulou, M. 2000. Analyzing navigation behaviour in web sites integrating multiple information systems. VLDB Journal, Special Issue on Databases and the Web, 9(1):56–75.

Etzioni, O. 1996. The world wide web: Quagmire or gold mine. Communications of the ACM, 39(1): 65–68.

Han, J. and Kamber, M. 2000. Data Mining: Concepts and Techniques. Morgan Kaufmann Publishers.

Han, J., Pei, J., Mortazavi-Asl, B., Chen, Q., Dayal, U., and Hsu, M.-C. 2000. FreeSpan: Frequent pattern-projected sequential pattern mining. In Proceedings of the 2000 Int. Conference on Knowledge Discovery and Data Mining (KDD'00). Boston, MA, U.S.A., pp. 355–359.

Han, J., Pei, J., Yin, Y., and Mao, R. 2004. Mining frequent patterns without candidate generation: A frequent-pattern tree approach. International Journal of Data Mining and Knowledge Discovery. Kluwer Academic Publishers, 8(1): 53–87.

Madria, S.K., Bhowmick, S.S., Ng, W.K., and Lim, E.P. 1999. Sampling large databases for finding association rules. In Proceedings of the First International Data Warehousing and Knowledge Discovery, DaWak99. pp. 303–312.

Mannila, M., Toivonen, H., and Verkamo, A.I. 1995. Discovering frequent episodes in sequence. In Proceedings of the First International Conference on Knowledge Discovery and Data Mining. Montreal, Quebec, pp. 144–155.

Masseglia, F., Cathala, F., and Poncelet, P. 1998. PSP: Prefix tree for sequential patterns. In Proceedings of the 2nd European Symposium on Principles of Data Mining and Knowledge Discovery (PKDD'98). Nantes, France, LNAI, pp. 176–184.

Masseglia, F., Poncelet, P., and Cicchetti, R. 1999. An efficient algorithm for web usage mining. Networking and Information Systems Journal (NIS), 2(5/6):571–603.

Nanopoulos, A. and Manolopoulos, Y. 2000. Finding generalized path patterns for web log data mining. Data and Knowledge Engineering, 37(3):243–266.

Nanopoulos, A. and Manolopoulos, Y. 2001. Mining patterns from graph traversals. Data and Knowledge Engineering, 37(3):243–266.

Park, J.S, Chen, M.S., and Yu, P.S. 1997. Using a Hash-based method with transaction trimming for mining association rules. IEEE Transaction on Knowledge and Data Engineering, 9(5):813–825.

Pei, J., Han, J., Mortazavi-Asl, B., and Zhu, H. 2000. Mining access patterns efficiently from web logs. In Proceedings of the Pacific-Asia Conference on Knowledge Discovery and Data Mining (PAKDD'00). Kyoto, Japan.

Pei, J., Han, J., Mortazavi-Asl, B., and Pinto, H. 2001. PrefixSpan: Mining sequential patterns efficiently by prefix-projected pattern growth. In Proceedings of the 2001 International Conference on Data Engineering (ICDE '01). Germany, Heidelberg, pp. 215–224.

Srikant, R. and Agrawal, R. 1996. Mining sequential patterns: Generalizations and performance improvements. In Proceedings of the Fifth International Conference On Extending Database Technology (EDBT '96). Avignon, France, pp. 3–17.

Srivastava, J., Cooley, R., Deshpande, M., and Tan, P. 2000. Web usage mining: Discovery and applications of usage patterns from web data. SIGKDD Explorations, Vol. 1.

Shaffer, C.A. 2000. A Practical Introduction to Data Structures and Algorithm Analysis. Prentice Hall Inc.

Spiliopoulou, M. 1999. The laborious way from data mining to web mining. Journal of Computer Systems Science and Engineering, Special Issue on Semantics of the Web, 14:113–126.

Christie I. Ezeife received her M.Sc. in Computer Science from Simon Fraser University, Canada in 1988 and a Ph.D in Computer Science from the University of Manitoba, Canada in 1995. She has held academic positions in a number of universities and is an associate professor of Computer Science at the University of Windsor, Canada. Her research interests include distributed object-oriented database systems, data warehousing and mining. She has authored several technical publications including three comprehensive journal articles in the Kluwer's international journals of Distributed and Parallel Databases, Data Mining and Knowledge Discovery, two articles in Elsevier's journal of Data and Knowledge Engineering and IDEA group international journal of Data Warehousing and Mining, as well as a book on Problem Solving and Programs with C by Thomson Learning Publishers.

Yi Lu received his M.Sc. in Computer Science from the School of Computer Science, University of Windsor, Canada in 2002. He had a prior B.Sc. in Computer Science from the Department of Computer Science and Technology in Hangzhou University (now known as Zhejiang University), Hangzhou, PR China in 1993. He is currently a Ph.D. student in the Department of Computer Science at Wayne State University, Detroit, Michigan, U.S.A. His research interests include web mining, bio-informatics and XML database.