2009

Mining very long sequences with PLWAPLong algorithms

Kashif Saeed
University of Windsor

Mining Very Long Sequences with PLWAPLong Algorithms

By

Kashif Saeed

A Thesis
Submitted to the Faculty of Graduate Studies
through the School of Computer Science
in Partial Fulfillment of the Requirements
for the Degree of Master of Science
at the
University of Windsor

Windsor, Ontario, Canada

2009

Your file Votre référence
ISBN: 978-0-494-57610-6
Our file Notre référence
ISBN: 978-0-494-57610-6

Mining Very Long Sequences with PLWAPLong Algorithms

By

Kashif Saeed

APPROVED BY:

Dr. Sévérien Nkurunziza, External Reader

Dept. of Mathematics & Statistics

Dr. Luis Rueda, Internal Reader

School of Computer Science

Dr. Christie I. Ezeife, Advisor

School of Computer Science

Dr. Robert Kent, Chair

School of Computer Science

Author's Declaration of Originality

I hereby certify that I am the sole author of this thesis and that no part of this thesis has been published or submitted for publication.

I certify that, to the best of my knowledge, my thesis does not infringe upon anyone's copyright nor violate any proprietary rights and that any ideas, techniques, quotations, or any other material from the work of other people included in my thesis, published or otherwise, are fully acknowledged in accordance with the standard referencing practices. Furthermore, to the extent that I have included copyrighted material that surpasses the bounds of fair dealing within the meaning of the Canada Copyright Act, I certify that I have obtained a written permission from the copyright owner(s) to include such material(s) in my thesis and have included copies of such copyright clearances to my appendix.

I declare that this is a true copy of my thesis, including any final revisions, as approved by my thesis committee and the Graduate Studies office, and that this thesis has not been submitted for a higher degree to any other University or Institution.

Abstract

Sequential pattern mining is the process of finding inter-transaction frequent sequential patterns from a sequential database, where records consist of ordered sets of events (or items), by applying data mining techniques on such sequential databases. Discovering sequential patterns in web server logs is an example application of sequential mining, which is useful for predicting visiting patterns of web users for such purposes as targeted advertisements. Position Coded Pre-order Linked Web Access Pattern (PLWAP) mining algorithm is one of the existing efficient web sequential pattern mining algorithms, which stores the frequently stored sequences of the entire sequential database in a compressed tree form with position coded nodes.

However, for very long sequences exceeding thirty two nodes, the number of bits an integer position code can hold, the PLWAP algorithm's performance begins to degrade because it employs linked lists to store conjunctions of long position codes and the linked list traversals slow down the algorithm both during tree construction and mining. PLWAP algorithm also uses each and every node in the frequent 1-item event queue to test for that event inclusion in the suffix tree root set during mining. This is a very expensive operation since except for one node all other nodes that are its ancestors and descendents are not included in the root set.

This thesis proposes two new algorithms, i.e. PLWAPLong1 and PLWAPLong2. Both of these new algorithms use a new position code numbering scheme where each node is assigned two numeric variables (startPosition, endPosition) instead of one. Using this scheme we can determine the ancestor node in O (1) operation by comparing the startPosition and endPosition of two nodes. PLWAPLong1 algorithm also proposes transforming the linked list based tree to an equivalent array representation and using binary search to find the immediate descendant in a suffix tree. PLWAPLong2 uses existing linked list based tree. Both PLWAPLong1 and PLWAPLong2 algorithms introduce a new technique called "Last Descendant" to eliminate the unwanted nodes from ancestor/descendent test when creating the suffix tree root set.

Keywords: Data mining, Web Mining, Association Rule Mining, Long Sequences, PLWAP Mining

Dedication

To My Family

Acknowledgement

I would like to give my sincere appreciation to all of the people who have helped me throughout my education. I express my heartfelt gratitude to my Mother, Father, Wife, Daughter and Sisters for their prayers, moral and financial support throughout my graduate studies.

I am very grateful to my supervisor, Dr. Christie Ezeife for her continuous support throughout my graduate study. She always guided me and encouraged me throughout the process of this research work, taking time to read all my thesis updates.

I would also like to thank my external reader, Dr. Sévérien Nkurunziza, my internal reader, Dr. Luis Rueda, and my thesis committee chair, Dr. Robert Kent for making time to be in my thesis committee, reading the thesis and providing valuable input. I appreciate all your valuable suggestions and the time, which have helped improve the quality of this thesis.

At last, I would express my appreciations to all my friends and colleagues, for their help and support. Thank you all!

Table of Contents

Table of Figures

Table of Tables

1. Introduction

Organizations that have large amount of data need to make decisions that impact their future activities. Data mining is a process of extracting relevant and important knowledge from that large data to facilitate decision making. Automatic discovery of user access patters from web usage log is known as web usage mining. Analysis of such pattern can help improve server performance, restructuring of a web site and better marketing strategies in e-commerce web sites. It can also be used to find significant user actions like sending an email or searching [SM+99]. In this thesis, we study efficient mining of sequential patterns from web usage log. This chapter is organized as follows: section 1.1 introduces web mining and its categories; section 1.2 introduces phases of web usage mining.

1.1. Web Mining

The growth of World Wide Web (WWW) has amazing impact on our everyday life. Online shopping, following news feeds, keeping track of stock market, weather update, banking or simply conducting online business, World Wide Web has grown in both volume and traffic. This growth has brought new challenges to web site design, web server design and also the web site navigation. When data mining techniques are applied on web data it becomes web mining. Web mining finds interesting patterns from the web by applying automatic mining techniques. Interesting patterns are discovered from web contents or web pages, web links in the web pages or web logs. This web data can be collected at the server-side, client-side, proxy-servers or organizations' database.

According to [BL99], [MBN+99] and [SCD+00] web mining can be categorized into web content mining, web structure mining and web usage mining.

1.1.1. Web Content Mining

Web sites are built up with text, hyperlinks, images, scripts and multimedia. Web content mining discovers useful information from these data items that make up the website.

1.1.2. Web Structure Mining

A website usually consists of more than one web page. These web pages are connected to each other using hyperlinks. Web structure mining discovers the patterns from such hyperlinks. These patterns represent the underlying model of the website and such a model then can be used to categorize the web pages. Using such patterns, similarity and relationship with different websites can be discovered.

1.1.3. Web Usage Mining

Although sequential pattern mining technique can be used with applications other than the web-based, focus of this thesis will only be on the web usage. Web usage mining finds relationships between and patterns in access log files generated by the user's visit to the website. It provides straightforward statistics, such as page access frequency along with finding common traversal paths with the website [CMS99]. According to [BM98], log files can be of three kinds: server log, error log and cookie log. A typical web log at least consists of entries shown in Table -1. An Example of a line of data in a web log is

137.207.76.120–[30/Aug/2001:12:03:24-0500] "GET/jdk1.3/docs/relnotes/deprecatedlist.html HTTP/1.0" 200 2781"

Where 137.207.76.120 is the host/ip, '–'represents annonymus user, [30/Aug/2001:12:03:24-0500] is the [date:time], "GET/jdk1.3/docs/relnotes/deprecatedlist.html HTTP/1.0" is the

"request url", '200' is the status of the URL request and 2781 is the number of bytes requested.

Field	Description	Example
Host/ip	Remote client IP address	111.125.12.112
User	Remote log user name	Xyz or '–' for anonymous user.
Date	Date, time and time zone of request	14/Jan/2008:10:01:01-0400
Request URL	User request identifier (URI) with the uniform resource locator(URL) string	URI: http, ftp, mailto etc URL: http://cricket.resultsvault.com/cricket/reports/matchmenu.asp
Status	Status code returned to the client	200 [series of success], 300 [series of redirect], 400 [series of failure] and 500 [series of server error]
Bytes	Bytes transferred (sent and received)	1024 bytes

Table 1 Sample Web Log

With the help of web usage mining, organizations can monitor the browsing behavior of users on the web-based applications. For example, if users access page with URL /products/games/hardware.html, 95% of times they also access page /products/games/accessories.html. Such information can be useful in laying out the links closer to each other hence providing easier browsing routes. Such patters can also help in doing better marketing of products to targeted users. Web usage data can be obtained from TCP-IP packets using sniffing software [SCD+0O], cookie log data, query data and click stream data [SCD+0O] [BM98].

1.2. Phases of web usage mining

Web usage mining can be categorized into three phases: preprocessing, Knowledge/pattern discovery and pattern analysis.

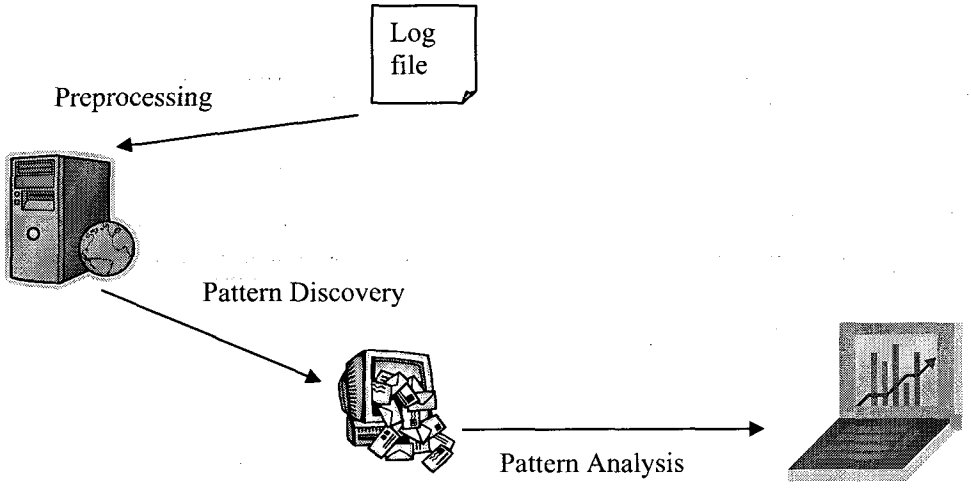


Figure 1 Web Usage Mining Phases

1.2.1. Preprocessing

Preprocessing constitutes about 80% of the work of data mining tasks [AKM+01]. According to [CMS99] following tasks may be included in the preprocessing phase: Data cleaning, User-Identification, session-identification, path-completion, transaction-identification, formatting.

1.2.1.1. Data Cleaning

Web pages these days are rich with images, multimedia, scripts and hyperlinks. When user visits the web site and requests for a web page, all the data ingredients of that page

get recorded in the server log file. For web log analysis, only the html page should be listed in the navigational path of the log file. Multimedia embedded files, images, scripts and hyperlinks all become noise data and should be cleaned.

1.2.1.2. User Identification

Web logs contain IP addresses to identify the users. Sometimes, IP address can not uniquely identify every user because users share their computers or are connected through proxy servers. According to [CMS99], log/site method can be employed to identify unique users. Users and user sessions also become difficult to identify from the web logs because of the stateless nature of HTTP [CMS99, CP95, P97, SP+98]

1.2.1.3. Formatting

Formatting is the final step in preprocessing. A predefined module is used to format the cleaned log that could then be used for data mining.

1.2.2. Pattern Discovery

Once the usage log is cleaned and formatted, it is ready to be operated by various algorithms and methods. Let's see some of the methods applicable to web usage mining.

Association Rule Mining:

According to [AIS93], association rule is an implication of the form $X \Rightarrow Y_i$, where X is a set of some items in Y, and Y_i is a single item in Y that is not present in X and Y being the set of all the items. Association rule mining on web usage data helps

capturing relationships among page views based on user navigational patterns [MDKN01].

Clustering:

Clustering allows grouping of users or data items on the basis of similar characteristics [CMS99]. Web server logs generated by user's visits to a website can be used to cluster user information to learn and develop future marketing plans.

Sequential Pattern Mining:

While association rule mining discovers intra-transaction patterns (patterns within same transaction), sequential pattern mining finds inter-transactions (patterns in more than one transaction) from a set of time-ordered items.

1.2.3. Pattern Analysis

Pattern analysis comes at the end of the web usage mining process. After applying mining techniques and algorithms patterns are analyzed and uninterested patterns are discarded. OLAP operations can also be performed on the load usage data once it is loaded into a data cube [SCD+00].

1.3. Sequential Pattern mining

Sequential pattern discovery is to find inter-transaction patterns such that presence of one set of items is followed by another set, where items are ordered by their respective transaction time. Sequential mining is a process of applying data mining techniques on such sequential databases. Discovering sequential patterns in web server logs is helpful in predicting visiting patterns of web users in turn helping targeted advertisement or grouping related information depending on frequent sequential web accesses.

1.4. Thesis Contribution

This thesis proposes two new algorithms, PLWAPLong1 and PLWAPLong2, to efficiently find sequential patterns from long sequences.

Both PLWAPLong1 and PLWAPLong2 are based on PLWAP algorithm [LE03] [LE05]. PLWAP algorithm stores the entire sequential database in a compressed tree form with position coded nodes. However, for very long sequences exceeding thirty two nodes, the PLWAP algorithm's performance begins to degrade because it employs linked lists to store conjunctions of long position codes and the linked list traversals slow down the algorithm both during tree construction and mining [ELL05]. PLWAP algorithm also uses each and every node in the frequent 1-item event queue to test for that event inclusion in the suffix tree root set during mining. This is a very expensive operation since except for one node all other nodes that are its ancestors and descendants are not included in the root set.

The new proposed algorithms, i.e. PLWAPLong1 and PLWAPLong2, use a new position code numbering scheme where each node is assigned two numeric variables (startPosition, endPosition) instead of one. Using this scheme we can determine the ancestor node in O(1) operation by comparing the startPosition and endPosition of two nodes. The PLWAPLong1 algorithm also proposes transforming the linked list based tree to an equivalent array representation and using binary search to find the immediate descendant in a suffix tree. The PLWAPLong2 algorithm uses same linked list based tree structure as that used by PLWAP algorithm. Both PLWAPLong1 and PLWAPLong2 algorithms introduce a new technique called "Last Descendant" to eliminate the unwanted nodes from ancestor/descendent test when creating the suffix tree root set.

1.5. Outline of the Thesis Proposal

The remaining of the thesis proposal is organized as follows: Chapter 2 reviews related work to this thesis. Chapter 3 details discussion of the problem addressed along with the new algorithms proposed. Chapter 4 gives performance analysis and experimental results. Chapter 5 draws the conclusion of this research and discusses future work.

2. Previous/Related Work

2.1. Association Rules

According to [AIS93], association rule is an implication of the form $X \Rightarrow Y_i$, where X is a set of some items in Y, and Y_i is a single item in Y that is not present in X and Y being the set of all the items. Association rule mining on web usage data helps capturing relationships among page views based on user navigational patterns.

[AIS93] introduced many algorithms to apply association rule mining on market basket data. Market basket data contains list of items bough by customers in each transaction. With the advancements in bar-code scanning, task of recording such information has also become easy and fast. Mining such basket-data helps management of supermarkets in making decision like what two items to be put together, what items to be put on sale or what coupons should be designed and etc. Association rule provides information in the form of "if-then" statement. The "if" or the first part is the antecedent and the "then" part is the consequence. Association rules make use of two numbers i.e. support and confidence, to measure the degree of uncertainty in the rule. Support is the number of transactions that include all items in the antecedent and consequence parts of the rule. Confidence on the other hand is ratio of the number of the transactions that include all the items in consequence as well as the antecedent.

2.2. Apriori

[AIS93] introduced an algorithm to apply association rule mining over the market-basket data. Their algorithm, also know as AIS algorithm, answered the key problems of rule mining, i.e.

1. Generation of large itemsets. Large itemsets are those that have fractional transaction support greater than a certain threshold called minsupport. This step is also called join step.
2. Generating all rules from the large itemset. This step is also called prune step.

AIS algorithm's main drawback is that it makes multiple passes over the database to find all association rules [AS94]. It turns out to be exponentially large. [AS94] has presented three algorithms that outperform AIS algorithm. They all use the common function called apriori-gen that reduces the candidate itemsets.

Transaction ID (TID)	Items purchased
1	I_1, I_2, I_5
2	I_2, I_4
3	I_2, I_3
4	I_1, I_2, I_4
5	I_1, I_3
6	I_2, I_3
7	I_1, I_3
8	I_1, I_2, I_3, I_5
9	I_1, I_2, I_3

Table 2 Transaction DB

The essence of apriori algorithm is that it employs iterative approach know as level-wise search. With such a technique, the algorithm uses prior knowledge of frequent itemset found at previous level to find the frequent itemset at present level. Algorithm starts with finding frequent-1 itemset called L_1 which is used to find L_2. The set of frequent-2

itemset is used to find L_3, and so on, until no more frequent k-itemsets can be found. Let us run through the transaction database shown in Table-2 to see how apriori algorithm works. Let us assume that the minsupport is 2 transactions. In our example it would be 2/9 = 22%.

1. In first iteration, each item belongs to the candidate 1-itemsets, C1. A database scan is performed to get the occurrence count of each item.

C_1

Itemset	Support count
I_1	6
I_2	7
I_3	6
I_4	2
I_5	2

Table 3 Candidate 1-itemset

2. Now the algorithm will generate the L_1, set of frequent 1-itemsets, by satisfying the minsupport of candidate 1-itemsets.

L_1

Itemset	Support count
I_1	6
I_2	7
I_3	6
I_4	2
I_5	2

Table 4 Frequent 1-itemsets

3. With the L_1 itemset in hand, algorithm will now perform a self join of L_1 i.e. L_1 x L_1 to generate a candidate 2-itemsets satisfying minsupport.

C_2

Itemset
$\{I_1, I_2\}$
$\{I_1, I_3\}$
$\{I_1, I_4\}$
$\{I_1, I_5\}$
$\{I_2, I_3\}$
$\{I_2, I_4\}$

{ I_2, I_5}
{ I_3, I_4}
{ I_3, I_5}
{ I_4, I_5}

Table 5 Candidate 2-itemsets

4. Algorithm will now use the C_2 to find the minsupport of each 2-itemset by scanning the database once more. Support count of each candidate itemset in C2 is accumulated as shown below.

C_2

Itemset	Support Count
{ I_1, I_2}	4
{ I_1, I_3}	4
{ I_1, I_4}	1
{ I_1, I_5}	2
{ I_2, I_3}	4
{ I_2, I_4}	2
{ I_2, I_5}	2
{ I_3, I_4}	0
{ I_3, I_5}	1
{ I_4, I_5}	0

Table 6 Support Count of C2

5. Using the last step data set, L_2 will be generated. By joining L_2 x L_2 we will get the C3 candidate itemset.

L_2

Itemset	Support Count
{ I_1, I_2}	4
{ I_1, I_3}	4
{ I_1, I_5}	2
{ I_2, I_3}	4
{ I_2, I_4}	2
{ I_2, I_5}	2

Table 7 Frequent 2-itemsets

C3

Itemset
{ I_1, I_2, I_3}
{ I_1, I_2, I_5}

Table 8 Candidate 3-itemsets

6. Database is scanned again to find count of each 3-itemsets using C_3 data.

C3

Itemset	Support Count
$\{I_1, I_2, I_3\}$	2
$\{I_1, I_2, I_5\}$	2

Table 9 Support Count of C3

7. Algorithm will now produce the L3 by comparing the candidate support count with minsupport.

L3

Itemset	Support Count
$\{I_1, I_2, I_3\}$	2
$\{I_1, I_2, I_5\}$	2

Table 10 Frequent 3-itemsets

8. A self join on L_3 is performed to produce candidate set of 4-itemsets, C_4. As a result, we will get $\{\{I_1, I_2, I_3, I_5\}\}$. Since the subset $\{I_2, I_3, I_5\}$ is not frequent, the 4-itemsets $\{\{I_1, I_2, I_3, I_5\}\}$ will be pruned resulting in empty C_4, causing the algorithm to terminate.

With the termination of the algorithm, the final large itemset obtained is the union of L1, L2 and L3 i.e.

$L = \{I_1, I_2, I_3, I_4, I_5, \{I_1, I_2\}, \{I_1, I_3\}, \{I_1, I_5\}, \{I_2, I_3\}, \{I_2, I_4\}, \{I_2, I_5\}, \{I_1, I_2, I_3\}, \{I_1, I_2, I_5\}\}$.

[AS94] introduced two more algorithms based on apriori i.e. AprioriTid and AprioriHybrid. In AprioriTid, the database is not scanned again for count support after the first scan for candidate 1-itemsets. This algorithm uses special encoding scheme to find the support count, hence reducing the database scans. AprioriHybrid on the other hand is a combination of Apriori and AprioriTid. AprioriHybrid addresses the memory limitation problem faced by AprioriTid because of keeping the auxiliary dataset in

memory. AprioriHybird switches between Apriori and AprioriTid depending on memory availability. We will not be addressing these algorithms here.

2.3. FP-Tree

Apriori-like algorithms [AIS [AIS93], AprioriTid [AS94], AprioriHybrid [AS94]] use candidate set generation and test approach. These Apriori like algorithms are costly, especially when large number of patterns exist [HPY00]. [HPY00] proposed a frequent-pattern tree (FP-tree) structure that compresses the database representing the frequent items and stores it in a prefix tree in descending order of their support. The FP-tree is then mined using the FP-growth mining method. Let us mine the database from Table 2 using the FP-growth method.

9. In the first scan frequent 1-itemsets are obtained with their support count. Keeping the same minsupport count of 2 as we used in previous section. The set obtained in first scan is stored in descending order of the support count denoted by *L*. L = [I_2:7, I_1:6, I_3:6, I_4:2, I_5:2].
10. After the first scan, the FP-tree construction process begins with the creation of the root node labeled 'null'.

- After this the database is scanned for the second time. All of the items in each transaction are processed in descending support count order and branch is created for each transaction. Let us create couple of branches from our example database.

- Transaction T100 has three items "I_1, I_2, I_5". Its L order would be "I_2, I_1, I_5". The tree branch created for this transaction is shown in Figure 2.

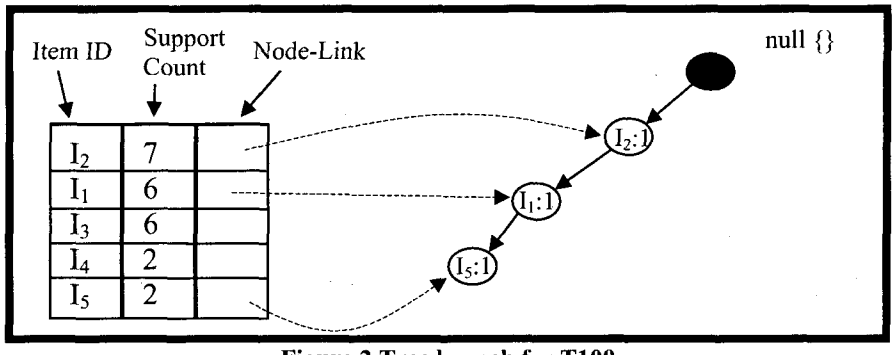


Figure 2 Tree branch for T100

- This completes the transformation of first transaction from the database to one of the branches of the FP-tree. Now let us work on the second transaction.
- Second transaction T200 consists of two items "I_2, I_4". Its L order would be "I_2, I_4". I_2 will be connected to the root and I_4 will be connected to the node I_2. The support count for I_2 will be incremented by 1 since this new branch shares the common prefix (I_2). Hence a new node I_4 will be created as a child note of I_2, as shown in Figure 3.

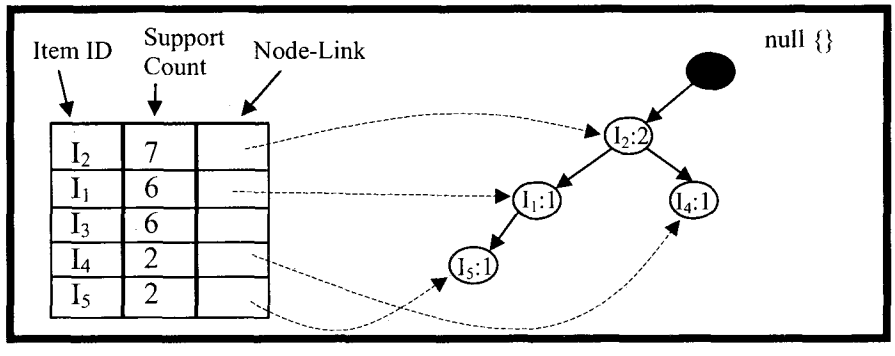


Figure 3 Tree branch for T200

- The algorithm will run for all the transactions in the database and the resulting tree looks as shown in Figure 4.

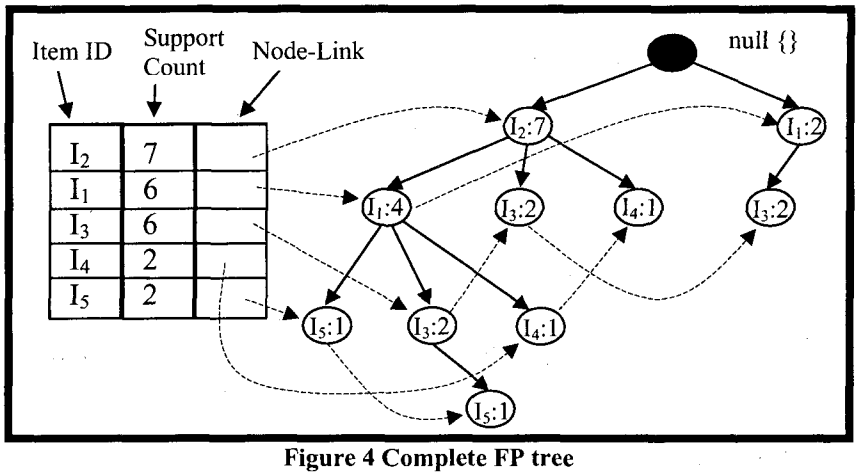


Figure 4 Complete FP tree

- Along with the FP-Tree data structure, this algorithm also maintains item header table. Each item from that table points to its occurrences in the FP-Tree using node-links.
- Once the FP-Tree construction is complete, mining process starts with the construction of the conditional pattern base from each frequent 1-itemset. Let us mine frequent patterns for I_3.
- I_3 has two branches in its conditional FP-tree as shown in Figure 5, which generates the following set of patterns: {I2 I3: 4, I1 I3:2, I2 I1 I3:2}.

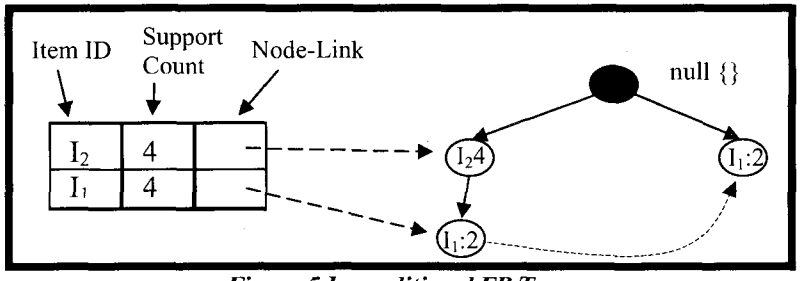


Figure 5 I_3 conditional FP Tree

- Complete mining of FP-tree is shown in table 11.

Item	*Conditional pattern base*	*Conditional FP-tree*	*Frequent patterns generated*
I_5	$[(I_2\ I_1{:}1), (I_2\ I_1\ I_3{:}1)]$	$(I_2{:}2, I_1{:}2)$	$I_2\ I_5{:}2, I_1\ I_5{:}2, I_2\ I_1\ I_5{:}2$
I_4	$[(I_2\ I_1{:}1),(I_2{:}1)]$	$(I_2{:}2)$	$I_2\ I_4{:}2$
I_3	$[(I_2\ I_1{:}2),(I_2{:}2),(I_1{:}2)]$	$(I_2{:}4,I_1{:}2),(I_1{:}2)$	$I_2\ I_3{:}4, I_1\ I_3{:}2, I_2\ I_1\ I_3{:}2$
I_1	$[(I_2{:}4)]$	$(I_2{:}4)$	$I_2\ I_1{:}4$

Table 11 Frequent Patterns

2.4. Sequential Pattern Mining Algorithms

Sequential pattern mining algorithms were first presented by [AS95]. These algorithms were based on the apriori algorithms presented by [AS94].

2.5. AprioriALL

AprioriAll is the first of two algorithms presented by [AS95] to mine sequential patterns. The essence of this algorithm is to find the *maximal* sequences. Each such maximal sequence represents a sequential pattern. A sequence is *maximal* if it is not *contained* in any other longer frequent sequence. This algorithm is based on the Apriori algorithm presented in [AS94] with the addition of maximal phase that prunes out all non-maximal sequences. AprioriAll algorithm works on a transformed database, shown in Table 15. Original customer-sequence database (Table 12) is transformed to a table where transactions are large itemset (Table 13) which is further replaced by litemset mapping, as shown in Table 14.

Cust ID	**Original Cust Seq**		Large Itemsets	Mapped To
1	[(30) (90)]		(30)	1

2	[(110 20)(30)(40 60 70)]		(40)	2
3	[(30 50 70)]		(70)	3
4	[(30)(40 70)(90)]		(40 70)	4
5	[(90)]		(90)	5

Table 12 Customer-sequence database

Table 13 Large Itemset mapping

Cust ID	Original Cust Seq	Transformed Seq	Mapped Seq
1	[(30) (90)]	[{(30)} {(90)}]	[{1} {5}]
2	[(110 20)(30)(40 60 70)]	[{(30)} {(40),(70),(40 70)}]	[{1} {2,3,4}]
3	[(30 50 70)]	[{(30), (70)}]	[{1,3}]
4	[(30)(40 70)(90)]	[{(30)}{(40),(70),(40 70)} {(90)}]	[{1} {2,3,4} {5}]
5	[(90)]	[{(90)}]	[{5}]

Table 14 Mapped sequences

Cust ID	Mapped Seq
1	[{1} {5}]
2	[{1} {2,3,4}]
3	[{1,3}]
4	[{1} {2,3,4} {5}]
5	[{5}]

Table 15 Transformed Database

The algorithm finds all the frequent patterns the same way as apriori algorithm does. It starts by scanning the database to get the support count of each frequent 1-itemset, Table 16. Assuming the user specified support to be 40% or 2 customer sequences, we will get the L_1 from C_1, shown in Table 17. We get the C_2 by doing a self join of L_1 using apriori-gen function as described in [AS94]. The algorithm will keep on generating the candidate itemsets and their corresponding large itemsets unless our candidate itemset becomes empty and we can not produce anymore large itemsets. Table 18 to Table 23 shows all of the candidate itemsets and large itemsets generated in sequence phase of AprioriAll algorithm. The maximal phase prunes out all of the non-maximal sequences from the large sequence sets to give the sequential patterns. Maximal phase uses the following algorithm to do this task [AS95]

For (k = n; k >1; k--) **do**

For each k-sequence s_k **do**

Delete from S all subsequences of s_k

S is the set of all large sequences generated in sequence phase. The maximal large sequences (sequential patterns) generated are shown in Table 24.

Seq	Supp
(1)	4
(2)	2
(3)	4
(4)	4
(5)	4

Table 16 C1

Seq	Supp
(1)	4
(2)	2
(3)	4
(4)	4
(5)	4

Table 17 L1

Seq	Supp
(1 2)	2
(1 3)	4
(1 4)	3
(1 5)	3
(2 3)	2
(2 4)	2
(2 5)	0
(3 4)	3
(3 5)	2
(4 5)	2

Table 19 C2

Seq	Supp
(1 2)	2
(1 3)	4
(1 4)	3
(1 5)	3
(2 3)	2
(2 4)	2
(3 4)	3
(3 5)	2
(4 5)	2

Table 18 L2

Seq	Supp
(1 2 3)	2
(1 2 4)	2
(1 3 4)	3
(1 3 5)	2
(1 4 5)	1
(2 3 4)	2
(3 4 5)	1

Table 23 C3

Seq	Supp
(1 2 3)	2
(1 2 4)	2
(1 3 4)	3
(1 3 5)	2
(2 3 4)	2

Table 22 L3

Seq	Supp
(1 2 3 4)	2

Table 20 C4

Seq	Supp
(1 2 3 4)	2

Table 21 L4

Seq	Supp
(1 2 3 4)	2
(1 3 5)	2
(4 5)	2

Table 24 Maximal Large Sequences

2.6. AprioriSome

AprioriSome is also presented in [AS95] along with AS. The main difference between AprioriAll and AprioriSome is that AprioriAll generates all large sequences, including non-maximal sequences. Whereas AprioriSome generates large sequences for some and avoids generating unnecessary non-maximal sequences for the others. AprioriSome has two phases: forward phase and a backward phase. In the forward phase, it generates all the large itemsets for certain length sequences. It uses a function *next* that gives the length of next sequence to be processed. In backward phase, algorithm counts the sequences for the lengths that it skipped in the forward phase along with deleting non-maximal sequences that were found in the forward phase. Taking the same database sample we used in Table 15 the AprioriSome forward phase will generate the C_1, L_1, C_2, L_2 and C_3. It will skip generating the L_3 and using the apriori-generate function will generate the C_4 from C_3. From C_4 it will get the L_4. Forward phase will halt here since C_5 turns out to be an empty set. Backward phase will start with deleting non-maximal

sequences from L_4. Since the only sequence in L_4 is (1 2 3 4), from Table 23, which is maximal, hence nothing will be deleted from L_4. Since we skipped the L_3 in forward phase, in backward we will delete all the subsequences of (1 2 3 4) that are in C_3. As a result, we are left with (1 3 5) and (3 4 5). Since (3 4 5) support count is only 1, it is also dropped. Going one step back, all sequences from L_2 are also deleted except (4 5). At the end, all of the sequences form the L_1 will be deleted since all of them are subsets of the thus far found maximal sequences. Final maximal large sequences are (1 2 3 4) (1 3 5) and (4 5).

2.7. WAP Tree

In the last section we saw apriori like algorithms for mining sequential patterns. WAP-tree or *Web Access Pattern Tree* was developed by [PHM+00] for mining the sequential patterns from web logs in non-apriori like fashion. WAP tree resembles more the FP-tree algorithm in the sense that WAP Tree also transforms the database into a compact tree like structure and then employees a mining algorithm on it. WAP-tree uses a WASD, *Web Access Sequence database.* WASD is obtained after the first scan of the database is done and the non-frequent parts of every sequence are discarded. Let us work on an example of web log sequence shown in Figure 6 as <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>

Figure 6 Web log sequence

These events are pre-processed in such a way that all access sequences for each user are grouped together to form a transaction database as shown in Table 25

TID	*WASD*
100	abdac
200	eaebcac
300	babfaec
400	afbacfc

Table 25 Web Access Sequence Database

After the transformation, dataset is scanned to get the frequency of each event. Assuming the minimum support of 75%, each sequence in WASD database is transformed into frequent subsequence as shown in Table 26. The support count for the events is a=4, b=4, c=4, d=1, e=2 and f=2. Since the support count for d, e and f is less that 75%, they are dropped from the web access sequences.

TID	*WASD*	*Frequent Subsequence*
100	abdac	abac
200	eaebcac	abcac
300	babfaec	babac
400	afbacfc	abacc

Table 26 WASD Frequent Subsequences

Construction of WAP tree starts the same way as of FP-tree. A header node table is created with frequent events from the frequent subsequences to facilitate the tree traversal. Lets us now look at the complete construction of the three in following steps.

- A virtual root is created.

- Each event from the sequence is inserted into the tree as a node with count 1 from Root if that node type does not exist in that path, but the count is incremented by 1 if that node type already exists.

- Taking first sequence *'abac'* from the Table 26, node with label 'a' will be inserted as a left child of the Root. Header node table is also updated and event 'a' in the header table is linked to this node. Since there is no immediate child of the root labeled 'a', we will assign 1 to this node 'a'. 'b' follows 'a' in this sequence and it will be made the let child of node labeled 'a'. After that we will insert 'a' as the right child of the node 'b'. At the end we will insert 'c' as the right child of the just created node 'a'. Counter for all these nodes will be set to 1. Final branch for sequence *'abac'* is shown in Figure 7

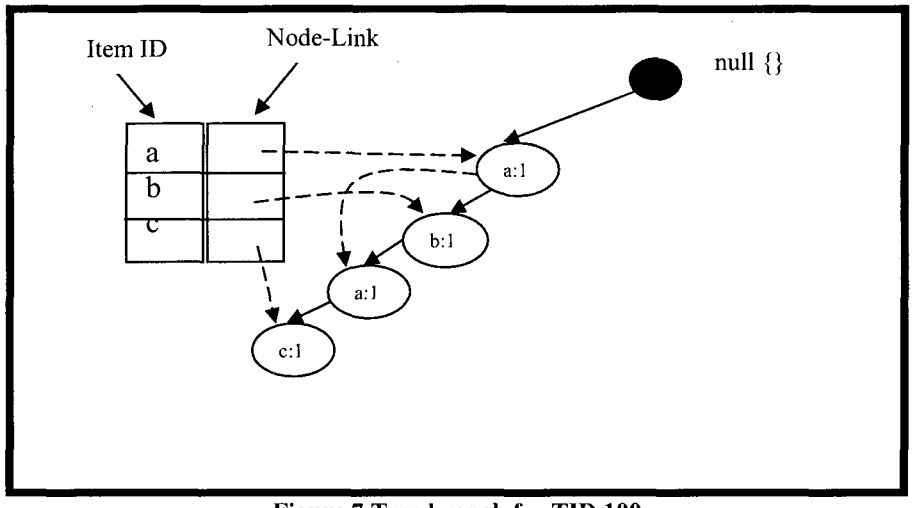


Figure 7 Tree branch for TID 100

- The second sequence 'abcac' is then processed. Since there is an immediate child of root with label 'a' exists, algorithm does not insert a new node with label 'a' as child instead it will increment the count of 'a' to 2. Counter for node with label 'b' will also be incremented by 1 since it also already exists. Since the next node in the existing path is 'a' which does not match the event 'c' in this sequence, hence a new node with c:1 will be inserted. Similarly a:1 will become the left

child of just inserted c:1 and c:1 will become the left child of the a:1, as shown in Figure 8.

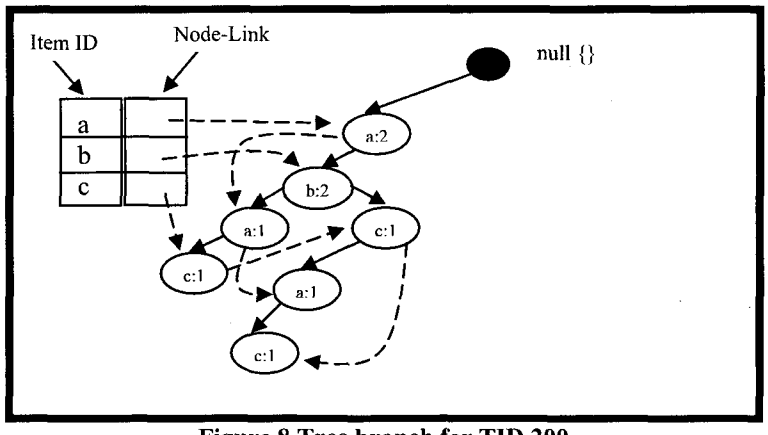


Figure 8 Tree branch for TID 200

- The algorithm will process the remaining two sequences the same way and the resulting WAP-tree will be similar to Figure 9.

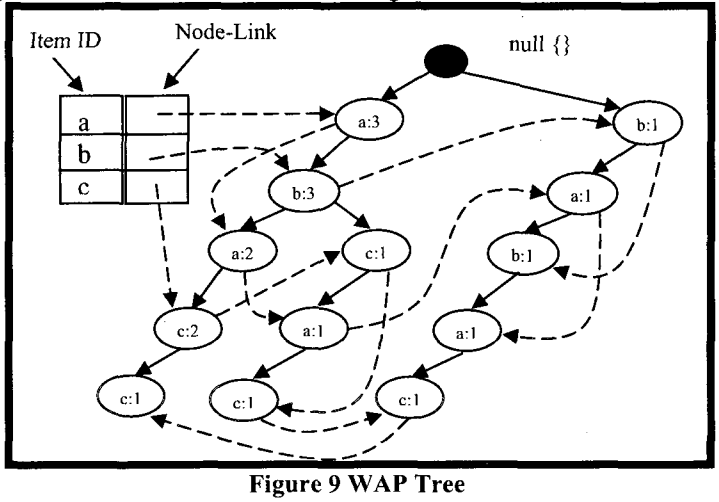


Figure 9 WAP Tree

Completion of the WAP-tree is followed by the WAP-mine algorithm that mines all frequent sequential patterns from the tree. Following steps are taken to complete the WAP-mine.

- Mining starts with the lowest entry in the header table which in our example is 'c' and all the conditional sequence base of c are discovered, i.e. aba:2, ab:1, abca:1, ab:-1, baba:1, abac:1, aba:-1. Those subsequences that have -1 will be deducted

from the list of conditional sequence base of 'c' because such sequences are prefix sequences of other sequences. For example, aba and ab have count -1 and you may notice that aba and ab are prefix sequences of conditional sequence aba. This deduction is done to avoid these sequences from contributing twice. Now for these events to qualify as a frequent conditional event, one event must have a count of at least 3. Getting the counts from the sequences above, we get a:4, b:4 and c:2. Since c:2 is less than the min support of 3, it will be discarded. After this elimination of 'c', the resulting conditional sequences based on c are aba:2, ab:1, aba:1, ab:-1, baba:1, aba:1.

- Using the above sequence, algorithm will now generate a conditional WAP tree, WAP-tree|c as shown in Figure 10.

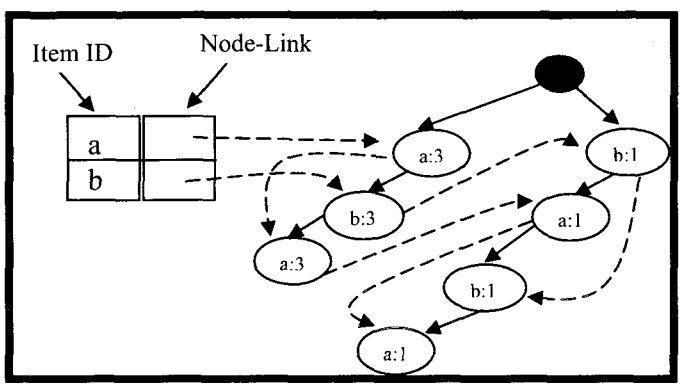


Figure 10 Conditional WAP-tree|c

- New conditional WAP tree is now ready to be mined, as shown in Figure 10.
- Next suffix subsequence bc is found as a:3, ba:1 and NULL. Since a is the only frequent event with count 3, 'b' will be discarded and a:4 becomes the frequent sequence base of 'bc'. The recursive re-construction of WAP tree based on bc ends here with c, bc and abc as the frequent sequences found so far. The recursion will continue with suffix path |c, |ac. Conditional sequence base for

suffix 'ac' computed from figure 10 is NULL, ab:3, b:1, bab:1, and b:-1. Using the just mentioned list, algorithm will build the conditional WAP-tree for base 'ac', as shown in Figure 11.

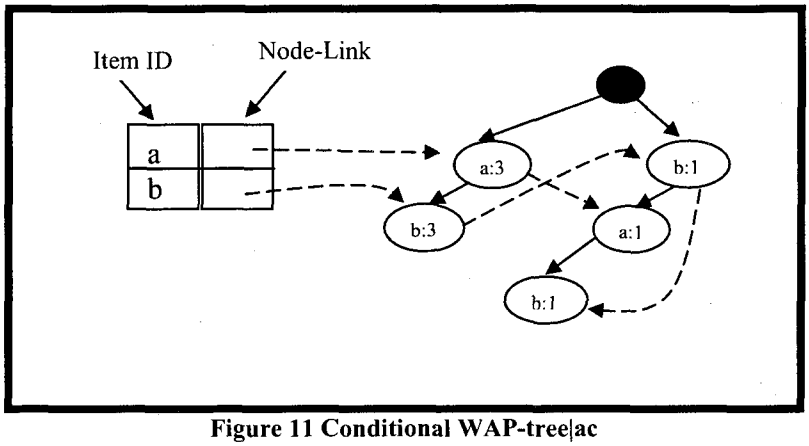


Figure 11 Conditional WAP-tree|ac

- The algorithm next finds the conditional sequence bases of bac as a:3 and ba:1. The only frequent sequence from here is a:4. Next the conditional WAP-tree|bac is built with a:4. From here, the algorithm will go back to complete the mining of suffix ac and starts mining for suffix aac. The only sequence we get from conditional sequence base aac is b:1 which is not frequent hence the conditional search for 'c' ends. The conditional search for events 'a' and 'b' is also done the same way as the mining for patterns with suffix 'c' is done.
- The complete frequent pattern set obtained at the end of WAP-mine algorithm is shown in figure11.

{c, aac, bac, abac, ac, abc, bc, ,b, ab, a, aa, ba, aba}

Figure 12 Complete frequent pattern set

Unlike Apriori-like algorithms that make multiple scans of the databases to mine sequential patterns, WAP-tree algorithm only scans the database twice and avoids

generation of candidate sets. But, at the same time, it introduces the construction of large recursive intermediate WAP-trees during mining process, hence introducing the problem of efficiently storing and managing the main memory for such intermediate WAP trees. In the next section we will see how PLWAP algorithm presented by [LE03], [LE05] and [ELL05] eliminates the construction of such intermediate WAP-trees and improves the performance of mining sequential patterns.

2.8. SPADE

In [Z00], the author proposed the SPADE (Sequential PAttern Discovery using Equivalent Class) algorithm. SPADE uses vertical format sequential pattern mining technique. In this technique an object is associated with each sequence in which it occurred along with the time stamp. Sequential pattern mining is implemented by growing the subsequences using apriori candidate generation. Bottleneck of SPADE is the generation of huge set of candidate sequences and the multiple scans of database in mining.

2.9. PrefixSpan

PrefixSpan was introduced by [PJ+01] and it is based on freeSpan. FreeSpan uses frequent items to recursively project sequences databases into a set of smaller projected databases and grow subsequence fragments in each projected database. The drawback of freeSpan is that the algorithm needs to keep the projected sequence in its original database without length reduction. PrefixSpan on the other hand is a prefix-based projection algorithm. It examines the prefix subsequence and projects their corresponding postfix subsequences. PrefixSpan algorithm does not generate any

candidate sequences and projected databases keeps on shrinking. Major cost of PrefixSpan is the construction of the project database and storing it in the memory. PrefixSpan also proposed two projection techniques 1) bi-level projection for reducing the number and sizes of projected database 2) Pseudo-projection that avoids physical copying of postfixes by using pointers to form projections.

2.10. PLWAP Tree

PLWAP or the Pre-order Linked Web Access Pattern Tree was introduced by [LE03], [LE05] and [ELL05]. It eliminates the need for recursively re-constructing the intermediate WAP-trees during mining. It employs binary position code assignment to the nodes that helps determine the suffix tree for any frequent pattern prefix and eliminates the need of constructing the intermediate trees. Rule 2.1 from [LE03] defines the assignment of binary code as

> *"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 of 2^{n-1} to the parent's code".*

PLWAP or the Pre-order Linked Web Access Pattern Tree was introduced by [LE03], [LE05] and [ELL05]. It eliminates the need for recursively re-constructing the intermediate WAP-trees during mining. It employs binary position code assignment to the nodes that helps determine the suffix tree for any frequent pattern prefix and eliminates the need of constructing the intermediate trees. Rule 2.1 from [LE03] defines the assignment of binary code as

> *"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 of 2^{n-1} to the parent's code".*

There are three main steps of PLWAP algorithm implementation, which are given below.

- **Step 1:** Frequent-1 events are obtained by scanning the access sequence database. All events that have support equal or greater than the minimum support are frequent. In the PLWAP-tree, each node stores node label, node count and node position code. The root of the tree is a special virtual node with an empty label and count 0.
- **Step 2:** Database is scanned for the second time to obtain the frequent sequences from each transaction. The non-frequent events in each sequence are deleted from the sequence, similar to WAP algorithm implementation. 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. Taking first sequence *'abac'* from Table 26, node with label 'a' will be inserted as a left child of the Root. Since there is no immediate child of the root labeled 'a', we will assign count 1 to this node 'a' and set its position code by applying Rule 2.1 above. Event 'b' follows 'a' in this sequence and it will be made the let child of node labeled 'a' with count set to '1' and position code set by again using rule 2.1. After that we will insert 'a' as the right child of the node 'b'. At the end we will insert 'c' as the right child of the just created node 'a'.

Once all of the sequences are inserted in the PLWAP-tree from table 26, the tree is traversed in pre-order fashion (by visiting the root first, the left subtree next and the right subtree finally), to create the frequent header node linkage. To assist node traversal during mining process, auxiliary node linkage structure is constructed. 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 for a PLWAP-tree, and the head of each event-node queue is registered.

The resulting PLWAP-tree is shown in figure 13.

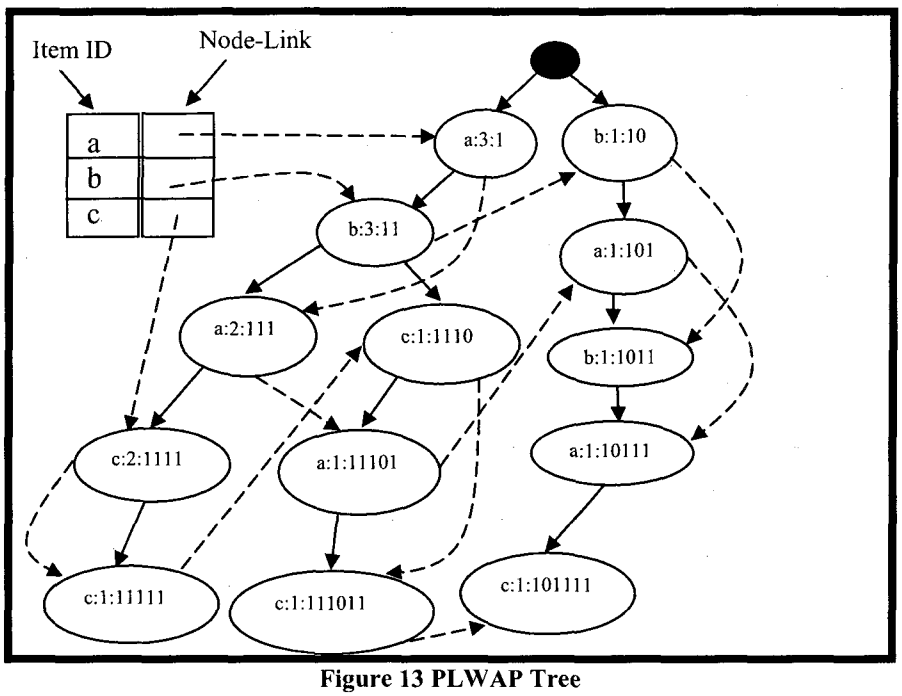


Figure 13 PLWAP Tree

Step 3: WAP-mine algorithm uses suffix subsequences to construct intermediate WAP-trees to find the frequent sequence. PLWAP on the other hand finds the prefix event first and then uses the suffix tree. Since PLWAP does not create intermediate WAP-trees, it uses property 2.1 [LE03] to determine quickly if the given node is an ancestor or descendant of the other node hence finding the suffix tree of a particular event without reconstructing the intermediate trees. The property 2.1 from [LE03] states

"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)."

Taking Figure 13 as an example with minimum support of 75%, let us find frequent access patterns.

- PLWAP Algorithm starts mining with the first element from the header linkage table. In our example it is 'a'. Following the 'a' link, the first occurrence of 'a' node in the two suffix trees of the root at a:3:1 and b:1:10 is mined. The first occurrence in both suffix trees is found at node a:3:1 and a:1:101. Since the sum of counts of both these nodes is greater than the minimum support, hence 'a' is considered as frequent 1-sequence.
- Next the algorithm will look at 2-sequence that starts with 'a'. The suffix trees of 1:3:1 and a:1:101 rooted at b:3;11 and b:1:1011 are mined. The first occurrences of 'a' in these suffix trees are found at nodes a:2:111, a:1:11101 and a:1:10111. Since the frequency count of these node is more than 3 hence 'a' is added to the last list of frequent sequence 'a' forming 'aa' frequent sequence.
- The algorithm will next mine the suffix trees of nodes mentioned in last step. The roots of these suffix trees c:2:1111, c:1:111011 and c:1:101111 will give 'c' frequent event to make 'aac' frequent sequence. The last suffix tree is c:1:11111 which is not frequent hence terminating the recursive search for 'a' and starts with the next event 'b' from the header linkage table. The algorithm backtracks and finds b:3:11 and b:1:1011 and generates 'b' frequent event giving 'ab' frequent sequence. The algorithm progresses and finds other frequent sequences with 'ab' as their prefix sequence i.e 'aba', 'abac' and 'abc'. The algorithm terminates here as no more frequent sequences are found and backtracks to find frequent sequences that have 'c' as prefix event. Algorithm finds frequent event 'c' from c:2;1111, c:1:111011 and c:1:101111 to give 'ac' as the frequent sequence. This completes finding all the frequent sequences that have 'a' as their prefix.

- The PLWAP algorithm then finds the frequent sequences starting with 'b' and 'c'. The complete set of frequent sequences found by PLWAP are {a,aa,aac,ab,aba,,abac,abc,ac,b,ba,bac,bc,c}.

2.11. PLWAP1 & PLWAP2

[WP08] introduced PLWAP1 and PLWAP2 algorithms that are based on WAP and PLWAP algorithms. In PLWAP1 algorithm implementation, a new header table is created in every recursive call during mining that links only those nodes that are under the new root set. With this approach they are avoiding redundant node checking. In PLWAP2 algorithm, no new header table is created but the algorithm filters out events that are not under the new roots. This filtering is achieved by traversing from the new roots and collection events to add into the new header table.

3. Position Coded Pre-Ordered linked WAP-Tree Long (PLWAPLong1 & PLWAPLong2)

3.1. Problem addressed

We have identified two problems that degrade the PLWAP algorithm performance.

Problem #1: In previous chapter we saw that PLWAP eliminates the need to generate intermediate conditional WAP trees by first assigning the position codes to each node and then identifying quickly if a node on a current suffix tree set belongs to a different subtree so that its count can contribute to the total support count of root set. PLWAP algorithm implementation represents binary position code of each node by storing its binary code in a linked list data structure. However, for very long sequences exceeding thirty two nodes, the number of bits an integer position code can hold, the PLWAP algorithm's performance begins to degrade because the linked list traversals slow down the algorithm both during tree construction and mining [ELL05]. This is because when the algorithm starts the mining process and needs to test the ancestor-descendant relationship of two nodes, it will first retrieve the complete position code of these nodes by following the linked lists associated with each one of them. If these nodes happen to be part of the tail of a very long (more than 32 items) sequence, the retrieval of position codes becomes slow because the linked list will need to make too many memory reads to traverse completely through the linked list.

Problem #2: Second problem seen in the PLWAP algorithm implementation is that during construction of the suffix trees the ancestor-descendant relationship check between the nodes from the root set and the nodes from the event queues performs many

unnecessary checks. This is because all events, with the same label, that are ancestor of event for which suffix tree is being explored, should not be tested for this relationship as they will never be counted in the suffix tree support. Similarly, those events, with the same labels, that are descendants of event for which suffix tree is being explored, should not be included in the support count of the suffix tree as well [LE03]. When dealing with long sequences where branches have hundreds of event nodes having repeated events, the algorithm will do many relationship checks just to ignore their support count. This support check affects the performance when we have very long sequences. From Figure-12 we can see that when algorithm starts the mining process with "root" node in the root set and event queue 'a', node "a:3:1" is the first node from the event queue 'a' found to be the first descendant of root and added to the new root set. Although all the descendants of this node "a:3:1" will be checked for ancestor-descendant relationship but their counts will not be added, hence taking up time and costing the performance especially when we have large sequences.

3.2. Proposed solution

To address the problems identified in previous section, we are proposition two new algorithms, i.e. PLWAPLong1 and PLWAPLong2. Both of these algorithms share same solution for problem #1. To address problem #2, PLWAPLong1 algorithm proposes the transformation of linked list based PLWAP tree into its equivalent array based representation and employing binary search to find the descendents during root set creation. On the other hand, PLWAPLong2 algorithm uses the same linked list base tree structure as used by PLWAP algorithm. Both of these new algorithms also share a new

technique called 'Last Descendant' to eliminate unwanted node comparisons during root set creation.

Solution for problem #1: To overcome the first problem, we are proposing a new position code numbering scheme. Our approach uses two new labels instead of one for each node i.e. 'startPosition' and 'endPosition' and assigns the numeric values to these labels during transformation of linked list tree into array based tree with pre-order traversal of the PLWAP-tree. Along with the assignment of new position code, we are also proposing the following rule that will be used during the mining process to determine the ancestor-descendant relationship of any two nodes.

Rule 1.0

"Given two nodes, n_1 and n_2, n_1 is ancestor of n_2 (or n_2 is descended of n_1) if n_1.startPosition < n_2.startPosition & if n_1.endPosition > n_2.endPosition".

In subsequent section we will discuss both new algorithms in detail and discuss how to address problem #2.

3.2.1. PLWAPLong1

Solution of problem #2: To address the second problem identified in section 3.1, we are proposing the following new processing:

1. Transform the PLWAP tree to its equal Array representation.
2. Maintain the position of the last descendant of each event.
3. Employ binary search to find the immediate descendant during root set creation.

Here is how rest of this chapter is organized. Section 3.2.1 details the new approach of transforming the linked list based PLWAP tree to its equivalent array based PLWAP-Long tree. Section 3.2.2 presents the details of maintaining 'Last Descendant'. Section

3.2.3 details the importance of using binary search to find immediate descendant. Section 3.2.4 presents the mining algorithm for PLWAP-Long with example. Section 3.2.5 outlines **PLWAPLong1, transformTree and buildDesc algorithms.**

3.2.1.1. Array Representation

The reason for transforming the linked list tree to its equivalent array representation is that with arrays we can jump from one node to the other known node in a O(1) time. In case of linked list based PLWAP-tree, memory references of parent, child and sibling are stored in nodes and memory seek is required in order to obtain the actual address. Another advantage of array representation is that we do not need to chain the same label events as we did in the linked list tree because during transformation event arrays are constructed using the pre-order traversal of the linked list tree and hence all events with same label are inserted in their respective event arrays in ascending startPosition. Once all of the events with same label are inserted in their respective event arrays, starting from index 0 and incrementing the index by 1 will explicitly chain these same label events. The header table is also represented using array of linkheader structure.

Note: To keep the transformed array based PLWAP-Long tree figures simple, all of the values a node holds are shown within <>. The order of these values is <event><occur><startPosition><endPosition><lastDesc>.

Taking the Figure-13 as example, let us transform the tree to its array representation using pseudo code of transformTree algorithm shown in Figure 21. A new method transformTree() is called that takes in root node of the linked list tree, NULL parent node and NULL leftChild node. transformTree method traverses through the linked list tree in a pre-order fashion. During this transformation the transformTree method also assigns

the startPosition and the endPosition. It starts with the event of the root node and looks up the header table array and finds the event.

Since it is the start of the transformation and the fact that root event has label -1, we won't find this event in the header table and hence create a new root node. startPosition '1' is assigned to the root node. This root node will point the transformed array represented tree. Following the algorithm from Figure-21, we will call the transform tree method with left child of the root (i.e. a:3:1), the newly created root and the NULL (since the newly created root has not yet assigned the left child). ***Note: The link header array is populated when the original linked list PLWAP tree is constructed. The event arrays are also dynamically created at this very moment and start of each array is linked to the event entry in the link header array. In this example from Figure-13, three event arrays will be created for event 'a' size 5, event 'b' size 2 and event 'c' with size 5.*** Algorithm will find the array index of this event from the link header array, i.e. 0, and also retrieve the insertPosition. InsertPosition gives the location where new node should be inserted within that event array. Since this is the first time event 'a' is seen, this node will be inserted at position 0, as shown in Figure 14.

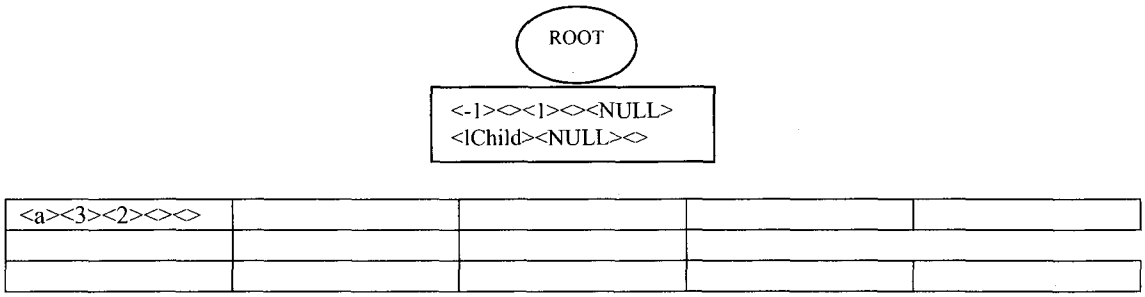
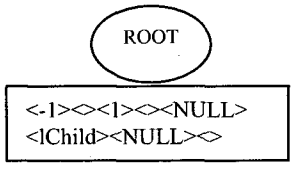


Figure 14 Transformed PLWAPLong-Tree with node a:3:1

Next we get the left child of this event which is b:3:11 and insert it into the event array 'b' pointed by the link header array index 1. Since this is the first time event b is recorded, the insertPosition is set to 0 and hence b:3:11 will be inserted at position 0 in the event array 'b' as seen in Figure 15. We will keep on traversing in the pre-order fashion until we get to the last event in leftmost branch of the original tree. The array values at that point will be as shown in Figure 16.

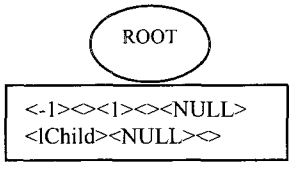


<a><3><2><><>				
<b><3><3><><>				

Link Header Array

Event =a Occur =5 insertPosition=1 link *	Event=b Occur =3 insertPosition=1 link *	Event=c Occur =5 insertPosition=0 link *

Figure 15 Transformed PLWAPLong-Tree with node b:3:11



<a><3><2><><>	<a><1><4><><>			
<b><3><3><><>				
<c><2><5><><>	<c><1><6><7><>			

Link Header Array

Event =a Occur =5 insertPosition=2 link *	Event=b Occur =3 insertPosition=1 link *	Event=c Occur =5 insertPosition=2 link *

Figure 16 Transformed PLWAPLong-Tree with node c:1:11111

Since the event c:1:11111 has a NULL left child, we will create a new node according to point 1.3 of Figure-21 and then set the end position according to the point 1.37 of Figure -21. The algorithm will then start going backwards till it finds any right sibling while assigning the endPosition to each of the nodes. The first node with right sibling is found at node a:2:111. The algorithm will start traversing the right sibling of a:2:111. It will create a new node and then set the right sibling of the leftChild to the new created node at this point. The transformed tree up to this point is shown in Figure 17.

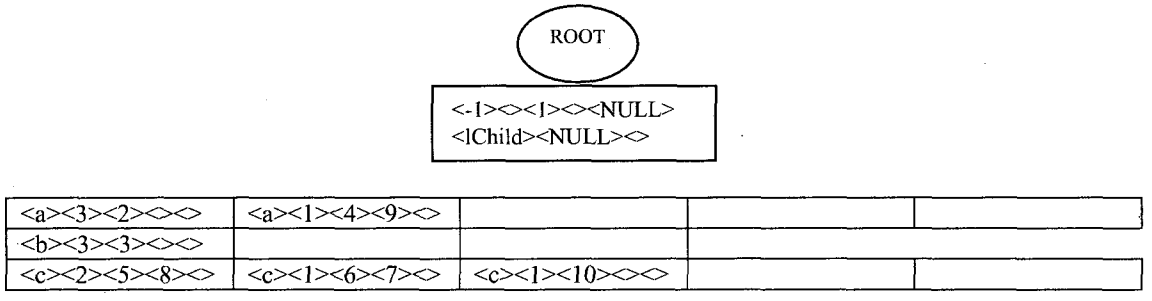
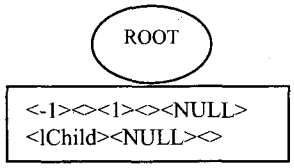


Figure 17 Transformed PLWAPLong Tree with c:1:1110

Continuing with the pre-order traversal from node c:1:1110 we will get to the last node in this branch i.e. c:1:111011 at which point algorithm will start backward traversal. Let us pause when the algorithm comes back to the node b:3:11 at which point it sets the endPosition of node b:3:11. The tree up to now is shown in Figure 18.



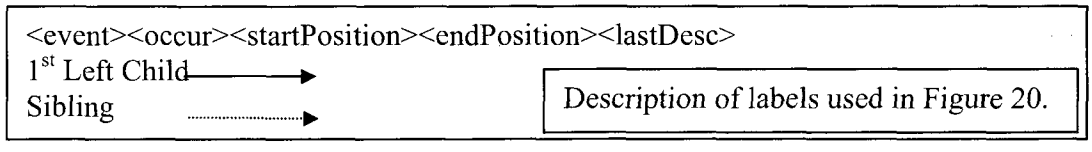
<a><3><2><><>	<a><1><4><9><>	<a><1><11><14><>		
<b><3><3><16><>				
<c><2><5><8><>	<c><1><6><7><>	<c><1><10><15><>	<c><1><12><13><>	

Link Header Array

Event =a Occur =5 insertPosition=3 link *	Event=b Occur =3 insertPosition=1 link *	Event=c Occur =5 insertPosition=4 link *

Figure 18 Transformed PLWAPLong Tree

Since there is no right sibling of node b:3:11 the recursive call to transformTree() will go back to node a:3:1 and assign the. Since the node a:3:1 has a right sibling, the algorithm will make a call to the transformTree(). The algorithm will continue to traverse the branch downwards starting at node b:1:10 and ending at node c:1:101111. Since the left child of c:1:101111 is NULL, the algorithm will start backward traversal until it reaches back the node, at which point root node will be assigned the endPosition. The complete transformed tree is shown in Figure 19 along with the pointers to the left child and the 2nd left child.



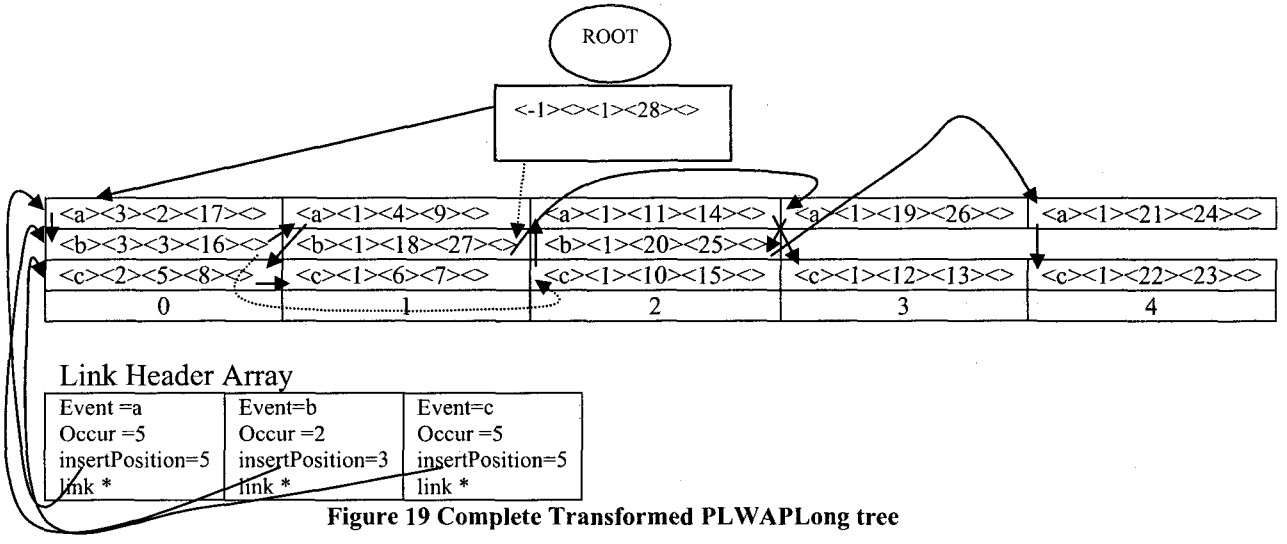


Figure 19 Complete Transformed PLWAPLong tree

3.2.2. PLWAPLong2

PLWAPLong2 algorithm differs from PLWAPLong1 in a sense that it does not transform the PLWAP tree to array based representation. Instead, it uses the same linked list based tree structure that is used by PLWAP algorithm implementation. PLWAPLong2 algorithm implementation is same as PLWAP algorithm with the addition of new position code scheme and maintaining the last descendant.

There are four main steps of PLWAPLong2 algorithm implementation, which are given below.

- **Step 1:** Frequent-1 events are obtained by scanning the access sequence database. All events that have support equal or greater than the minimum support are frequent. Each node in the tree stores following items

```
{       event; /*event name of the node*/
        occur; /*occurrence for the node*/
        startPosition; /*start position of the node*/
        endPosition; /*end position of the node*/
        CountSon;/*the sum of occurrence of sons*/
        *nextLink; /*the linkage to next node with same event name*/
```

***lastDesc; /*pointer to its last descendant*/**
***lSon;/*the pointer to its left Son*/**
***rSibling;/*the pointer to its right sibling*/**
***parent;/*the pointer to its parent*/**

}

. The root of the tree is a special virtual node with an empty label and count 0.

- **Step 2:** Database is scanned for the second time to obtain the frequent sequences from each transaction. The non-frequent events in each sequence are deleted from the sequence, similar to PLWAP algorithm implementation. PLWAPLong2 algorithm also builds a prefix tree data structure, same as PLWAP tree, by inserting the frequent sequence of each transaction in the tree the same way the ***PLWAP*** algorithm would insert them. The insertion of frequent subsequence is started from the root of the tree. Taking first sequence *'abac'* from Table 26, node with label 'a' will be inserted as a left child of the Root. Since there is no immediate child of the root labeled 'a', we will assign count 1 to this node 'a'. Event 'b' follows 'a' in this sequence and it will be made the left child of node labeled 'a' with count set to '1'. After that we will insert 'a' as the right child of the node 'b'. At the end we will insert 'c' as the right child of the just created node 'a'.

Once all of the sequences are inserted in the PLWAP-tree from table 26, the tree is traversed in pre-order fashion (by visiting the root first, the left subtree next and the right subtree finally), to create the frequent header node linkage. To assist node traversal during mining process, auxiliary node linkage structure is constructed. 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 ei is also called ei –queue. There is one header table for tree, and the head of each event-node queue is registered. After this another pre-order traversal is used to assign startPosition and endPosition for all of the nodes.

Let us now go through an example and see how new position codes are assigned during transformation. Since the PLWAPLong2 tree will be the same as the PLWAP tree with the exception that it will not have the binary position codes, we will assign the new position codes, i.e. startPosition and endPosition, to the PLWAP-tree of Figure 13. Taking figure 13, algorithm will start in a pre-order fashion i.e. starting from the root and assigning it the value of '1' to the 'left' label. It will then go to the left child of the root and assign the value 2 to the left label of node a:3. It will keep on going and assign the value 6 to the 'left' label of c:1. Since there is no more left or right child of c:1, algorithm will assign the value 7 to its 'right' label and traverse back by assigning value 8 to the 'right' label of c:2 and 9 to the 'right' label of a:1. Here the parent of a:1 has a right child i.e. c:1 and hence the traversal will continue to assign 'left' label values in this new branch. Once the pre-order traversal comes back to the b:2, it will get value 16 for its 'right' label. The traversal will continue and it terminates when it reaches back the root where value 28 will be assigned to the 'right' label of the root. The complete numeric position coding for 'startPosition' and 'endPosition' labels is shown in Figure 26.

- **Step 3:** After assigning the start and end positions, the tree is traversed once again in a pre-order fashion to create the last descendants for each node. PLWAPLong2 algorithm also uses the same algorithm used by PLWAPLong1 to create these last descendants by employing algorithm shown in Figure 27.
- **Step 4:** In step 4, PLWAPLong tree is mined.

3.2.3. Maintaining Last Descendant

A new feature added with the array implementation of the PLWAP tree is to maintain the index of the last descendant of the same event in its subtree for each node. By adopting this technique we can eliminate browsing and checking ancestor-descendant relationship of many unwanted nodes. This approach is very useful and efficient in mining long sequences, although our test results show that this technique is also effective for short sequences. Since it's not possible to run example of long sequences on paper, we will create arbitrary PLWAP tree and show how this technique benefits in creating root sets.

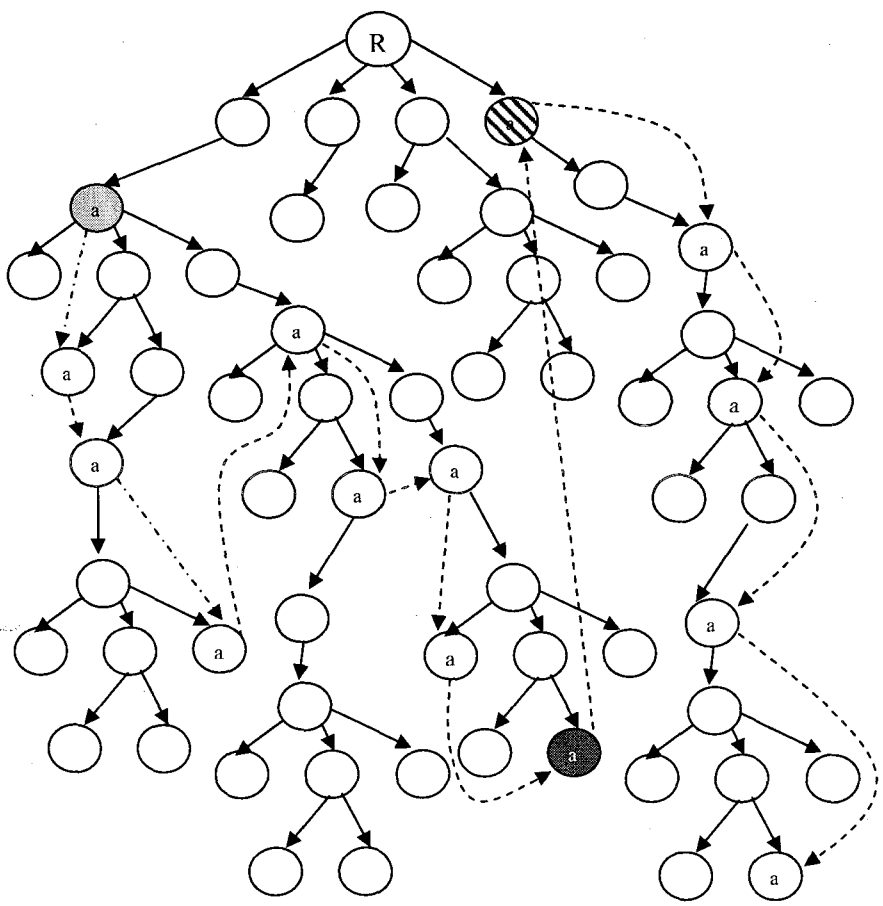


Figure 20 Example PLWAP Tree

From figure 20, assume that we are finding the first occurrence of event labeled 'a' that is descendant of root. The first such event is colored light grey happened to be the first event in 'a' linked header as well. Original PLWAP algorithm after recording this node in the rootSet will keep on traversing through the event 'a' link and check for the ancestor-descendant relationship of all of the event 'a' nodes that are descendant of the grey colored node. Count for these nodes will not be added to the total count and neither will they be added to the root set. Hence, PLWAP algorithm spent time in doing comparison of events that we should have avoided. The only way we could avoid this is to keep track of the last descendant for each event. Going back to Figure 22, the last descendant of light grey colored event 'a' is the node that is colored dark grey. In between there are seven event 'a' nodes that are also checked for the ancestor-descendant relationship before the node with event 'a' represented with black stripes is checked and found to be the next descendant of root that should be added to the root Set. To maintain the last descendant, we are introducing a new method buildDesc() that is called as soon as the transformed tree is ready. In the example above when the code had found that grey colored event 'a' is added to the root Set, it would have jumped directly to the last descendant +1, ignoring all of the descendant nodes and hence saving time from doing unnecessary checks. Let us go through the array represented PLWAPLong1 tree shown in Figure 20 and build the last descendants for event a. ***Please note that in example below, wherever we use notations like e_j or e_{j+1}, we mean event at index j or event at index j+1.***

Starting with index j=0 for event 'a' array and using point 1.1.2 from Figure 25, we find out that event at index j i.e. event 'a' with startPosition=2 and endPosition=17 is ancestor

of event at index j + 1 i.e. event 'a' with startPosition=4 and endPosition=9. We will push event at index j to the stack.

Stack
<a><3><2><17><>

J
1

Next, we will test the ancestor-descendant relationship between j=1 and j +1. In this case, event at index j=1 has startPosition=4 and endPosition=9 and event at index j+1 has startPosition=11 and endPosition=14. It turns out that e_j is not ancestor of e_{j+1}. Using point 1.1.3 from Figure 25 we will set the lastDesc of ej to the value of j, which in this case is 1. Updated event 'a' array as shown below with lastDesc updated for node at index = 1.

Event 'a' array

<a><3><2><17><>	<a><1><4><9><1>	<a><1><11><14><>	<a><1><19><26><>	<a><1><21><24><>
0	1	2	3	4

Updated stack and index j shown below

Stack
<a><3><2><17><>

J
2

Next we will test events at j=2 and j+1 and find out that e_j is not ancestor of e_{j+1}. Again using point 1.1.3 from Figure 25, algorithm will set the lastDesc of ej to the value of j. In this case, value of j=2 hence updating the lastDesc of event at index = 2 is shown below.

Event 'a' array

<a><3><2><17><>	<a><1><4><9><1>	<a><1><11><14><2>	<a><1><19><26><>	<a><1><21><24><>
0	1	2	3	4

From point 1.1.3.1 from Figure 25 the stack is not empty so we will test if stack.front is not ancestor of e_{j+1}. Stack has only one event with startPosition=2 and endPosition=17 and it turns out that this event is not an ancestor of e_{j+1} hence algorithm will pop the stack and set the lastDesc of popped event to j, which in this case is 2. The updated event 'a' array looks like this

Event 'a' array

<a><3><2><17><2>	<a><1><4><9><1>	<a><1><11><14><2>	<a><1><19><26><>	<a><1><21><24><>
0	1	2	3	4

Updated stack and j

Stack

J
3

Next we will test for j=3 and j+1. Event e_j has startPosition=19 and endPosition=26 and e_j+1 has startPosition=21 and endPosition=24. Event e_j turns out to be the ancestor of e_j+1 and hence will be pushed to the stack. Updated stack and j looks like

Stack
<a><1><19><26><>

J
4

Now since j=4 and j+1 is greater than the event 'a' array size, following point 1.1.1 from Figure 25 we set the e_j.lastDesc to 4. Since stack is not empty, algorithm will assign value of j as the lastDesc to all of the items still in the stack. In our example only item in the stack is event with startPosition=19 and endPosition=26. The updated event 'a' array is shown below.

Event 'a' array

<a><3><2><17><2>	<a><1><4><9><1>	<a><1><11><14><2>	<a><1><19><26><4>	<a><1><21><24><4>
0	1	2	3	4

The algorithm here finishes assigning lastDesc for all of the events in event 'a' array. It will perform the same algorithm on the remaining two event arrays i.e. event 'b' and event 'c' arrays and the resulting arrays will look like as shown below

<a><3><2><17><2>	<a><1><4><9><1>	<a><1><11><14><2>	<a><1><19><26><4>	<a><1><21><24><4>
<b><3><3><16><0>	<b><1><18><27><2>	<b><1><20><25><2>		
<c><2><5><8><1>	<c><1><6><7><1>	<c><1><10><15><3>	<c><1><12><13><3>	<c><1><22><23><4>
0	1	2	3	4

3.2.4. Mining Process – PLWAPLong1

The PLWAPLong1 is the version of the algorithm, which transforms the linked list tree to its array representative before mining. The main logic of mining process is the same as that of the PLWAP algorithm. PLWAPLong-Mine algorithm uses last descendant and binary search to speed up the mining process for long sequences. The other main difference between the PLWAP mine and PLWAP-Long mine process is how both algorithms build the suffix trees. Let us see how these two algorithms differ in that respect. PLWAP algorithm implementation when starts the mining process and finds first occurrence of event from the event queue, the subtrees of all those first occurred event become the roots of the suffix trees of these events. Considering Figure 12 as an example, we see that when PLWAP algorithm starts the mining process, it starts with finding the first occurrence of event 'a'. First occurrences of these events are found at node a:3:1 and a:1:101. The suffix trees of these two nodes will then be rooted at b:3:11 and b:1:1011. These two nodes will become the rootset and passed to the next round of mining. In the next round of mining, algorithm will first attempt to find pattern 'aa'. In order to find this pattern PLWAP mine algorithm will start from the beginning of event

'a' queue and see if any event in that queue is descendent of rootSet roots b:3:11 and b:1:1011.

On the other hand, when PLWAP-Long1 algorithm starts mining the same Figure 13, it will add the first occurrence events of event 'a' in the rootSet, i.e. a:3:1 and a:1:101, instead of adding roots of their suffix trees. Advantage of this approach is that rootSet will always have events of same label. This will help when the mining process attempts to find repetitive events (e.g. aa, aaa, acbb, abaa, etc). When rootSet a:3:1 and a:1:101 is passed to the next round of mining to find frequent pattern 'aa', PLWAP-Long1 mining algorithm implementation will test, in the event 'a' array, the very next event after event 'a:3:1' to see if it is descendant of it or not. Here, notice the advantage of this new approach. PLWAP-Long1 mining algorithm already knew the last occurrence of frequent event 'a', i.e. a:3:1, and hence did not need to start searching from the start of the event 'a' array for the first occurrence of event 'a' that is descendant of a:3:1.

The algorithm PLWAPLong1-Mine is shown in Figure 28. Let us run through the example of PLWAP-Long tree shown in Figure 20 and updated event array shown in Figure 24. First time root node will be in the root set R. Following the link header array, event 'a' array is the first one explored to find the descendants of root. Binary search function is called with iteration = 0 (since it is the start of the event 'a' array exploration), last = size of event 'a' -1 and key is 1 (root's start position is 1). Binary search will return the index 0 of event 'a' array and this event is the descendant of root, hence its count is added to C and this event is added to the R' Next iteration is set to the last descendant +1 of the just found event and binary search is called again. This time binary search will return the index 3. The event at index 3 is also the descendant of root and

hence its count is also added to C and event itself added to R'. Next iteration is set to the last descendant + 1 of the just added event. This time, the value of iteration is 5 which is greater than the value of *last*. The algorithm will check if the count is greater than the minimum support, which turns out to be true and hence event 'a' is added to the F' and outputted. For clarity of representation, let us color the linked list PLWAP tree as we continue with the mining process. The PLWAP tree at this moment is shown in figure 27. Next algorithm will try to find the descendants of event <a:3:2:17:2> and <a:1:19:26:4> in event 'a' array in hope of fining frequent event aa. The algorithm will set the variable *last* to the lastDesc of root event <a:3:2:17:2> and calls the binary search using point 2.1.1 of Figure 26. The first descendant of this event is found at index 1 i.e. <a><1><4><9><1> and added to R' and count to C. It again sets the iteration to the last descendant value +1 of just found descendant. Next descendant of <a:3:2:17:2> is found at index = 2 and its count added to C and event itself added to R'. At this point the iteration value is set to the last descendant +1, which turns out to be not descendant of <a:3:2:17:2> hence causing the next root node to be retrieved from the root set R. Algorithm will continue with point 2.1.1 of Figure 28 and in next binary search will return index 4. Event at this index is <a><1><21><24><4> and it is the immediate descendant of the root <a:1:19:26:4>. This event is added to R' and its count added to C. *iteration* is set to the last descendant +1 of the just found event i.e., 5. Next, binary search will run out of bounds of event 'a' array. At this point, algorithm will check if the count is greater than the minimum support. In this case it is, hence 'a' is appended to F' which already contains 'a'. Hence our new frequent pattern is aa. The updated PLWAP tree is shown in Figure 22. Algorithm PLWAPLong1-Mine will continue mining rest of

the event arrays using the algorithm presented in Figure 28. The final set of frequent patterns using minimum support of 75% is {a,aa,aac,ab,aba,,abac,abc,ac,b,ba,bac,bc,c}.

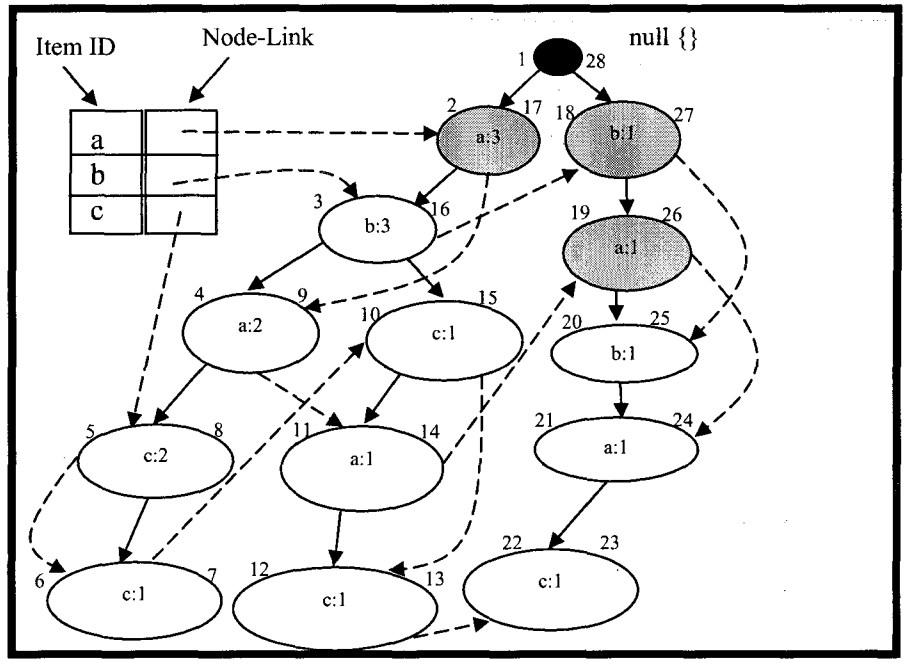


Figure 21 PLWAPLong Mine with root set a:3 and a:1

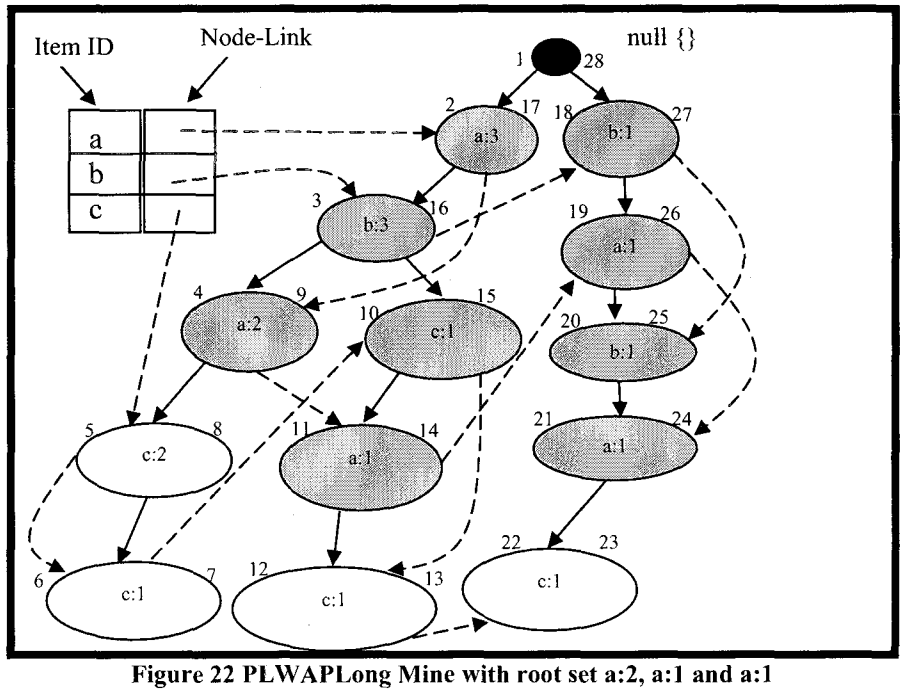


Figure 22 PLWAPLong Mine with root set a:2, a:1 and a:1

3.2.5. Mining Process – PLWAPLong2

The PLWAPLong2 is the version of the proposed algorithm, which is based on the linked list tree without transformation to an array. Formal definition of PLWAPLong2-Mine algorithm is given in Figure 29. Taking Figure 13 as an example with minimum support of 75%, let us find frequent access patterns.

- PLWAPLong2 Algorithm starts mining with the first element from the header linkage table. In our example it is 'a'. ***Please note that we will reference node in our example below by starting with node label followed by its occurrence count, followed by its start and end positions***. Following the 'a' link, the first occurrence of 'a' node in the two suffix trees of the root at a:3:2:17 and b:1:18:27 is mined. The first occurrence in both suffix trees is found at node a:3:2:17 and a:1:19:26. *During the mining process once we find that a:3:2:17 is descendant of the root and should be added to the new root set, we discard checking all of the descendants of node a:3:2:17 in its subtree with label 'a' because none of them will be added to the root set during this iteration. We accomplish this by jumping to the last descendant of node a:3:2:17.* The last descendant of node a:3:2:17 is a:1:11:14. The sum of counts of both these nodes, i.e. a:3:2:17 and a:1:19:26, is greater than the minimum support, hence 'a' is considered as frequent 1-sequence.
- Next the algorithm will look at 2-sequence that starts with 'a'. The suffix trees of a:3:2:17 and a:1:19:26 rooted at b:3:3:16 and b:1:20:25 are mined. The first

occurrences of 'a' in these suffix trees are found at nodes a:2:4:9, a:1:11:14 and a:1:21:24. Since the frequency count of these node is more than 3 hence 'a' is added to the last list of frequent sequence 'a' forming 'aa' frequent sequence.

- The algorithm will next mine the suffix trees of nodes mentioned in last step. The roots of these suffix trees c:2:5:8, c:1:12:13 and c:1:22:23 will give 'c' frequent event to make 'aac' frequent sequence. The last suffix tree is c:1:6:7 which is not frequent hence terminating the recursive search for 'a' and starts with the next event 'b' from the header linkage table. The algorithm backtracks and finds b:3:3:16 and b:1:20:25 and generates 'b' frequent event giving 'ab' frequent sequence. The algorithm progresses and finds other frequent sequences with 'ab' as their prefix sequence i.e. 'aba', 'abac' and 'abc'. The algorithm terminates here as no more frequent sequences are found and backtracks to find frequent sequences that have 'c' as prefix event. Algorithm finds frequent event 'c' from c:2;5:8, c:1:12:13 and c:1:22:23 to give 'ac' as the frequent sequence. This completes finding all the frequent sequences that have 'a' as their prefix.

The PLWAPLong2 algorithm then finds the frequent sequences starting with 'b' and 'c'. The complete set of frequent sequences found by PLWAPLong2 are {a,aa,aac,ab,aba,,abac,abc,ac,b,ba,bac,bc,c}.

3.3 Formal Definitions of PLWAPLong1 and PLWAPLong2 Algorithms

This section includes formal definitions of all of the new algorithms proposed, namely

PLWAPLong1 and PLWAPLong2, transformTree, buildDesc and PLWAPLong1-Mine and PLWAPLong2-Mine Methods. Below we will give definitions of the key terms used in these algorithms as well.

Tree: A data structure of interconnected nodes whose access starts at its root.

Node: A node of a tree could be a leaf of interior node. A leaf is an item with no child. Interior node has one or more child nodes and it becomes parents of these child nodes.

Suffix tree: Branches from node e_i to the leaf node represent suffix sequence and these suffix branches of e_i are called suffix tree of e_i.

Last Descendant:

Every node, except leaf, in a tree has at least once child node and hence has at least one suffix tree. The root of such a suffix tree will become ancestor of all of the nodes in its suffix tree. If there are more than once suffix trees for a given node, the node with same label as that of the root node in the right most suffix tree will become the last descendant of the root of this suffix tree. For example, the node a:3:2:17 in Figure 26 is a root of suffix tree rooted at b:3:3:16. The last descendant for this root node will be the farthest most node with same label in its suffix tree, i.e. a:1:11:14. Complete tree with last descendants is shown in Figure 27-1.

Input: Web Access Sequence Database (WASD), Minimum Support λ
Output: Complete set of frequent patterns F_i
Begin:

1. Scan WASD once, find all frequent f_1 events.
2. Scan WASD again, build PLWAP tree with seq list without position code with preorders F_1 header linkage
3. Create link header data summary array with event label, occurrence, insert header event link position, link to the frequent first event occurrence on the header link
4. Transform the PLWAP tree into Array representation by calling **transformTree(Root-P, Root-PL, LChild-PL)** as
    a. Traverse PLWAP tree pre-order fashion to generate the array as; Event, Occurrence, Start Position, End Position, Parent Link, Left Child, Right Sibling, Last Descendant index.
5. Build Last Descendant, by calling **buildDesc(Root-PL, F1 array),** of each event node using its suffix trees and using both the array representation of the PLWAP-Long tree and the header linkage
6. Mine the DB using both the header linkage array and the event array using binary search to quickly construct the suffix tree by calling **PLWAPLong-Mine(Root-PL,Frequent m-sequence F).**

End //PLWAPLong1()

Figure 23 Algorithm PLWAPLong1()

Input: Web Access Sequence Database (WASD), Minimum Support λ
Output: Complete set of frequent patterns F_i
Begin:

1. Scan WASD once, find all frequent f_1 events.
2. Scan WASD again, build PLWAP tree with seq list with new F_1 header linkage and position code
3. Build Last Descendant, by calling **buildDesc(Root-PL, F1 array),** of each event node using its suffix trees and using both the PLWAP-Long tree and the header linkage
4. Mine the DB using the header linkage table by calling **PLWAPLong2-Mine(Root-PL,Frequent m-sequence F).**

End //PLWAPLong2()

Figure 24 Algorithm PLWAPLong2()

Algorithm transformTree(): Formal definition of transformTree method is presented in figure ?. Running example of this algorithm is presented in section 3.2.1.

Input: start-node , parent node, leftChild of the parent
(start node here is the node from the linked list tree, at the start of the program root node will be passed)
(parent node is the equivalent of the start node in the array represented tree. At the start of the method call, this will be NULL)
(leftChild of the parent is the left child of 2nd parameter. At the start of the method call, this will be NULL. It will also be NULL when traversal of tree continues to browse the left child of the branch. It will not be NULL when traversal jumps to the sibling of the current node)
Output: A complete tree presentation using arrays with numeric position code.
Intermediate Variables: insertAt (position to insert the next event in the event array)
positionNumber (position number given to the start and end position labels of newly created nodes in the array represented tree)
Begin:
1. Find the event of the start-node in the link header array.
   1.1. **If** event not found, meaning start-node is the root node,
      1.1.1.Create a new root to point to the array represented tree.
   1.2. **If** start-node.lefftChild is != NULL */* i.e not a leaf */*
      1.2.1.**If** we have created the root **Then**
         1.2.1.1.assign startPosition = positionNumber++
         1.2.1.2.Call algorithm transformTree with start-node.leftChild, node created in step 1.1.1 and NULL LeftChild.
         1.2.1.3.assign endPosition = positionNumber++
      1.2.2.**ELSE** /*If we did not create a root node*/
         1.2.3. Set insertAt = Position of the array where next event should be inserted
         1.2.4. Create a new node, copying values from the start node.
         1.2.5. If leftChild != NULL link the newly created node to its left sibling.
         1.2.6. else Link to the parent if traversing in the left subtree
         1.2.7. Call transformTree with start-node.leftChild, parent of node created in step 1.2.4 and NULL pointer.
         1.2.8. assign the endPosition = positionNumber++
   1.3. **If** event found and leftChild is == NULL */*i.e leaf node */*
      1.3.1.If root was created, assign positionNumber++ to the startPosition and endPosition of the newly created root node.
      1.3.2.If we did not create a root node Then
      1.3.3.Set insertAt = Position of the array where next event should be inserted
      1.3.4.Create a new node, copying values from the start node.
      1.3.5.If leftChild != NULL link the newly created node to its left sibling.
      1.3.6.else Link to the parent if traversing in the left subtree
      1.3.7.assign the endPosition = positionNumber++
   1.4. **If** start->rSibling != NULL
      1.4.1.Call transformTree with start-node.rSibling, parent of the node created in either step 1.2.4 or step 1.3.4 and node created in either step 1.2.4 or 1.3.4.
End

Figure 25 Algorithm transformTree()

Let us now go through an example and see how new position codes are assigned during transformation. To keep the example simple, we will assign the new position codes to the PLWAP-tree instead of the array based PLWAP-Long tree. Taking figure 13, algorithm will start in a pre-order fashion i.e. starting from the root and assigning it the value of '1' to the 'left' label. It will then go to the left child of the root and assign the value 2 to the left label of node a:3. It will keep on going and assign the value 6 to the 'left' label of c:1. Since there is no more left or right child of c:1, algorithm will assign the value 7 to its 'right' label and traverse back by assigning value 8 to the 'right' label of c:2 and 9 to the 'right' label of a:1. Here the parent of a:1 has a right child i.e. c:1 and hence the traversal will continue to assign 'left' label values in this new branch. Once the pre-order traversal comes back to the b:2, it will get value 16 for its 'right' label. The traversal will continue and it terminates when it reaches back the root where value 28 will be assigned to the 'right' label of the root. The complete numeric position coding for 'startPosition' and 'endPosition' labels is shown in Figure 25.

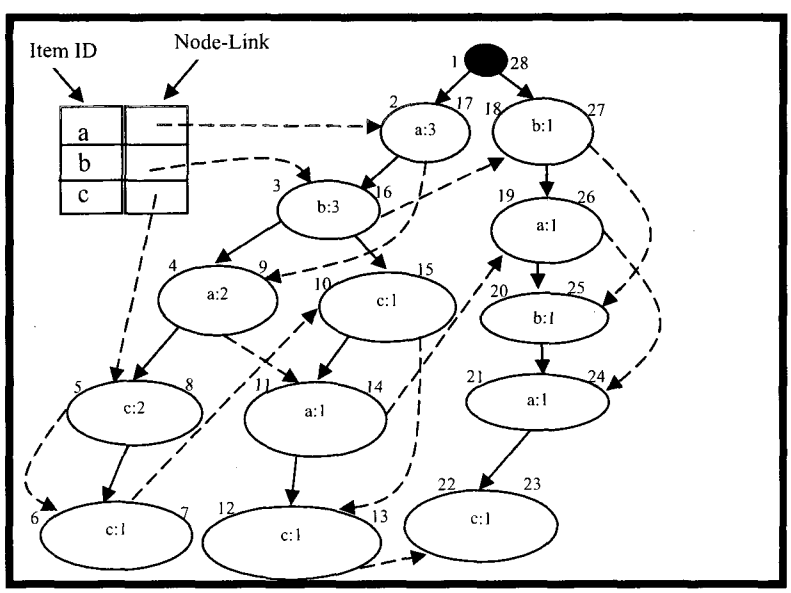


Figure 26 Complete new numeric position code tree

Algorithm buildDesc(): After the transformation of the linked list based PLWAP tree to array based PLWAP-Long tree, buildDesc() method is called to create last descendant of each node. Running example was assigning last descendant was presented in section 3.2.2. Formal definition of this method is given below.

Input: Tree ,F_1-list
Output: All nodes assigned index of last descendant of same event.
Intermediate variables: unassignedNode stack to keep track of nodes that have not yet assigned the lastDesc.
Begin:
1. for each event, ei, in link header
    1.1. for j = 0 to number of e_i events
        1.1.1. If j + 1 > number of e_i events
                set e_j.lastDesc = j
            1.1.1.1. if stack is not empty
                Set (stack.pop).lastDesc = j
        1.1.2. If e_j is ancestor of e_{j+1}, push (ej) to the stack
        1.1.3. Else set e_j.lastDesc = j
            1.1.3.1. if stack is not empty
                while stack.front not ancestor of e_{j+1}
                Set (stack.pop).lastDesc = j
End

Figure 27 Algorithm buildDesc()

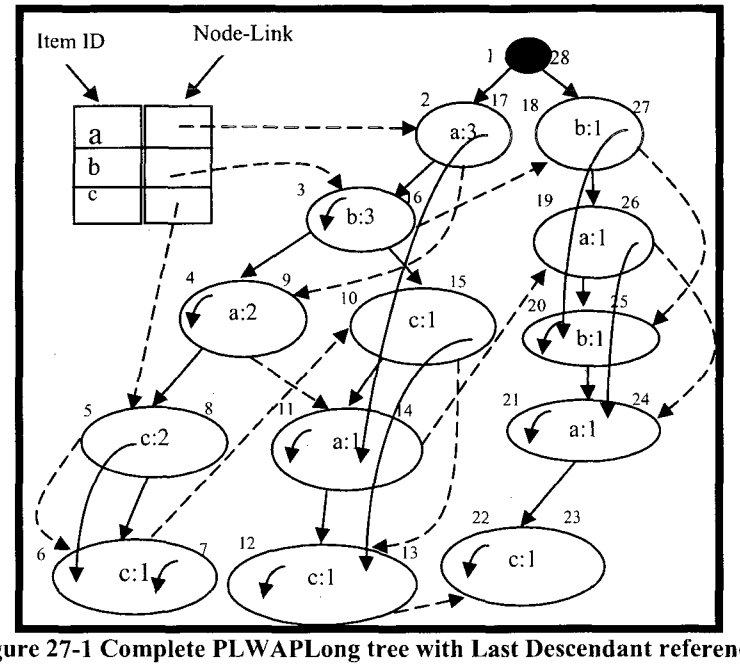


Figure 27-1 Complete PLWAPLong tree with Last Descendant references

Algorithm PLWAPLong1-Mine:.

Input: PLWAP-Long tree T, header linkage table L, minimum support λ ($0 < \lambda \leq 1$), 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. I stores the event array index

Begin

(1) If R is empty, return

(2) For each event, ei in L, find the suffix tree of e_i in T (i.e, e_i|suffixtree), do

a. Save first event in e_i-queue to S

b. Following the e_i-queue

If event e_i.label equals event.label in R

while root set R is not empty,

Set I = array index of event e_r of R

If e_{I+1} is descendant of event e_r in R

Insert e_{I+1} into suffix-tree header set R'

Add count of e_{I+1} to C

Set I = index of last Descendant of e_{I+1}

Else

Set I = array index of event e_r of R

Else

If event ei 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

c. If C is greater that λ

Append e_i after F to F' and output F'

Call Algorithm PLWAPLong-Mine passing R' and F'

End

Figure 28 Algorithm PLWAPLong1-Mine

Algorithm PLWAPLong2-Mine:.

Input: PLWAPLong tree T, header linkage table L, *minimum support* λ ($0 < \lambda \leq 1$), 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. I stores the event array index

Begin

(3) If R is empty, return

(4) For each event, ei in L, find the suffix tree of e_i in T (i.e, e_i|suffixtree), do

a. Save first event in e_i-queue to S

b. Following the e_i-queue

If event ei is the descendant of any event in R, and is not descendant of S,

Insert it into suffix-tree-header set R'

Jump to the last descendant of e_i

Add count of e_i to C

Replace the S with e_i

c. If C is greater that λ

Append e_i after F to F' and output F'

Call Algorithm PLWAPLong-Mine passing R' and F'

End

Figure 29 Algorithm PLWAPLong2-Mine

4. Performance Analysis

The PLWAPLong1 and PLWAPLong2 algorithms have various advantages over the PLWAP algorithm, which include

1. For very long sequences exceeding thirty two nodes, the PLWAP algorithm's performance begins to degrade because it employs linked lists to store conjunctions of long position codes and the linked list traversals slow down the algorithm both during tree construction and mining. PLWAPLong1 and PLWAPLong2 use new position code numbering scheme that speeds up the processing of determining the ancestor/descendant relationship between two nodes.
2. PLWAPLong1 and PLWAPLong2 algorithm use 'last descendant' to skip unnecessary comparisons when creating new root set during mining. Once the node is found that is added to the root set PLWAPLong takes O(1) time to jump to the next suffix tree. PLWAP algorithm, on the other hand, does not know that all the descendants of the node that is added to the root set should not be tested anymore for ancestor/descendant relationship. Hence it ends ups testing all of events in the event queue.

4.1 Experimental Evaluation

This section compares the experimental performance of PLWAP against PLWAPLong1 PLWAPLong2 algorithms. All these algorithms are implemented in C++. PLWAPLong1 experiments against PLWAP are performed on 2.26 GHz Intel machine with 2 GB of RAM. The operating system is Windows XP. PLWAPLong2 experiments against PLWAP are performed on high speed UNIX SUN microsystem with a total of 16384 Mb memory and 8 x 1200 MHz processor speed. Synthetic datasets are generated using publicly available synthetic data set generation program of IBM Quest data mining project at http://www.almaden.ibm.com/cs/quest/. We performed experiments on two sets of data 1) Short sequence data (Sequence length < 32) and 2) Long sequence data (Sequence length > 32). Following parameters were used to generate the dataset |D| = Number of sequences in the database, |C| = Average length of the sequences and |N| = Number of events. The data generated with IBM quest had no repetition of events in any sequences meaning that no user ever visited the same web page more than once. This does not reflect the real life web usage model where users tend to visit same page of a given web domain more than once. To have this data behavior in our dataset we used modulus operator to decrease the values within a range and have repetitions. For example, if we have a sequence like

10000 10 45 91 101 165 179 654 679 777 876 986

After taking modulus by 15 we will have following resulting sequence with repetition as desired

10000 10 0 1 11 0 14 9 4 12 6 11

For large sequences we used modulus 45 and for short sequences we used modulus 15.

4.1.1 PLWAPLong1 vs PLWAP

4.1.2 Short Sequence Experiments

First we used short sequence datasets and performed 4 experiments which are as follows.

4.1.2.1 Experiment 1

For experiment 1 we generated 50K records (|D| = 50K) with |C| = 16 and |N| = 15.

50K Records				
Min Support	**5**	**10**	**15**	**25**
PLWAP	52	10	5	1
PLWAPLong	22	6	3	1

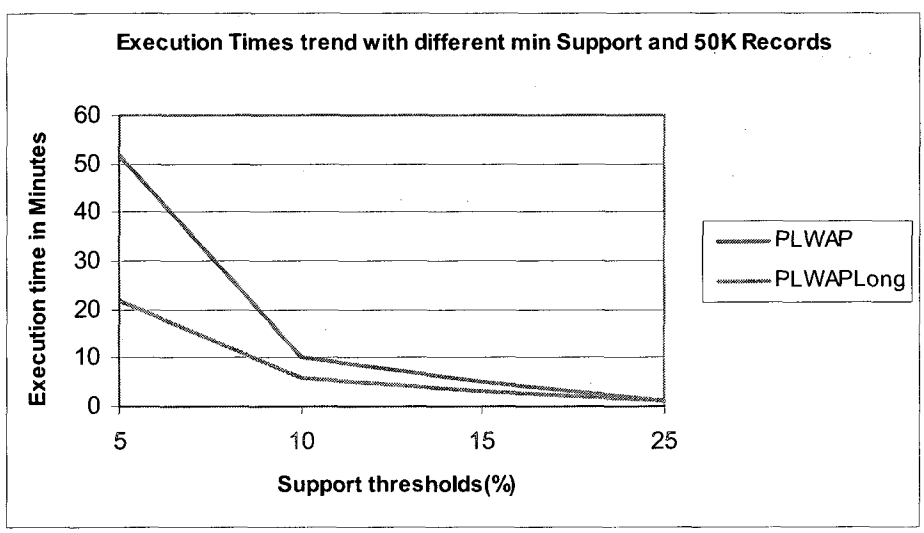


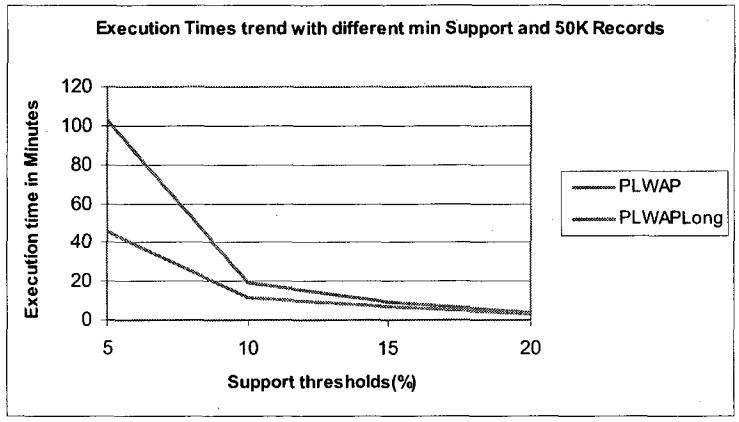
Figure 30 Short Sequence- Experiment 1 (PLWAPLong1)

Tests were run with 4 different minimum support thresholds. From the results we can see that almost plwaplong outperformed plwap with all thresholds tested and the difference in the execution times was more than double as we lowered the min support.

4.1.2.2 Experiment 2

For experiment 2 we generated 100K records (|D| = 100K) with |C| = 16 and |N| = 15.

100K Records				
Min Support	5	10	15	20
PLWAP	103	19	9	4
PLWAPLong	46	12	7	3



Tests were run with 4 different minimum support thresholds.

Figure 31 Short Sequence-Experiment 2 (PLWAPLong1)

From the results we can see that almost plwaplong outperformed plwap with all thresholds tested and the difference in the execution times was more than double as we lowered the min support.

4.1.2.3 Experiment 3

For experiment 3 we generated 300K records (|D| = 300K) with |C| = 16 and |N| = 15.

300K Records				
Min Support	5	10	15	20
PLWAP	312	58	28	11
PLWAPLong	146	37	20	9

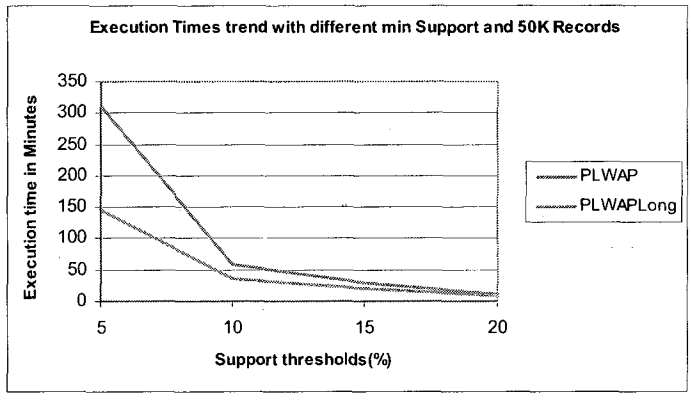


Figure 32 Short Sequence-Experiment 3 (PLWAPLong1)

Tests were run with 4 different minimum support thresholds.

From the results we can see that almost plwaplong outperformed plwap with all thresholds tested and the difference in the execution times was more than double as we lowered the min support.

4.1.2.4 Experiment 4

In this experiment we ran the data generated in previous 4 experiments at minsupport of 5%. From the results we see that as the database size increases the execution time becomes double for plwap.

5% Min Support			
DB Size (K)	50	100	300
PLWAP	52	103	312
PLWAPLong	22	46	146

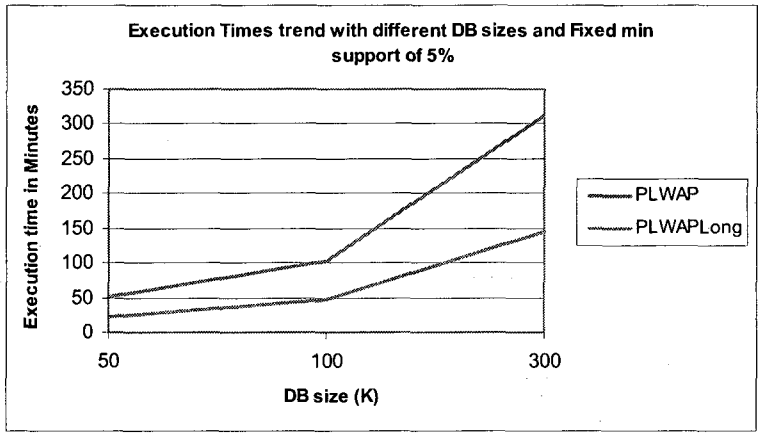


Figure 33 Short Sequence-Experiment 4 (PLWAPLong1)

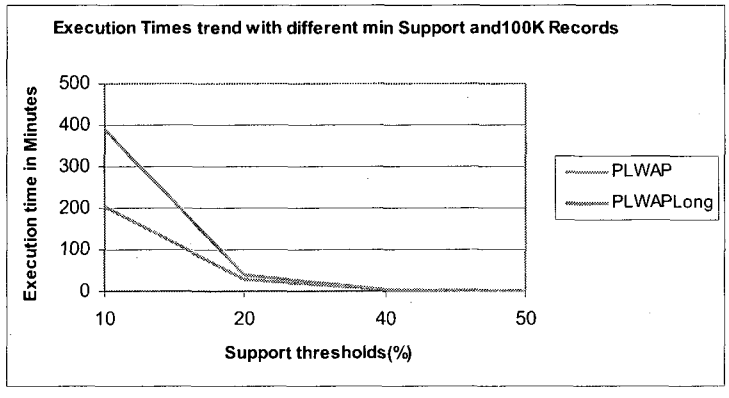
4.1.3 Long Sequence Experiments

For Long sequences we performed 4 experiments which are as follows

4.1.3.1 Experiment 1

For experiment 1 we generated 100K records (|D| = 100K) with |C| = 39 and |N| = 45. Tests were run with 4 different minimum support thresholds.

From the results we can see that almost plwaplong outperformed plwap with all thresholds tested and the difference in the execution times was more than double as we lowered the min support.



100K Records				
Min Support	10	20	40	50
PLWAP	388	40	3	1
PLWAPLong	203	27	2	1

Figure 34 Long Sequence- Experiment 1 (PLWAPLong1)

4.1.3.2 Experiment 2

For experiment 2 we generated 300K records (|D| = 300K) with |C| = 39 and |N| = 45. Tests were run with 4 different minimum support thresholds. From the results we can see that plwaplong outperformed plwap with all thresholds tested and the difference in the execution times was more than double as we lowered the min support.

300K Records				
Min Support	10	20	30	40
PLWAP	1389	125	29	8
PLWAPLong	653	86	22	7

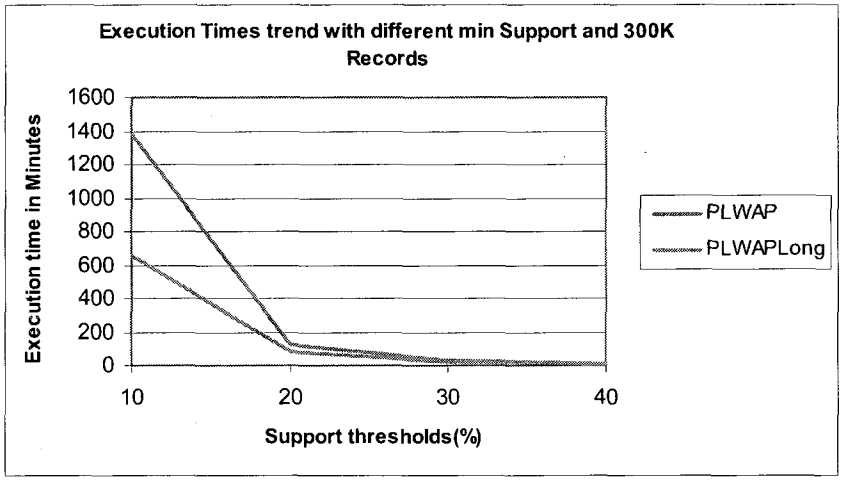


Figure 35 Long Sequence- Experiment 2 (PLWAPLong1)

4.1.3.3 Experiment 3

For experiment 3 we generated 600K records (|D| = 600K) with |C| = 39 and |N| = 45.

600K Records				
Min Support	40	50	55	60
PLWAP			4	3
PLWAPLong	15	8	4	3

Tests were run with 4 different minimum support thresholds. Plwap program crashed with segmentation fault with minsupport 50. plwaplong program on the other hand was able to run up to min support 40.

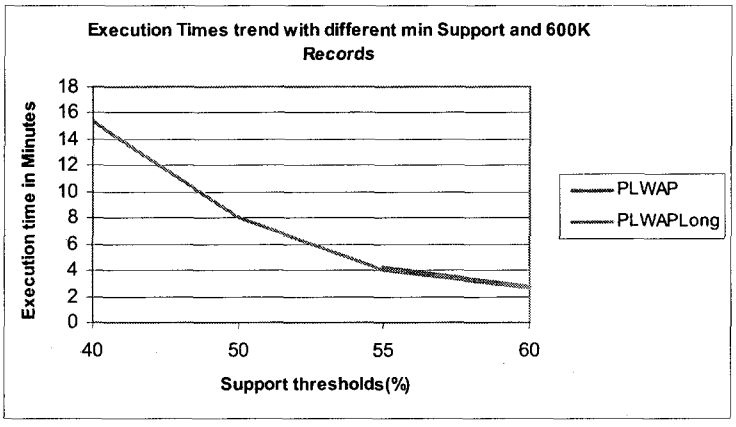


Figure 36 Long Sequence-Experiment 3 (PLWAPLong1)

Some tests were also conducted on UNIX machine for PLWAPLong1 against PLWAP and results indicate that PLWAPLong1 runs slower than PLWAP on UNIX machine. All tests were run at 50%.

Long Sequences		
PLWAP		
	Memory Usage	**Execution time**
100K	198M	143
300K	594M	420
600K	1182M	850
PLWAPLong		
	Memory Usage	**Execution time**
100K	238M	238
300K	710M	789
600K	1402M	1640
Short		

PLWAP		
	Memory Usage	**Execution time**
50K	1182M	859
100K	1182M	854
300K	1182M	861
600K	462M	182
PLWAPLong		
	Memory Usage	**Execution time**
50K	1402M	1674
100K	1402M	1652
300K	1402M	1679
600K	594M	317

4.2 PLWAPLong2 vs PLWAP

4.2.1 Long Sequences

We used the data from PLWAPLong1 testing for testing PLWAPLong2 as well. The |C| was set at 39 and |N| at 45 for all of the data sets for long sequences. We performed several tests which are as follows

4.2.1.1 Experiment 1

In this experiment we tested small database samples in the range of 2K to 14K records with fix minimum support of 15%. From the results we can see that as we increased the database size the speedup of plwaplong2 algorithm was almost 50% as compared to plwap algorithm.

Small DB 2K - 14K with 15% min support							
DB Size	2K	4K	6K	8K	10K	12K	14K
# of Freq Patterns	5159	4707	4633	4417	4708	4686	4413
plwap	225	583	942	1248	1754	2163	2388
Plwaplong2	148	310	560	746	990	1228	1372

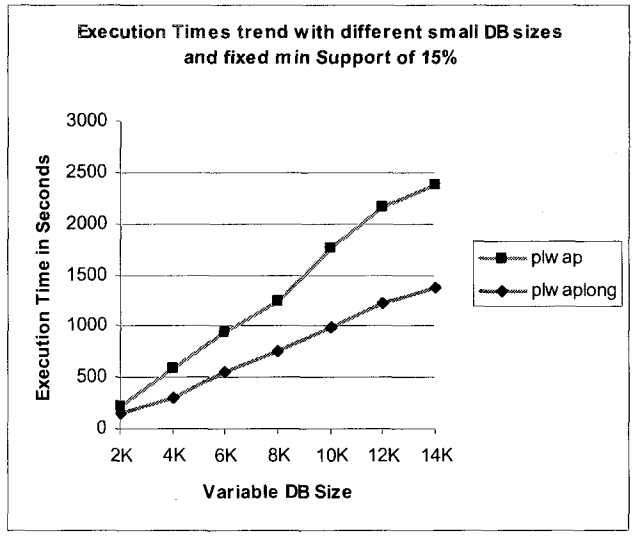


Figure 37 Long Sequence- Exp1 (PLWAPLong2)

4.2.1.2 Experiment 2

In this experiment we used 1 million records database and tested it with varying minimum support threshold, i.e. from 30% to 50%. From the experiments we can see that as we lowered the min support to 35% and 30%, plwaplong2 algorithm produced more frequent patterns and also ran much faster than the plwap algorithm.

Large Size DB (1M) with variable support 30% - 50%							
Min Support	30	35	40	45	50	55	60
# of Freq Patterns	282	124	61	47	32	16	7
plwap	13749	6799	3512	2622	1571	570	250
Plwaplong2	8095	4281	2369	1852	1029	462	235

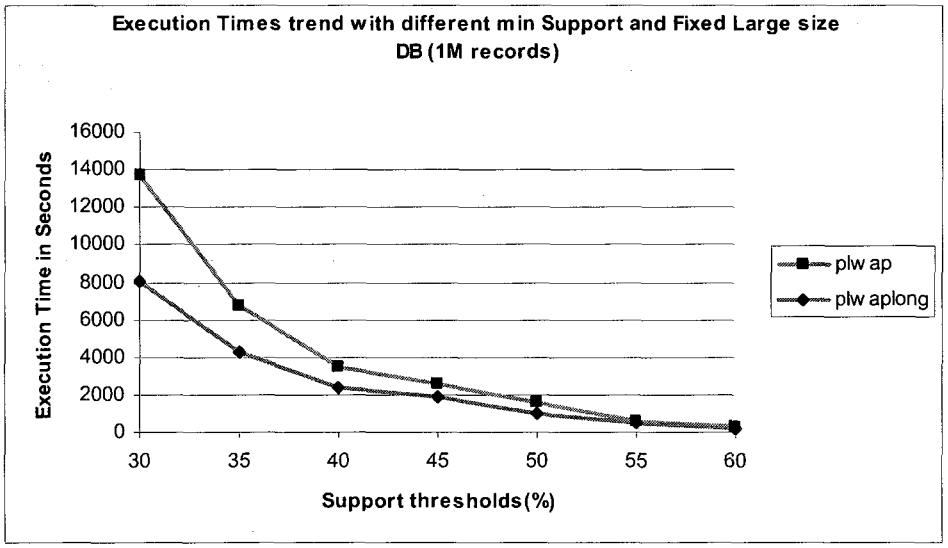


Figure 38 Long Sequence- Exp2 (PLWAPLong2)

4.2.1.3 Experiment 3

In this experiment we test fixed size small database sample of 40K records and tested it with varying minimum support threshold from 15% to 45%. From the experimental results we can see that plwaplong2 algorithm clearly outperformed plwap algorithm for all test points and especially when we lowered the min support.

Small DB (40K) with variable support 15% - 45%							
	15	20	25	30	35	40	45
# of Freq Patterns	4581	1623	712	280	123	61	47
Plwap	7307	2913	1435	635	314	165	128
Plwaplong2	4283	1924	917	414	214	124	92

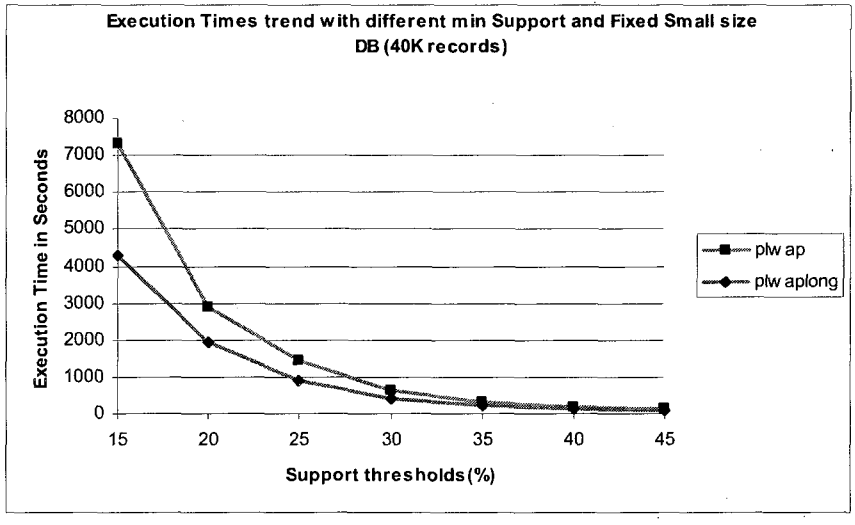


Figure 39 Long Sequence- Exp3 (PLWAPLong2)

4.2.1.4 Experiment 4

In this experiment we tested medium size database of 300K records with varying minimum support from 40% to 65%. Although plwaplong2 algorithm outperformed plwap algorithm for all test points but it was more effective when we tested these algorithms at min support threshold of 40%.

Medium DB (300K) with variable support						
Min Support	40	45	50	55	60	65
plwap	1051	888	433	182	76	61
Plwaplong2	821	626	335	144	70	60

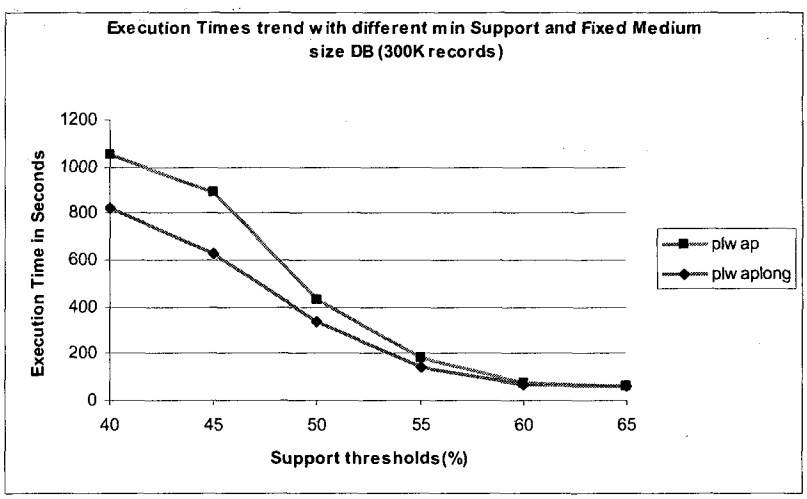


Figure 40 Long Sequence- Exp4 (PLWAPLong2)

4.2.1.5 Experiment 5

In this experiment we tested varying medium size database sample from 20K to 200K with fix minimum support threshold of 35%. From the test results we can clearly see that plwaplong2 algorithm greatly outperformed plwap algorithm as we increased the dataset size.

Medium DB (20K - 200 K) with fixed min support of 35%						
DB Size	20K	40K	60K	80K	100K	200K
plwap	147	297	445	607	712	1494
Plwaplong2	93	186	278	381	474	967

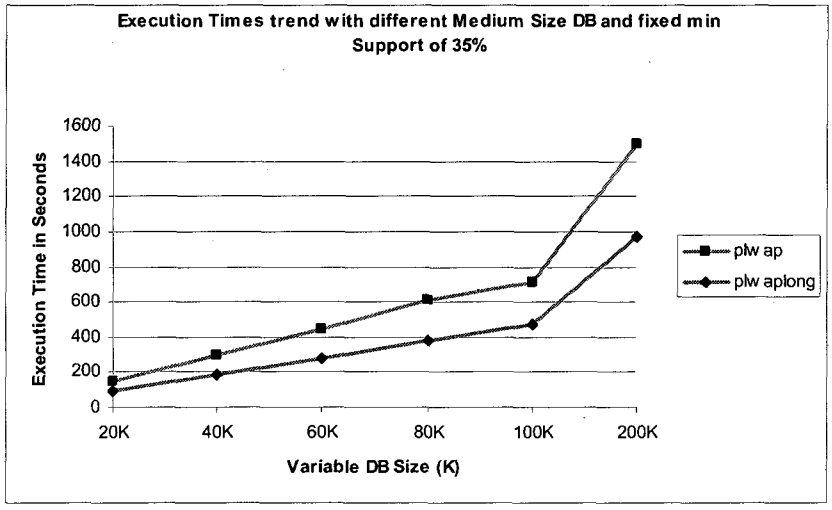


Figure 41 Long Sequence- Exp5 (PLWAPLong2)

4.2.1.6 Experiment 6

In this experiment we tested varying large database set from 400K to 900K records at fix minimum support threshold of 50%. As we increased the database size, plwaplong2 algorithm showed great speedup as compared to the plwap algorithm.

Large DB (400K - 900K) with fixed min support of 50%						
DB Size	400K	500K	600K	700K	800K	900K
plwap	525	876	1057	1207	1420	1581
Plwaplong2	376	600	714	815	960	1060

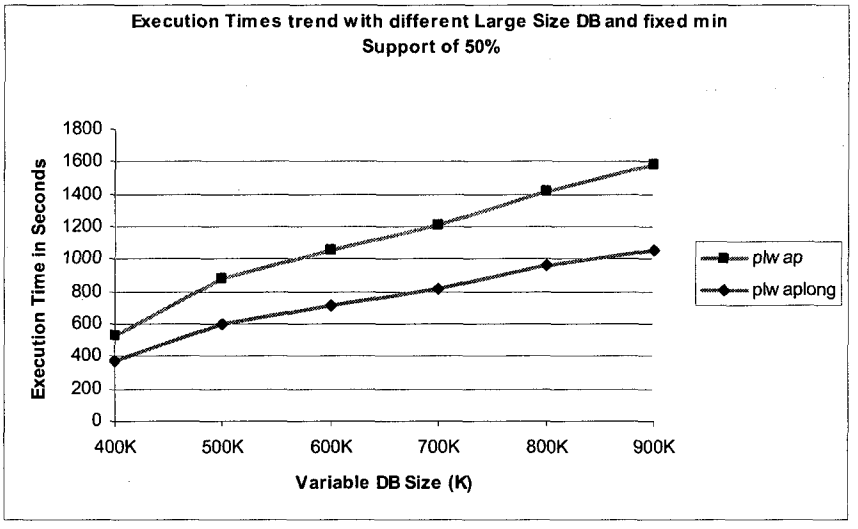


Figure 42 Long Sequence- Exp6 (PLWAPLong2)

4.2.1.7 Experiment 7

In this experiment we tested the memory usage of the two algorithms against varying large database sample from 400K to 900K records at fix minimum support threshold of 50%. From the results we can clearly see that plwaplong2 algorithm managed memory more efficiently than plwap algorithm especially at large database sample of 900K.

Large DB (400K - 900K) with fixed min support of 50%- Memory Usage						
DB Size	400K	500K	600K	700K	800K	900K
plwap	591	991	1179	1375	1582	1763
Plwaplong2	411	659	807	939	1067	1211

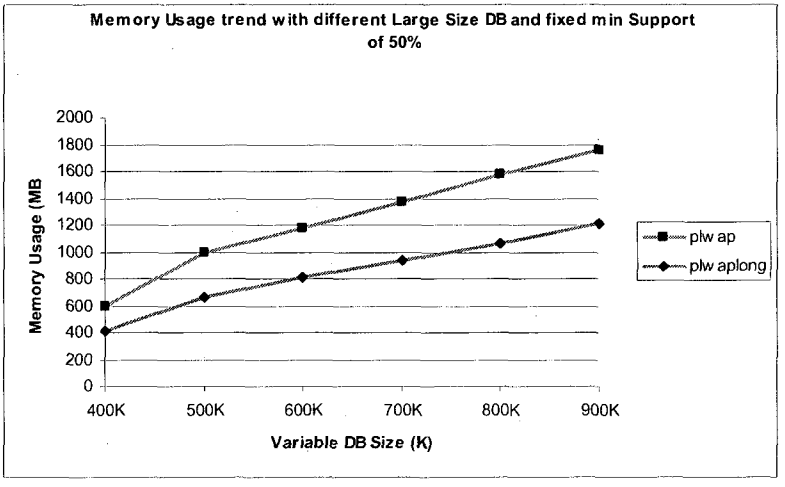


Figure 43 Long Sequence- Exp7 (PLWAPLong2)

4.2.2 Short Sequences

We used the data from PLWAPLong1 testing for testing PLWAPLong2 for short sequences as well. The |C| was set at 16 and |N| at 15 for all of the data sets for short sequences. We performed several tests which are as follows

4.2.2.1 Experiment 1

In this experiment we tested small database sample of 100K records with varying minimum support threshold from8% to 30%. We can see from the results that plwaplong2 algorithm outperformed plwap algorithm for every test point and showed a great speedup when we lowered the min support threshold.

Small DB (100K) with variable min support (8% - 30%)						
Min Support	**8**	**10**	**15**	**20**	**25**	**30**
plwap	4724	3206	1267	475	341	251
Plwaplong2	3388	2203	880	341	236	170

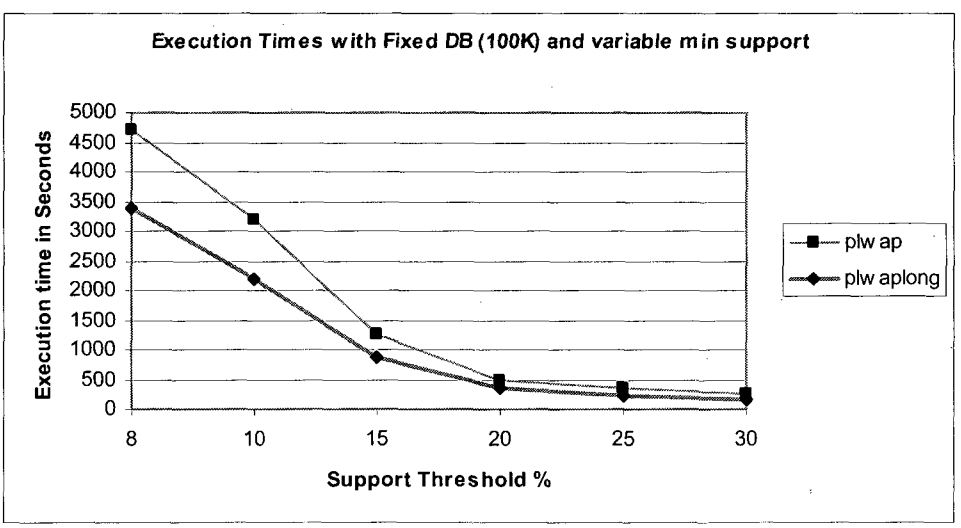


Figure 44 Short Sequence- Exp1 (PLWAPLong2)

4.2.2.2 Experiment 2

In this experiment we tested medium size database of 300K records against varying minimum support threshold from15% to 45%. Experimental results show that plwaplong2 algorithm outperformed plwap algorithm for all test points.

Medium DB (300K) with variable min support (15% - 45%)						
Min Support	**15**	**20**	**25**	**30**	**35**	**40**
Plwap	3691	1411	944	742	455	285
Plwaplong2	2594	1005	667	325	185	106

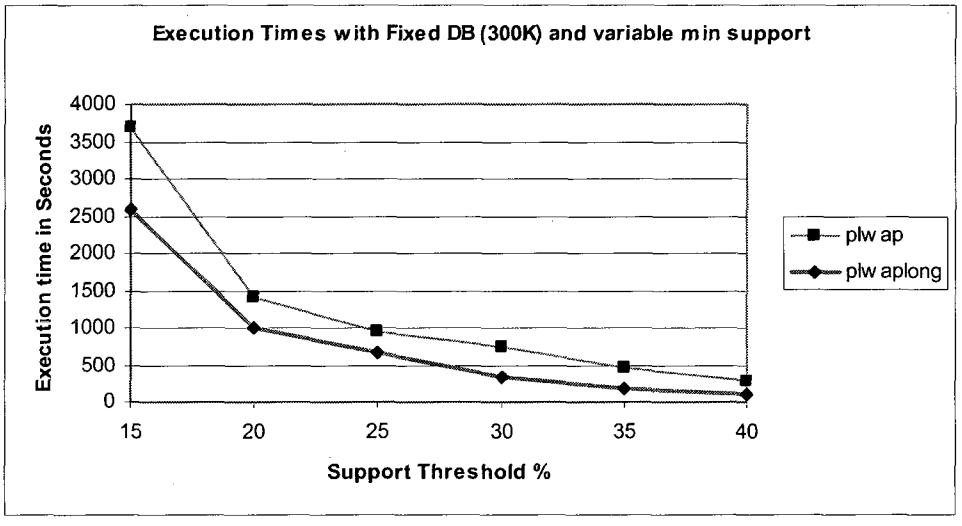


Figure 45 Short Sequence- Exp2 (PLWAPLong2)

4.2.2.3 Experiment 3

In this experiment we tested variable database size samples with fix minimum support threshold of 15%. Experimental results show that plwaplong2 algorithm outperformed plwap algorithm for all test points and especially for 200K dataset, plwaplong2 speedup was almost 50%.

Variable DB (1K - 200K) 15% min support						
	1K	10K	20K	40K	80K	200K
plwap	10	148	334	725	1426	3436
Plwaplong2	6	67	160	366	721	1772

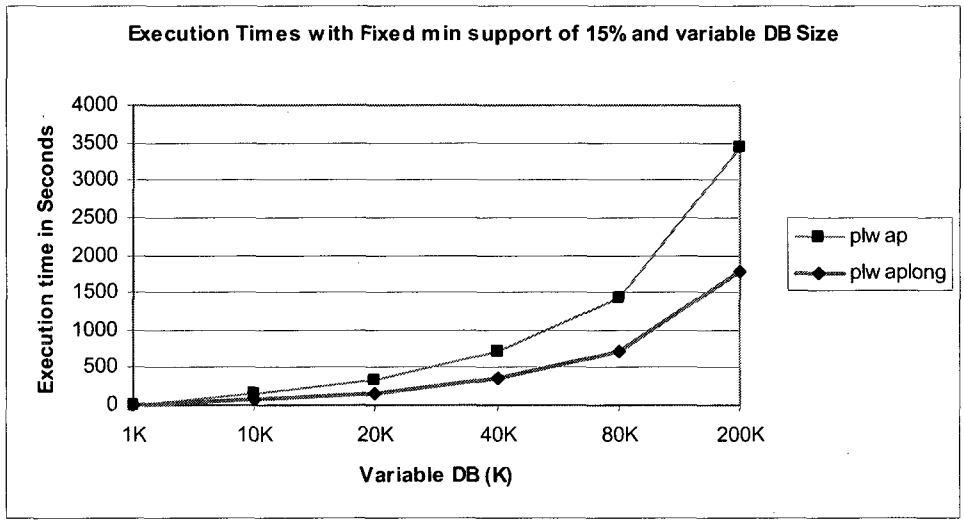


Figure 46 Short Sequence- Exp3 (PLWAPLong2)

4.2.2.4 Experiment 4

In this experiment we tested the memory usage of the two algorithms by testing variable database size with fixed min support threshold of 15%. From the results we can see that plwaplong2 algorithm efficiently manage the memory usage for all test points as compared to plwap algorithm.

Variable DB (1K - 200K) 15% min support				Memory Usage		
	1K	10K	20K	40K	80K	200K
plwap	3.9	15	23	43	79	191
Plwaplong2	3.7	11	19	35	63	143

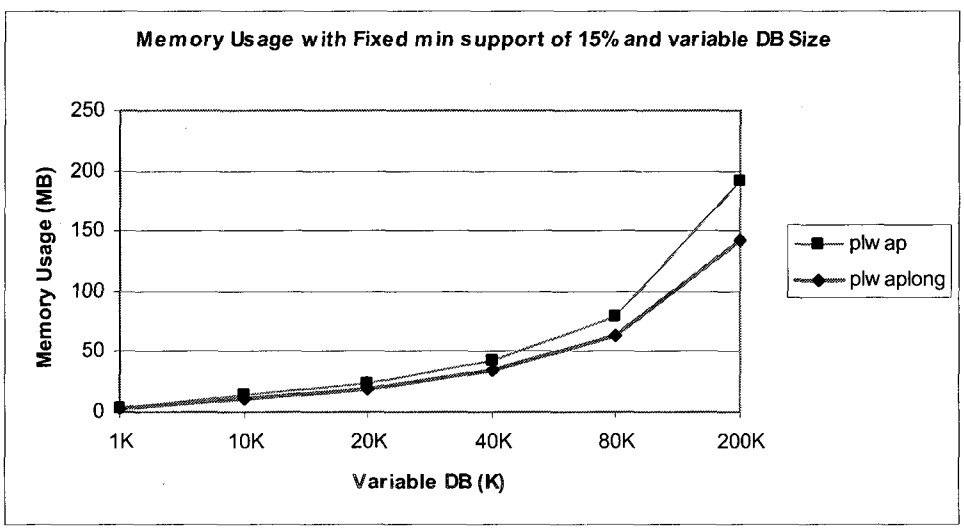


Figure 47 Short Sequence- Exp4 (PLWAPLong2)

5. Conclusions & Future Work

This thesis presented two new algorithms (PLWAPLong1 and PLWAPLong2) for efficiently mining very long sequences from the web usage log. PLWAPLong1 adapts the PLWAP tree structure to initially store the frequent patterns but later transforms it to an array data structure equivalent to the PLWAP tree. In order to avoid expensive and useless comparisons of event nodes to determine the suffix trees as done by PLWAP algorithm, PLWAPLong1 algorithm employees using 'Last Descendant' technique that quickly eliminates the unwanted nodes from ancestor/descendant comparison and jumps to the next root to continue finding the suffix tree. PLWAPLong1 algorithm also uses binary search to quickly find the next node that should be tested for suffix tree root set. The experiments indicate that PLWAPLong1 outperforms PLWAP when tested on Windows based OS but PLWAP outperforms PLWAPLong1 on UNIX based machine. PLWAPLong2 algorithm outperformed PLWAP with great improvement in both execution time and memory usage for all of the test scenarios.

5.1 Future Work

Since this is a very first effort to use very long sequence for sequential pattern mining, we feel there is still room of improvement. After careful study of the plwap and plwaplong algorithms, we feel that during the mining of the plwaplong tree, at times same suffix tree is mined more than once. Future work could look at eliminating this redundant

exploration of same suffix trees and instead use the frequent sequences that are already found in that suffix tree.

6. References

1. [A96] R. S. R. Agrawal. Mining sequential patterns: Generalizations and performance improvements. In Proceedings of the Fifth International Conference On Extending Database Technology (EDBT '96) Avignon France, 1996. Pages 3-17.

2. [AIS93] Rakesh Agrawal, Tomasz Imielinski, and Arun Swami. Mining Association rules between sets of items in large databases. *In Proc. Of the ACM SIGMOD conference on management of data,* Washington, DC, 1993. Pages 207-216.

3. [AKM+01] Suhail Ansari, Ron Kohavi, LIew Mason, Zijian Zheng. Integrating e-commerce and data mining: architecture and challenges, *Proceedings IEEE International Conference on Data Mining*, San Jose, CA .USA., *2001.* Pages: 27-34.

4. [AS94] Rakesh Agrawal, Ramakrislman Srikant. Fast Algorithms for Mining Association Rules, *Proceedings of the* 20th *VLDB conference*, Santiago, Chile, 1994. , Pages 487-499.

5. [AS95] Rakesh Agrawal, Ramakrishnan Srikanth. Mining Sequential Patterns, *Research Report, IBM Almaden Research Center 650 Hariy Road, San Jose, CA 95120, IEEE Publication,* 1995, Pages 1-22.

*6. [AS95b]*R. Agrawal and R. Srikant. Mining sequential patterns. In Proceedings of the 11th International Conference on Data Engineering (ICDE '95) Taipei Taiwan, 1995. Pages 3–14.

7. [BL99] Jose Borges, Mark Levene: Data mining of user navigation patterns. In Proceedings of the KDD Workshop on Web Mining.San Diego, California, 1999. Pages. 31–36.

8. [BM98] Alex G. Buchner, Maurice D. Mulvenna. Discovering Internet Marketing Intelligence through Online Analytical Web Usage Mining, SIGMOD Record, Vol.27, No.4, New York, NY, USA, 1998. Pages 54-61.

9. [BS03] Blackwell B. and Ravada S.: Oracle's Technology for Bioinformatics and Future Directions. First Asia-Pacific Bioinformatics Conference, Adelaide, Australia. Conferences in Research and Practice in Information Technology, Vol 19, 2003, Pages. 35 -42.

10. [CH+96] M.S. Chen, J. Han, and P.S. Yu. Data mining: An overview from a database perspective. IEEE Transac-tions on Knowledge and Data Engineering, 8(6), 1996: Pages 866-883.

11. [CMS99] R. Cooley, B. Mobasher, J. Srivastava. Data preparation for mining world wide web browsing patterns, *Knowledge and Information Systems,* 1(1), 1999. Pages 1-26.

12. [CP95] L. Catledge, J. Pitkow: Characterizing browsing behaviors on the World Wide Web, Computer Networks and ISDN Systems, 1995. Page 27(6).

13. [E96] *Oren Etzioni: The world wide web: Quagmire or gold mine. Communications of the ACM, 39(1).* Pages 65–68.

14. [ELL05] C.I Ezeife, Yi Lu, Yi Liu: PLWAP Sequential Mining, Open Source Code, proceedings of the Open Source Data Mining Wrokshop on Frequent Pattern Mining Implementations, in conjunction with ACM SIGKDD, Chicago, IL, U.S.A., 2005

15. [ECO4a] C.I.. Ezeife and Mm Chen, Mining Web Sequential Patterns Incrementally with Revised PLWAP Tree, *proceedings of the fifth International Conference on Web-Age Information Management (WAIM 2004) Dalian, 15* . 17 July 2004, published in

LNCS by Springer Verlag.

16. [ECO4b] C.I. Ezeife and Mm Chen, Incremental Mining of Web Sequential Patterns Using PLWAP Tree on Tolerance MinSupport, *proceedings of the IEEE 8t1~ International Database Engineering and Applications Symposium,* Coimbra, Portugal, July 7th to 9th 2004.

17. [FL05] Facca, F. and Lanzi, P.. Mining interesting knowledge from weblogs: A survey. Data Knowledge and Engineering, 2005. Pages 225-241.

18. [GK+05] Cong G., Tan K., Tung A.K.H. and Xu X.: Mining biological and medical data: Mining top-K covering rule groups for gene-expression data. SIGMOD 2005 June 14-16, 2005. Pages 670-681.

19. [GT+04] Cong G., A. K. H. Tung, X. Xu, F. Pan, and Yang J. Farmer: Finding interesting rule groups in microarray datasets. In 23rd ACM International Conference on Management of data, 2004, Pages. 143-154.

20. [Hin02] Phillip Hingston. Using Finite State Automation for Sequence Mining, *Proceeding of the twenty-fifth Australasian conference on computer science - volume 4,* Melbourne, Victoria, Australia, 2002. Pages 105 - 110.

21. [LH+98] H. Lu, J. Han, and L. Feng. Stock movement and n-dimensional inter-transaction association rules. In Proc.1998 SIGMOD Workshop Research Issues on Data Mining and Knowledge Discovery (DMKD'98), Seattle, WA, June 1998. Pages 12:1–12:7.

22. [HP+00] J. Han, J. Pei, B. Mortazavi-Asl, Q. Chen, U. Dayal, and M.-C. Hsu. Freespan: Frequent pattern-projected sequential pattern mining. In Proc. 2000 Int. Conf.

Knowledge Discovery and Data Mining (KDD'00), Boston, MA, Aug. 2000. Pages 355–359.

23. [HPY00] Jiawei Han, Jian Pei, Yiwen Yin. Mining frequent patterns without candidate generation, *Proceedings of the 2000 ACM SIGMOD international conference on Management of data and Symposium on Principles of Database Systems*, Dallas, Texas, USA, 2000. Pages 1-12.

24. [IBM] http://www.almaden.ibm.com/cs/quest. Quest Data Mining Project, IBM Almaden Research center, San Jose, CA 95120.

25. [J04] Sikorski J.: Mining Association Rules in Microarray Data. Guest Speaker presentation at the Pace University, USA, Course: DCS861A Emerging IT II, Spring 2004

26. [JC04] Cohen J.: Bioinformatics—An Introduction for Computer Scientists. ACM Computing Surveys, Vol. 36, No. 2, June 2004, Pages. 122-158.

27. [JM+05] Wang T.L.J, Zaki J.M., Toivonen T.T Hannu, Shasha Dennis :Data Mining in Bioinformatics (Book). Springer publication 2005. ISBN: 1-85233-671-4

28. [LE03] Y. Lu and C. I. Ezeife. Position Code Pre-Ordered Linked WAP-Tree for Web log sequential pattern mining, *In Proceedings of the 7th Pac~flc-Asia Conference on Knowledge Discovery and Data Mining (PAKDD 2003),* Seoul, Korea, 2003.

29. [LE05] Yi Lu and C.I. Ezeife, "Mining Web Log sequential Patterns with Position Coded Pre-Order Linked WAP-tree", the International Journal of Data Mining and Knowledge Discovery (DMKD), Vol. 10, pages. 5-38, Kluwer Academic Publishers, June 2005

30. [LP00] Juhnyoung Lee, Mark Podlaseck. Using a Starfield Visualization for

Analyzing Product Performance of Online Stores, *Proceedings of the 2nd ACM conference on Electronic commerce table of contents,* Minneapolis, Minnesota, United States, 2000. Pages 168 *-175.*

31. [MA04] Sushmita M, Tinku A. DATA MINING, Multimedia, Soft Computing and bioinformatics. (BOOK). A John Wiley & Sons,INC., Publication .2004 ISBN- 9812-53-063-0

32. [MBN+99] Sanjay Kumar Madria, Sourav S Bhowmick, W. -K Ng, E. P. Lim: "Research issues in Web Data Mining". DaWaK'99, LNCS 1676, 1999. Pages. 303-312.

33. [MD+01] Bamshad Mobasher, Honghua Dai, Tao Luo, Miki Nakagawa: "Effective Personalization Based on Association Rule Discover from Web Usage Data" In Proceedings of WIDM 2001, Atlanta, GA, USA, Pages. 9-15

34. [MW97] Martin F Arlitt and Carey L Williamson. Internet web servers: Workload Characterization and performance implications. IEEE/ACM Transactions on Networking, 5(5): 1997. Pages 631-645.

35. [P97] J. Pitkow, In search of reliable usage data on the WWW, Sixth International World Wide Conference, 1997.

36. [PC+03] F. Pan, G. Cong, A. K. H. Tung, J. Yang, and M. J. Zaki. Carpenter: Finding closed patterns in long biological datasets. In Proc. 2003 ACM SIGKDD Int. Conf. on Knowledge Discovery and Data Mining (KDD'03), 2003, Pages 637-642

37. [PG] Parmar, J. and Garg, S.: Modified Web Access Pattern (mWAP). Approach for Sequential Pattern Mining. Int. J. of Computer Cciences, Vol. 6, No.2, 46-54.

38. [PH+01] J. Pei, J. Han, B. Asl, Q. Chen, U. Dayal, and M.Hsu, Pre¯xspan: mining sequential patterns efficiently by prefix-projected pattern growth, ICDE, 2001.

39. [PHM+00] Jian Pei, Jiawei Han, Behzad Mortazavi-asi, Hua Zhu. Mining Access Patterns Efficiently from web logs, *Proceedings 2000 Pac~flc-Asia conference on Knowledge Discovery and data Mining*, Kyoto, Japan, 2000. Pages 3 96-407

40. [PHM+01] J. Pei, J. Hart, B. Mortazavi-Asl, H. Pinto, Q. Chen, U. Dayal, and M.-C. Hsu. PrefixSpan mining sequential patterns efficiently by prefix projected pattern growth. In ICDE 2001, Heidelberg, Germany, Apr. 2001. Pages 215-26

41. [SA96] Ramakrishnan Srikanth and Rakesh Agrawal. Mining Sequential Patterns: generalizations and performance improvements, Research Report, IBM Almaden ResearchCenter 650 I-larry Road San Jose, CA 95120, 1996. Pages 1-15.

42. [SA+01] Doddi S, Marathe A, Ravi SS, Torney DC. Discovery of association rules in medical data. PubMed. 2001 Jan-Mar.26(1): Pages 25-33

43. [SCD+00] J. Srivastava, R. Cooley, M. Deshpande, P. N. Tan. Web usage mining: Discovery and apageslications of usage patterns from web data, S1GKL~D Explorations, Volume 1, Issue 2, 2000. Pages 12-23

44. [SM+99] Eric Schmitt, Harley Manning, Yolanda Paul, Joyce Tong: Measuring Web Success Forrester Report, Nov 1999.

45. [SP+98] Shahana Sen, Balaji Padmanabhan, Alexander Tuzhilin, Norman H. White, Roger Stein: The identificationand satisfaction of consumer analysis-driven information needs of Marketing, Vol. 32 No. 7/8 1998

46. [WX04] Ke Wang, Yabo Xu and Jeffrey Xu Yu: Scalable Sequential Pattern Mining for Biological Sequences. CIKM'04, November 8–13, 2004, Washington, DC, USA.Copyright 2004 ACM 1581138741/04/0011

47. [XG+04] Xu X., Cong G., Ooi B.C., Tan K. and Tung A.K.H: Semantic Mining and

Analysis of Gene-expression Data. Proceedings of the 30th VLDB Conference, Toronto, Canada, 2004. Pages. 1261-1264

48. [YX+04] Ye Y., Wu X., Subramanian K.R. and Zhang L.: GenExplore: Interactive Exploration of Gene Interactions from Microarray Data. Proceedings of the 20th International Conference on Data Engineering (ICDE'04).. March 30 - April 2, 2004 Omni Parker House Hotel Boston, USA. Page-680

49. [WP08] Wenjia Wang (IEEE Member) and Phuong Thanh Cao-Thai : Novel Position-Coded Methods for Mining Web Access Patterns. Intelligence and Security Informatics, 2008. ISI 2008. IEEE International Conference on 17-20 June 2008 Pages:194-196

50. [Z01] M. J. Zaki, SPADE: An efficient algorithm for mining frequent sequences, Machine Learning Journal, Special Issue on Unsupervised Learning, Vol. 42, No. 1/2, 2001. Pages. 31-60.

Vita Auctoris

NAME	Kashif Saeed
PLACE OF BIRTH	Lahore, Pakistan
YEAR OF BIRTH	1979
EDUCATION	Department of Computer Science University of Windsor, Windsor, ON, Canada B.Sc. (2005)
	Department of Computer Science University of Windsor, Windsor, Ontario, Canada M.Sc. (2005-2009)