PLWAP Sequential Mining: Open Source Code [*]

C.I. Ezeife
School of Computer Science
University of Windsor
Windsor, Ontario N9B 3P4
cezeife@uwindsor.ca

Yi Lu
Department of Computer Science
Wayne State University
Detroit, Michigan
luyi@wayne.edu

Yi Liu
School of Computer Science
University of Windsor
Windsor, Ontario N9B 3P4
woddlab@uwindsor.ca

ABSTRACT

PLWAP algorithm uses a preorder linked, position coded version of WAP tree and eliminates the need to recursively re-construct intermediate WAP trees during sequential mining as done by WAP tree technique. PLWAP produces significant reduction in response time achieved by the WAP algorithm and provides a position code mechanism for remembering the stored database, thus, eliminating the need to re-scan the original database as would be necessary for applications like those incrementally maintaining mined frequent patterns, performing stream or dynamic mining.

This paper presents open source code for both the PLWAP and WAP algorithms describing our implementations and experimental performance analysis of these two algorithms on synthetic data generated with IBM quest data generator. An implementation of the Apriori-like GSP sequential mining algorithm is also discussed and submitted. A web log pre-processor for producing real input to the algorithms is made available too.

Keywords

sequential patterns, web usage mining, WAP tree, pre-order linkage

1. INTRODUCTION

Basic association rule mining with the Apriori algorithm [1] finds database items (attributes) that occur most often together in database transactions. Thus, given a set of transactions (similar to database records), where each transaction is a set of items (attributes), an association rule is of the form $X \rightarrow Y$, where X and Y are sets of items and $X \cap Y = \emptyset$. Association rule mining algorithms generally first find all frequent patterns (itemsets) as all combinations of items or attributes with support (percentage occurrence in the entire database), greater or equal to a predefined minimum support. Then, in the second stage of mining, association rules are generated from each frequent pattern by defining all possible combinations of rule antecedent (head) and consequent (tail) from items composing the frequent patterns such that $antecedent \cap consequent = \emptyset$ and $antecedent \cup consequent = frequentpattern$. Then, only rules with confidence (number of transactions that contain the rule divided by the number of transactions containing the antecedent) greater than or equal to a pre-defined minimum confidence are retained as valuable, while the rest are pruned.

Sequential mining is an extension of basic association rule mining that accommodates ordered set of items or attributes, where the same item may be repeated in a sequence. While basic frequent pattern has a set of non-ordered items that have occurred together up to minimum support threshold, frequent sequential pattern has a sequence of ordered items that have occurred frequently in database transactions at least as often as the minimum support threshold. Thus, the measures of support and confidence used in association rule mining for deciding frequent itemsets are used in sequential mining for deciding frequent sequences. Just as an i-itemset contains i items, an n-sequence contains n ordered items (events). One application of sequential mining is web usage mining for finding the relationship among different web users' accesses from web access logs [5], [11], [4] and [19]. Analysis of these access data can help for server performance enhancement and direct marketing in e-commerce as well as web personalization. Before applying sequential mining techniques to web log data, the web log transactions are pre-processed to group them into set of access sequences for each user identifier and to create web access sequences in the form of a transaction database (e.g., *abdac*, *eaebcac*, *babfaec*, *babfaec*). Access sequence $S' = e'_1e'_2 \dots e'_l$ is called a subsequence of an access sequence, $S = e_1e_2...e_n$, and S is a super-sequence of S', denoted as $S' \subseteq S$, if and only if for every event e'_j in S', there is an equal event e_k in S, while the order that events occurred in S is the same as the order of events in S'. For example, with $S' = ab$, S = *babcd*, S' is a subsequence of S, and *ac* is a subsequence of S, although there is *b* occurring between *a* and *c* in S. In the sequence *babcd*, while *bcd* is a suffix subsequence of *bab*, *bab* is a prefix subsequence of *bcd*. Techniques for mining sequential patterns from web logs fall into Apriori or non-Apriori.

This paper presents discussions of the implementations of three key sequential mining algorithms PLWAP, WAP and GSP used in [7].

[*](A full version of this paper is available in the Journal of Data Mining and Knowledge Discovery 10, 5-38, 2005.)

1.1 Related Work

Work on mining sequential patterns in web log include the GSP [3], the PSP [13], the G sequence [18] and the graph traversal [15] algorithms. Agrawal and Srikant proposed three algorithms (Apriori, AprioriAll, AprioriSome) for sequential mining in [2]. The GSP (Generalized Sequential Patterns) [3] algorithm is 20 times faster than the Apriori algorithm. The GSP Algorithm makes multiple passes over data. The first pass determines the frequent 1-item patterns (L_1). Each subsequent pass starts with a seed set: the frequent sequences found in the previous pass (L_{k-1}). The seed set is used to generate new candidate sequences (C_k) by performing an Apriori gen join of L_{k-1} with (L_{k-1}). This join requires that every sequence s in the first L_{k-1} joins with other sequences s' in the second L_{k-1} if the last *k-2* elements of s are the same as the first *k-2* elements of *s*'. For example, if frequent 3-sequence set L_3 has the following 6 sequences: {((1,2)(3)), ((1,2)(4)), ((1)(3, 4)), ((1,3)(5)), ((2)(3,4)), ((2)(3)(5))}, to obtain frequent 4-sequences, every frequent 3-sequence should join with the other 3-sequences that have the same first two elements as its last two elements. Sequence s=((1,2)(3)) joins with s'=((2)(3,4)) to generate a candidate 4-sequence ((1,2)(3,4)) since the last 2 elements of s, (2)(3), match with the first 2 elements of s'. Similarly, ((1,2) (3)) joins with ((2)(3)(5)) to form ((1,2)(3)(5)). There are no more matching sequences to join in L_3. The join phase is followed with the pruning phase, when the candidate sequences with any of their contiguous (k-1)-subsequences having a support count less than the minimum support, are dropped. The database is scanned for supports of the remaining candidate k-sequences to find frequent k-sequences(L_k), which become the seed for the next pass, candidate (k+1)-sequences. The algorithm terminates when there are no frequent sequences at the end of a pass, or when there are no candidate sequences generated. The GSP algorithm uses a hash tree to reduce the number of candidates that are checked for support in the database.

The PSP (Prefix Tree For Sequential Patterns) [13] approach is much similar to the GSP algorithm [3], but stores the database on a more concise prefix tree with the leaf nodes carrying the supports of the sequences. At each step k, the database is browsed for counting the support of current candidates. Then, the frequent sequence set, L_k is built.

The Graph Traversal mining [14], [15], uses a simple unweighted graph to store web sequences and a graph traversal algorithm similar to Apriori algorithm to traverse the graph in order to compute the k-candidate set from the (k-1)-candidate sequences without performing the Apriori-gen join. From the graph, if a candidate node is large, the adjacency list of the node is retrieved. The database still has to be scanned several times to compute the support of each candidate sequence although the number of computed candidate sequences is drastically reduced from that of the GSP algorithm. Other tree based approaches include [18] called G sequence mining. This algorithm uses wildcards, templates and construction of Aggregate tree for mining.

The FP-tree structure [9] first reorders and stores the frequent non-sequential database transaction items on a prefix tree, in descending order of their supports such that database transactions share common frequent prefix paths on the tree. Then, mining the tree is accomplished by recursive construction of conditional pattern bases for each frequent 1-item (in ordered list called f-list), starting with the lowest in the tree. Conditional FP-tree is constructed for each frequent conditional pattern having more than one path, while maximal mined frequent patterns consist of a concatenation of items on each single path with their suffix f-list item. FreeSpan [8] like the FP-tree method, lists the f-list in descending order of support, but it is developed for sequential pattern mining. PrefixSpan [16] is a pattern-growth method like FreeSpan, which reduces the search space for extending already discovered prefix pattern p by projecting a portion of the original database that contains all necessary data for mining sequential patterns grown from p.

Table 1: Sample Web Access Sequence Database for WAP-tree

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

Web access pattern tree (WAP), is a non-Apriori algorithm, proposed by Pei et al. [17]. The WAP-tree stores the web log data in a prefix tree format similar to the frequent pattern tree [9] (FP-tree). WAP algorithm first scans the web log to compute all frequent individual events, then it constructs a WAP-tree over the set of frequent individual events of each transaction before it recursively mines the constructed WAP tree by building a conditional WAP tree for each conditional suffix frequent pattern found. The process of recursive mining of a conditional suffix WAP tree ends when it has only one branch or is empty.

An example application of the WAP-tree algorithm for finding all frequent events in the web log (constructing the WAP-tree and mining the access patterns from the WAP tree) is shown with the database in Table 1. Suppose the minimum support threshold is set at 75%, which means an access sequence, s should have a count of 3 out of 4 records in our example, to be considered frequent. Constructing WAP-tree, entails first scanning database once, to obtain events that are frequent, a, b, c. When constructing the WAP-tree, the non-frequent (like d, e, f) part of every sequence is discarded. Only the frequent sub-sequences shown in column three of Table 1 are used as input. With the frequent sequence in each transaction, the WAP-tree algorithm first stores the frequent items as header nodes for linking all nodes of their type in the WAP-tree in the order the nodes are inserted. When constructing the WAP-tree, a virtual root (Root) is first inserted. Then, each frequent sequence in the transaction is used to construct a branch from the Root to a leaf node of the tree. Each event in a sequence is inserted as a node with count 1 from Root if that node type does not yet exist, but the count of the node is increased by 1 if the node type already exists. Also, the head link for the inserted event is connected (in broken lines) to the newly inserted node from the last node of its type that was inserted or from the header node of its type if it is the very first node of that event type inserted. Once the frequent sequential data are stored on the complete WAP-tree (Figure 1), the tree is mined for frequent patterns starting with the lowest frequent event in the header list, in our example,

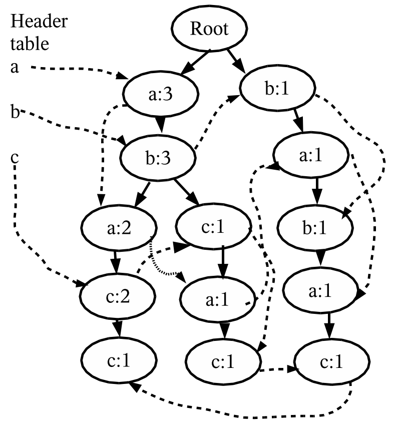

Figure 1: Construction of the Original WAP Tree

starting from frequent event c as the following discussion shows. From the WAP-tree of Figure 1, it first computes prefix sequence of the base c or the conditional sequence base of c as: aba : 2; ab : 1; abca : 1; ab : -1; baba : 1; abac : 1; aba : -1 The conditional sequence list of a suffix event is obtained by following the header link of the event and reading the path from the root to each node (excluding the node). After discarding the non-frequent part c in the above sequences, the conditional sequences based on c are listed below:
aba : 2; ab : 1; aba : 1; ab : -1; baba : 1; aba : 1; aba : -1. Using these conditional sequences, a conditional WAP tree, WAP-tree|c, is built using the same method as shown in Figure 1. The re-construction of WAP trees that progressed as suffix sequences |c, |bc discovered frequent patterns found along this line c, bc and abc. The recursion continues with the suffix path |c, |ac. The algorithm keeps running, finding the conditional sequence bases of *bac*, *b*, *a*. Figure 2 shows the WAP trees for mining conditional pattern base c. After mining the whole tree, discovered frequent pattern set is: {c, aac, bac, abac, ac, abc, bc, b, ab, a, aa, ba, aba}.

Although WAP-tree algorithm scans the original database only twice and avoids the problem of generating explosive candidate sets as in Apriori-like algorithm, its main drawback is recursively re-constructing large numbers of intermediate WAP-trees and patterns during mining taking up computing resources.

Pre-Order linked WAP tree algorithm (PLWAP) [7], [10], is a version of the WAP tree algorithm that assigns unique binary position code to each tree node and performs the header node linkages pre-order fashion (root, left, right). Both the pre-order linkage and binary position codes enable the PLWAP to directly mine the sequential patterns from the one initial WAP tree starting with prefix sequence, without re-constructing the intermediate WAP trees. To assign position codes to a PLWAP node, the root has null code, and the leftmost child of any parent node has a code that appends '1' to the position code of its parent, while the position code of any other node has '0' appended to the position code of its nearest left sibling. The PLWAP technique presents a much better performance than that achieved by the WAP-tree technique as shown in extensive performance analysis.

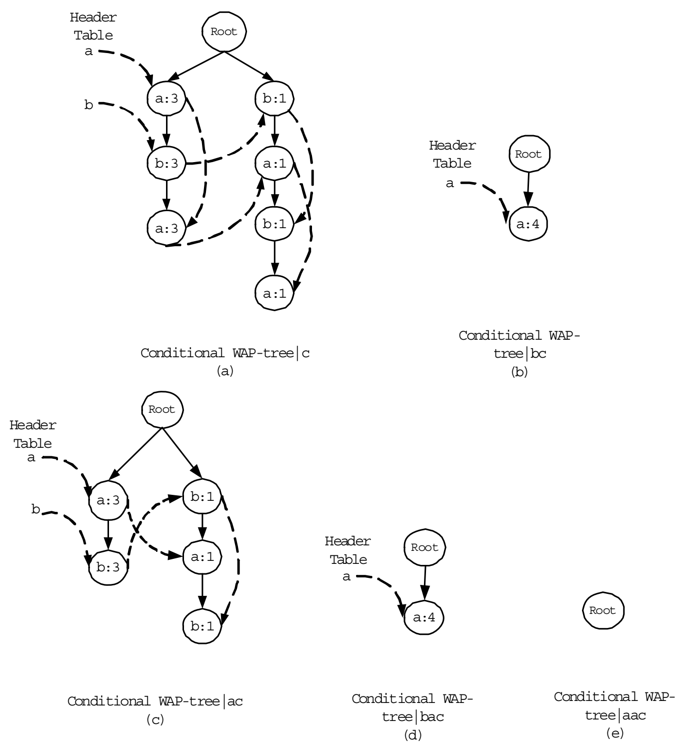

Figure 2: Reconstruction of WAP Trees for Mining Conditional Pattern Base c

1.2 Motivations and Contributions

PLWAP algorithm [7], a recently proposed sequential mining tool in the Journal of Data Mining and Knowledge Discovery has many attractive features that makes it suitable as a building block for many other sophisticated sequential data mining approaches like incremental mining [6], web classification and personalization. This paper supplements the detailed and formal presentations of the PLWAP algorithm, its properties and theorems given in [7] by focusing on details of the code implementations of PLWAP, WAP and GSP sequential miners as well as making available a real web log data pre-processor and providing further indepth analysis. This paper thus, contributes by discussing our code implementations of the PLWAP and two other key sequential mining algorithms (WAP and GSP) used in performance studies of work in [7]. These algorithms have been tested thoroughly on publicly available data sets (http://www.almaden.ibm.com/software/quest/Resources/index.shtml) and with real data. Through this paper, a real web log pre-processor for preparing real input data for the miners is also made available (ask author if needed). A more indepth performance analysis that confirms the stability of the PLWAP algorithm is another contribution of paper. Such code availability motivates adoption of algorithm as a building block for more sophisticated data mining processes like stream and dynamic mining, distributed, incremental, drill-down and roll-up mining, semantic mining and sensor network mining.

1.3 Outline of the Paper

Section 2 discusses example mining with the Pre-Order Linked WAP-Tree (PLWAP) algorithm. Section 3 discusses the C++ implementations of the PLWAP, WAP and the GSP algorithms for sequential mining. Section 4 discusses experimental performance analysis, while section 5 presents conclusions and future work.

2. AN EXAMPLE SEQUENTIAL MINING WITH PLWAP ALGORITHM

Unlike the conditional search in WAP-tree mining, which is based on finding common suffix sequence first, the PLWAP technique finds the common prefix sequences first. The main idea is to find a frequent pattern by progressively finding its common frequent subsequences starting with the first frequent event in a frequent pattern. For example, if *abcd* is a frequent pattern to be discovered, the WAP algorithm progressively finds suffix sequences *d*, *cd*, *bcd* and *abcd*. The PLWAP tree, on the other hand, would find the prefix event *a* first, then, using the suffix trees of node *a*, it will find the next prefix subsequence *ab* and continuing with the suffix tree of *b*, it will find the next prefix subsequence *abc* and finally, *abcd*. Thus, the idea of PLWAP is to use the suffix trees of the last frequent event in an *m-prefix sequence* to recursively extend the subsequence to *m+1* sequence by adding a frequent event that occurred in the suffix trees. Using the position codes of the nodes, the PLWAP is able to know the descendant and sibling nodes of oldest parent nodes on the suffix root set of a frequent header element being checked for appending to a prefix subsequence if it is frequent in the suffix root set under consideration. An element is frequent if the sum of the supports of the oldest parent nodes on all its suffix root sets is greater than or equal to the minimum support.

Assume we want to mine the web access database (WASD) of Table 1 for frequent sequences given a minimum support of 75% or 3 transactions. Constructing and mining the PLWAP tree goes through the following steps. (1) Scan WASD (column 2 of Table 1 once to find all frequent individual events, L as {a:4, b:4, c:4}. The events d:1, e:2, f:2 have supports less than the minimum support of 3 and
(2) Scan WASD again, construct a PLWAP-tree over the set of individual frequent events (column 3 of Table 1), by inserting each sequence from root to leaf, labeling each node as (node event: count: position code). Then, after all events are inserted, traverse the tree pre-order way to connect the header link nodes. Figure 3 shows the completely constructed PLWAP tree with the pre-order linkages.
(3) Recursively mine the PLWAP-tree using common prefix pattern search: The algorithm starts to find the frequent sequence with the frequent 1-sequence in the set of frequent events(FE) {a, b, c}. For every frequent event in FE and the suffix trees of current conditional PLWAP-tree being mined, it follows the linkage of this event to find the first occurrence of this frequent event in every current suffix tree being mined, and adds the support count of all first occurrences of this frequent event in all its current suffix trees. If the count is greater than the minimum support threshold, then this event is appended (concatenated) to the last subsequence in the list of frequent sequences, F. The suffix trees of these first occurrence events in the previously mined conditional suffix PLWAP-trees are now in turn, used for mining the next event. To obtain this conditional PLWAP-tree, we only need to remember the roots of the current suffix trees, which are stored for next round mining. For example, the

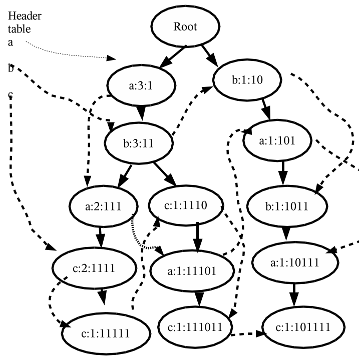

Figure 3: Construction of PLWAP-Tree Using Pre-Order Traversal

algorithm starts by mining the tree in Figure 4(a) for the first element in the header linkage list, *a* following the *a* link to find the first occurrencies of *a* nodes in *a:3:1* and *a:1:101* of the suffix trees of the Root since this is the first time the whole tree is passed for mining a frequent 1-sequence. Now, the list of mined frequent patterns F is {a} since the count of event *a* in this current suffix trees is 4 (sum of a:3:1 and a:1:101 counts), and more than the minimum support of 3. The mining of frequent 2-sequences that start with event *a* would continue with the next suffix trees of *a* rooted at {b:3:11, b:1:1011} shown in Figure 4(b) as unshadowed nodes. The objective here is to find if 2-sequences *aa*, *ab* and *ac* are frequent using these suffix trees. In order to confirm aa frequent, we need to confirm event *a* frequent in the current suffix tree set, and similarly, to confirm *ab* frequent, we should again follow the *b* link to confirm event *b* frequent using this suffix tree set, same for *ac*. The process continues to obtain same frequent sequence set {a, aa, aac, ab, aba, abac, abc, ac, b, ba, bac, bc, c} as the WAP-tree algorithm.

3. C++ IMPLEMENTATIONS OF THREE SEQUENTIAL MINING ALGORITHMS

Although, we lay more emphasis on the PLWAP algorithm developed in our lab, we also provide the source codes of the WAP and GSP algorithms used for performance analysis. The source codes are discussed under seven headings of (1) development and running environment, (2) input data format and files, (3) minimum support format, (4) output data format and files, (5) functions used in the program, (6) data structures used in the program and (7) additional information. All of the codes are documented with information on this section and more for code readability, maintainability and extendability. Each program is stored in a .cpp file and compiled with "g++ filename.cpp". It is worth noting that all three algorithms were implemented in the year 2002, when no other versions of the WAP and GSP algorithms were publicly available. While we cannot still find a publicly available version of the GSP program, there are some subtle differences between our implementation of the WAP algorithm and that now available at http://www.cs.ualberta.ca/ tszhu/software.html. One difference we have noticed is with the input formats of the two implementations: while Alberta version accepts the input in one sequence, we accept a list of sequences belonging to different users as in web log mining. It also seems like Alberta

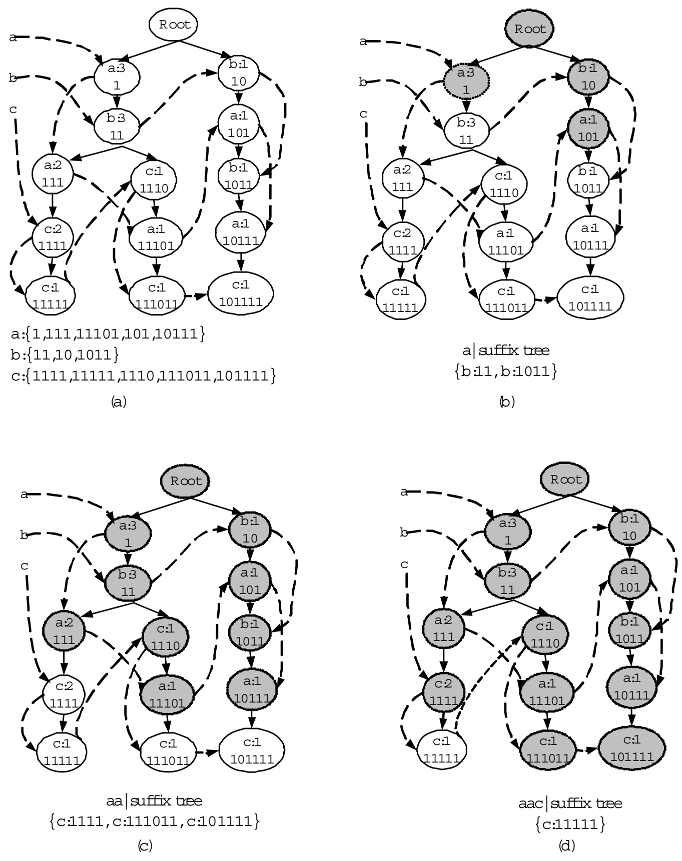

Figure 4: Mining PLWAP-Tree to Find Frequent Sequence Starting with aa

version is memory-only as intermediate pattern input is not saved on disk, while it is saved on disk in our implementation. The implication is that running the memory-only version on an environment with less memory than data size would result in operating system swapping of data onto disk.

3.1 C++ Implementation of the PLWAP Sequential Mining Algorithm

This is the PLWAP algorithm program, which is based on the description in [7], C.I. Ezeife and Y. Lu. "Mining Web Log sequential Patterns with Position Coded Pre-Order Linked WAP-tree" in DMKD 2005.
1. DEVELOPMENT ENVIRONMENT: Although initial version is developed under the hardware/software environment specified below, the program runs on more powerful and faster multiprocessor UNIX environments as well. Initial environment is: (i)Hardware: Intel Celeron 400 PC, 64M Memory; (ii)Operating system: Windows 98; (iii)Development tool: Inprise(Borland) C++ Builder 6.0. The algorithm is developed under C++ Builder 6.0, but compiles and runs under any standard C++ development tool.
2. INPUT FILES AND FORMAT: Input file is test.data: For simplifying input process of the program, we assume that all input data have been preprocessed such that all events belonging to the same user id have been gathered together, and formed as a sequence and saved in a text file, called, "test.data". The "test.data" file may be composed of hundreds of thousands of lines of sequences where each line represents a web access sequence for each user. Every line of the input data file ("test.data") includes UserID, length of sequence and the sequence which are separated by tab spaces. An example input line is: 100 5 10 20 40 10 30 . Here, 100 represents UserID, the length of sequence is 5, the sequence is 10,20,40,10,30.
3. MINIMUM SUPPORT FORMAT: The program also needs to accept a value between 0 and 1 as minimum support. The minimum support input is entered interactively by the user during the execution of the program when prompted. For a minimum support of 50%, user should type 0.5, and for minsupport of 5%, user should type .05, and so on.
4. OUTPUT FILES AND FORMAT: result_PLWAP.data: Once the program terminates, we can find the result frequent patterns in a file named "result_PLWAP.data". It may contain lines of patterns, each representing a frequent pattern.
5. FUNCTIONS USED IN THE CODE: (i)BuildTree: builds the PLWAP tree, (ii)buildLinkage: Builds the pre-order linkage for PLWAP tree, (iii)makeCode: makes the position code for a node, (1v)checkPosition: checks the position between any two nodes in the PLWAP tree, (v)MiningProcess: mines sequential frequent patterns from the PLWAP tree using position codes.
6. DATA STRUCTURE: Three struct are used in this program: (i) the node struct indicates a PLWAP node which contains the following information: a.the event name, b.the number of occurrence of the event, c. a link to the position code, d. length of position code, e. the linkage to next node with same event name in PLWAP tree, f. a pointer to its left son, g. a pointer to its right sibling, h. a pointer to its parent, i. the number of its sons. (ii) a position code struct implemented as a linked list of unsigned integer to make it possible to handle data of any size without exceeding the maximum integer size. (iii) a linkage struct.
7. ADDITIONAL INFORMATION: The run time is displayed on the screen with start time, end time and total seconds for running the program.

3.2 C++ Implementation of the WAP Sequential Mining Algorithm

This is the WAP algorithm program based on the description in [17]: Jian Pei, Jiawei Han, Behzad Mortazavi-asl, and Hua Zhu, "Mining Access Patterns Efficiently from Web Logs", PAKDD 2000.
1.DEVELOPMENT ENVIRONMENT: The same as described for PLWAP and can run on any UNIX system as well.
2. INPUT FILES AND FORMAT:
i) Input file test.data: Pre-processed sequential input records are read from the file "test.data". The "test.data" file, which is the same format as described for PLWAP above.
ii) Input file middle.data: used to save the conditional middle patterns. During the WAP tree mining process, following the linkages, once the sum of support for an event is found greater than minimum support, all its prefix conditional patterns are saved in the "middle.data" file for next round mining. The format of "middle.data" is as follows: Each line includes the length of the sequence, the occurrence of the sequence, and the events in the sequence. For example, given a line in middle.data: 5 4 10 20 40 10 30, the length of sequence is 5, 4 indicates the sequence occurred 4 times in the previous conditional WAP tree and the sequence is 10,20,40,10,30.
3. MINIMUM SUPPORT:

The program also needs to accept a value between 0 and 1 for minimum support or frequency. The user is prompted for minimum support when the program starts.
4. OUTPUT FILES AND FORMAT: result_WAP.data
Once the program terminates, the result patterns are in a file named "result_WAP.data", which may contain lines of patterns.
5. FUNCTIONS USED IN THE CODE:
(i)BuildTree: Builds the WAP tree/conditional WAP tree
(ii)MiningProcess: produces sequential pattern/conditional prefix sub-pattern from WAP tree/conditional WAP tree.
6. DATA STRUCTURE: three struct are used in this program: (i) the node struct indicates a WAP node which contains the following information: a.the event name, b.the number of occurrence of the event, c. the linkage to next node same event name in WAP tree, d. a pointer to its left son, e. a pointer to its rights sibling, f. a pointer to its parent.
(ii) a linkage struct described in the program.
7. ADDITIONAL INFORMATION: The run time is displayed on the screen with start time, end time and total seconds for running the program.

3.3 C++ Implementation of the GSP Sequential Mining Algorithm

This is a GSP algorithm implementation, which demonstrates the result described in [3]: R. Srikant and R. Agrawal. "Mining sequential patterns: Generalizations and performance improvements", 1996.
1.DEVELOPMENT ENVIRONMENT: The same as described for PLWAP and runs in any UNIX system as well.
2. INPUT FILES AND FORMAT: Input file test.data: The main input file "test.data" had the same format as for PLWAP described above.
3. MINIMUM SUPPORT: The program also needs to accept a value between 0 and 1 as minimum support. The user is prompted for minimum support when the program starts.
4. OUTPUT FILES AND FORMAT: result_GSP.data At the termination of the program, the result patterns are in a file named "result_GSP.data", which may contain lines of frequent patterns.
5. FUNCTIONS USED IN THE CODE: (i)GSP: reads the file and mines levelwise according to the algorithm.
6. DATA STRUCTURES: There are struct for i) candidate sequence list and its count and ii) sequence.
7. ADDITIONAL INFORMATION: The run time is displayed on the screen with start time, end time and total seconds for running the program.

4. PERFORMANCE AND EXPERIMENTAL ANALYSIS OF THREE ALGORITHMS

The PLWAP algorithm eliminates the need to store numerous intermediate WAP trees during mining, thus, drastically cutting off huge memory access costs. The PLWAP annotates each tree node with a binary position code for quicker mining of the tree. This section compares the experimental performance analysis of PLWAP with the WAP-tree mining and the Apriori-like GSP algorithms. The three algorithms were initially implemented with C++ language running under Inprise C++ Builder environment. All initial experiments were performed on 400MHz Celeron PC machine with 64 megabytes memory running Windows 98 (for work in [7]). Current experiments are conducted with the same implementations of the programs and still on synthetic datasets generated using the resource data generator code from http://www.almaden.ibm.com/software/quest/Resources/index.shtml. This synthetic dataset has been used by most pattern mining studies [3, 12, 17]. Experiments were also run on real datasets generated from web log data of University of Windsor Computer Science server and pre-processed with our web log cleaner code. The correctness of the implementations were confirmed by checking that the frequent patterns generated for the same dataset by all algorithms are the same. A high speed UNIX SUN microsystem with a total of 16384 Mb memory and 8 x 1200 MHz processor speed is used for these experiments. The synthetic datasets consist of sequences of events, where each event represents an accessed web page. The parameters shown below are used to generate the data sets.
$|D|$ Number of sequences in the database
$|C|$: Average length of the sequences
$|S|$: Average length of maximal potentially frequent sequence
$|N|$: number of events
For example, C10.S5.N2000.D60K means that $|C| = 10$, $|S| = 5$, $|N|= 2000$, and $|D| = 60$K. It represents a group of data with average length of the sequences as 10, the average length of maximal potentially frequent sequence is 5, the number of individual events in the database is 2000, and the total number of sequences in database is 60 thousand. The datasets with different parameters test different aspects of the algorithms. Generally, if the number of these four parameters becomes larger, the execution time becomes longer. Experiments are conducted to test the behavior of the algorithms with respect to the four parameters, minimum support threshold, database sizes, number of database table attributes and the average length of sequences. For each experiment, while one of the parameters changes, others are pegged at some constant values. Observations are made at three levels of small, medium and large (e.g., small database size may consist of a table with records less than 40 thousands, medium size table has between 50 nd 200 thousands records, while large has over 300 thousand).
Experiment 1: Execution time trend with different minimum supports (small size database, 40K records):
This experiment uses fixed size database and different minimum support to compare the performance of PLWAP, WAP and GSP algorithms. The datasets are described as C2.5.S5.N50.D40K, and algorithms are tested with minimum supports between 0.05% and 20% against the 40 thousand (40K) database with 50 attributes and average sequence length of 2.5. The results of this experiment are shown in Table 2 and Figure 5 with the number of frequent patterns found reported as Fp. From this result, it can be seen that at minimum support of 0.05%, the number of frequent patterns found is highest and is 2729, the PLWAP algorithm ran almost 3 times faster than the WAP algorithm. As the minimum support increases, the number of frequent patterns found decreases and the gain in performance by the PLWAP algorithm over the WAP algorithm decreases. It can be seen that the more the number of frequent patterns found in a dataset, the higher the performance gain achieved by the PLWAP algorithm over the WAP algorithm. This is

Table 2: Execution Times for Dataset at Different Minimum Supports (small database)

Alg	Runtime (in secs) at Different Supports(%)						
	0.05 Fp= 2729	0.1 Fp= 1265	0.5 Fp = 268	1 Fp= 111	5 Fp= 23	10 Fp= 10	15 Fp 0
GSP	6663	3646	1054	636	157	30	1
WAP	149	66	20	8	3	1	1
PLWAP	54	26	7	3	1	1	1

because the two algorithms spend about the same amount of time scanning the database and constructing the tree, but the PLWAP algorithm saves on storing and reading intermediate re-constructed tree, which are as many as the number of frequent patterns found. The execution times of the two algorithms are the same when there are nearly no frequent patterns.

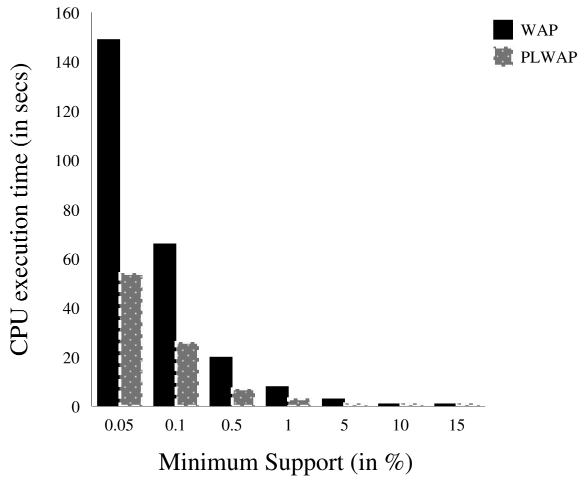

Figure 5: Execution Times Trend with Different Minimum Supports (small database)

Experiment 2: Execution times trend with different minimum supports (Medium size database, 200K records):
This experiment uses fixed size database and different minimum support to compare the performance of PLWAP, WAP and GSP algorithms. The algorithms are tested with minimum supports between 0.05% and 15% against the 200 thousand (200K) database, 50 attributes and average sequence length of 2.5. The results of this experiment are shown in Table 3 and Figure 6 with the number of frequent patterns found reported as Fp. It can be seen that the trend in performance is the same as with small database. When the minimum support reaches 15%, there are no frequent patterns found and the running times of the two algorithms become the same.
Experiment 3: Execution Times for Dataset at Different Minimum Supports (large database):
This experiment uses fixed size database and different minimum support to compare the performance of PLWAP, WAP and GSP algorithms. The algorithms are tested with minimum supports between 0.05% and 15% against one million (1M) record database. The results of this experiment are shown in Table 4 and Figure 7.

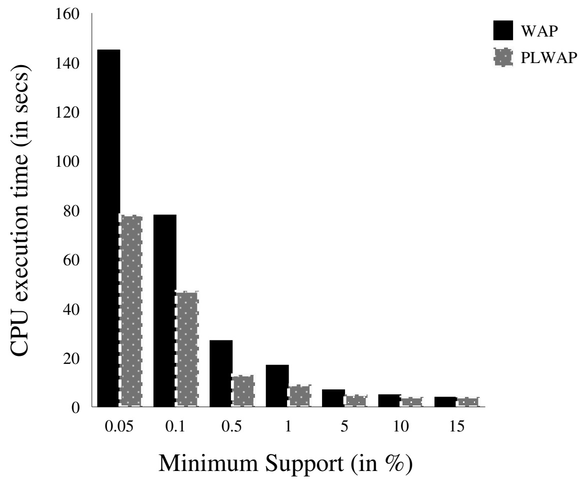

Figure 6: Execution Times Trend with Different Minimum Supports (medium database)

Table 3: Execution Times for Dataset at Different Minimum Supports (medium database)

Alg	Runtime (in secs) at Different Supports(%)						
	0.05 Fp= 2630	0.1 Fp= 1271	0.5 Fp = 20	1 Fp= 114	5 Fp= 23	10 Fp= 10	15 Fp 0
GSP	34275	10021	11742	3320	785	150	4
WAP	145	78	27	17	7	5	4
PLWAP	78	47	13	9	5	4	4

Experiment 3a: Execution Times for Dataset at Different Minimum Supports (large database but very low minimum supports):
This experiment uses fixed size database and different minimum support to compare the performance of PLWAP and WAP algorithms (GSP not included because running times get too big or process is killed). The algorithms are tested with minimum supports between 0.001% and 0.02% against one million (1M) record database. The results of this experiment are shown in Table 5 and Figure 8.

Table 4: Execution Times for Dataset at Different Minimum Supports (large database)

Alg	Runtime (in secs) at Different Supports(%)						
	0.05 Fp= 2646	0.1 Fp= 1269	0.5 Fp = 273	1 Fp= 116	5 Fp= 23	10 Fp= 10	15 Fp 0
GSP	73084	41381	12854	7358	1715	328	9
WAP	387	100	41	27	14	10	9
PLWAP	236	87	28	17	10	9	8

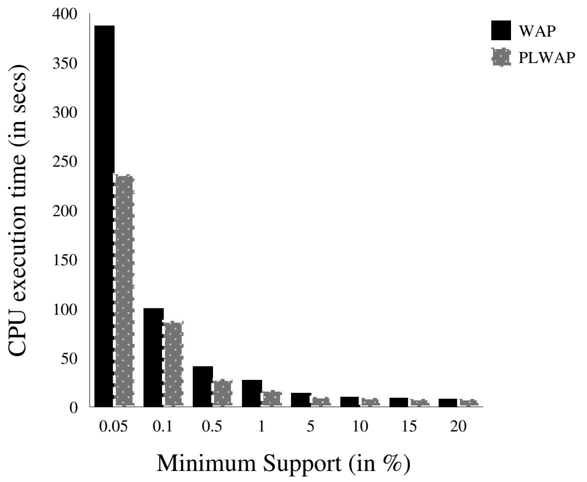

Figure 7: Execution Times for Dataset at Different Minimum Supports (large database)

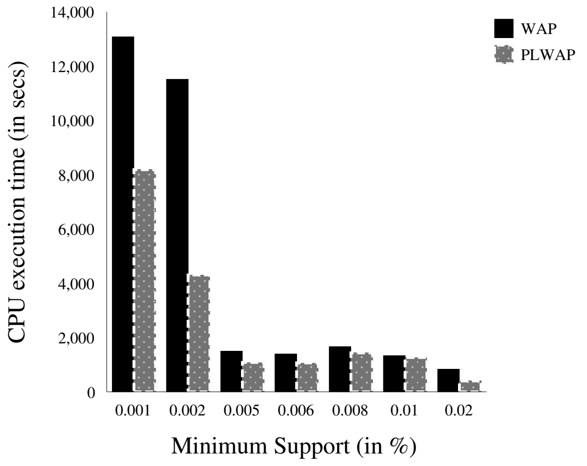

Figure 8: Execution Times for Dataset at Different Minimum Supports (large database and low minsupports)

Experiment 4: Execution times trend with different database sizes (small size database, 2K to 14K):
This experiment uses fixed minimum support and different size database to compare the performance of PLWAP, WAP and GSP algorithms. The algorithms are tested on databases of sizes 2K to 14K at minimum support of 1%. The gain in performance by the PLWAP algorithm is constant across different sizes because the number of frequent patterns in different sized datasets generated by the data generator seem to be about the same at a particular minimum support. The results of this experiment are shown in Table 6 and Figure 9.
Experiment 5: Execution times trend with different database sizes (medium size database, 20K to 200K):
This experiment uses fixed minimum support and different size database to compare the performance of PLWAP and WAP algorithms. The algorithms are tested with database sizes between 20 and 200 thousands at minimum support of 1%. The results of this experiment are shown in Table 7 and

Table 5: Execution Times for Dataset at Different Minimum Supports (large database and low minsupports)

Alg	Runtime (in secs) at Different Supports(%)						
	0.001 Fp= 222K	0.002 Fp= 161K	0.005 Fp = 38174	0.006 Fp= 38174	0.008 Fp= 14931	0.01 Fp= 12228	0.02 Fp 6196
WAP	13076	11512	1494	1396	1661	1329	832
PLWAP	8183	4303	1083	1066	1435	1261	403

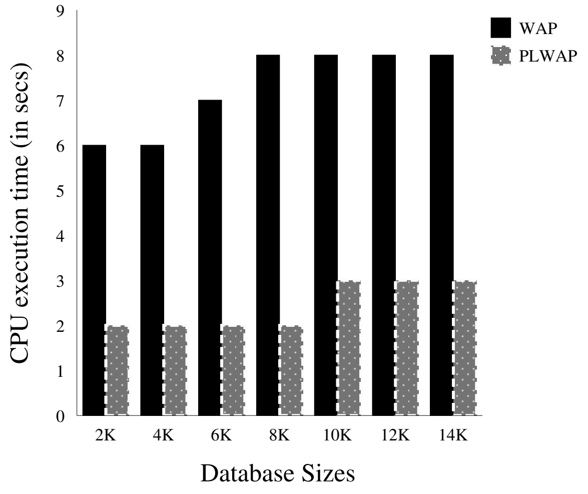

Figure 9: Execution Times Trend with Different Minimum Supports (small database)

Figure 10.
Experiment 6: Execution times trend with different database sizes (large size database, 300K to 900K):
This experiment uses fixed minimum support and different size database to compare the performance of PLWAP, WAP and GSP algorithms. The algorithms are tested with database sizes between 300K and 900K records at minimum support of 1%. The results of this experiment are shown in Table 8 and Figure 11. Since the CPU execution time difference between the WAP and PLWAP from this experiment, seems to diminish as the database size increases, a further experiment was run on larger databases of between one million and 20 million records to check if and when WAP would run faster than the PLWAP. Result of this experiment

Table 6: Execution Times for Different Database Sizes at Minsupport (small database)

Alg	Runtime (in secs) at Different Supports(%)						
	2K	4K	6K	8K	10K	12K	14K
GSP	30	59	91	125	161	193	214
WAP	6	6	7	8	8	8	8
PLWAP	2	2	2	2	3	3	3

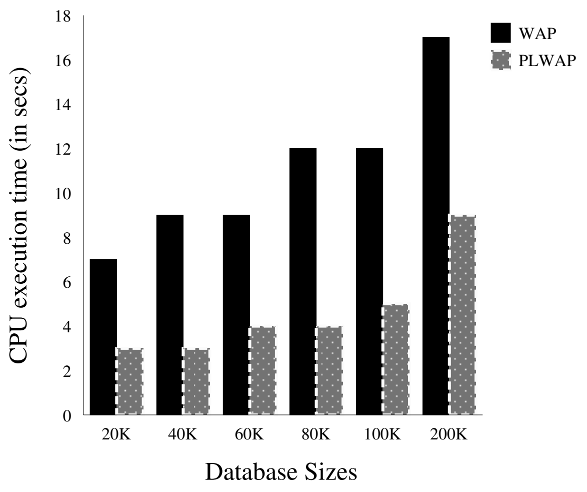

Figure 10: Execution Times Trend with Different Database sizes (medium database)

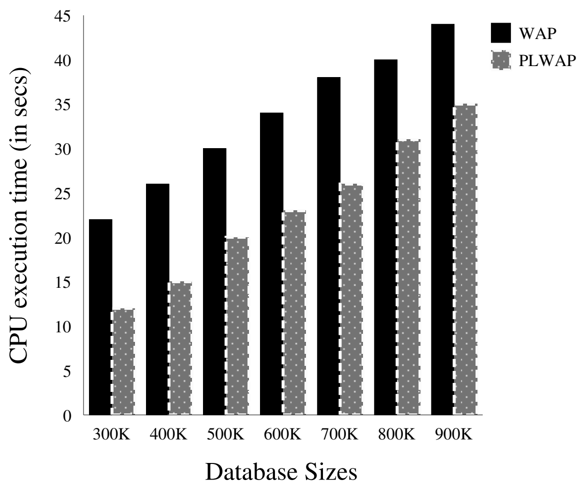

Figure 11: Execution Times Trend with Different Database sizes (large database)

Table 7: Execution Times for Different Database Sizes at Minsupport of 1%(medium database)

Alg	Runtime (in secs) at Different Db sizes(%)					
	20K	40K	60K	80K	100K	200K
GSP	306	639	1015	1338	2004	3320
WAP	7	9	9	12	12	17
PLWAP	3	3	4	4	5	9

Table 9: Execution Times for Different Database Sizes at Minsupport of 1% (larger database)

Alg	Runtime (in secs) at Different DB sizes(%)					
	1M	2M	4M	8M	10M	20M
WAP	23	47	93	186	57	464
PLWAP	23	47	93	186	57	463

on larger databases is given in Table 9. From this experiment, we found that at the same minimum support of 1%, when the database size increases much, there are few or no frequent patterns found because an item needs to appear in about 200 thousand sequential records in the 20 million database to be frequent and that is not very possible in the synthetic data leading to about the same execution times for the two algorithms. However, a run on the same 20M records at a minimum support of 0.1% found 350 patterns and took 560 seconds of CPU time for PLWAP, while the WAP tree algorithm could not complete successfully.

Experiments were also run to check the behavior of the algorithms with varying sequence lengths and number of database attributes, and the PLWAP always runs faster than the WAP algorithm when the average sequence length is less than 20 and there are some found frequent patterns. However, the PLWAP performance starts to degrade when the average sequence length of the database is more than 20 because with extremely long sequences, there are nodes with position codes that are more than "maxint", in our current implementation, we use a number of variables to store a node's position code that are linked together. Thus, managing and retrieving the position code for excessively long sequences could turn out to be time consuming. Our experiment on real data of size 10K having about 2500 attributes with only a few very long sequences of up to 166 items while most of other records are less than 7 items long, reveals PLWAP faster than WAP by a factor of over 11 times at a very low minimum support of 0.05% with their execution times as 449, 5157, at 0.06%, they are 234, 1187 and at 0.07%, 144, 1071 respectively. The execution times of the two algorithms start being the same from minimum support 1% when there are no found frequent patterns. A test on memory usage of the WAP and PLWAP algorithms reveals about the same amount of memory allocated to both programs.

Table 8: Execution Times for Different Database Sizes at Minsupport of 1% (large database)

Alg	Runtime (in secs) at Different DB sizes(%)						
	300K	400K	500K	600K	700K	800K	900K
GSP	5329	7110	8778	12K	13K	15K	17K
WAP	22	26	30	34	38	40	44
PL-WAP	12	15	20	23	26	31	35

5. CONCLUSIONS

This paper discusses the source code implementation of the PLWAP algorithm presented in [7] as well as our implementations of two other sequential mining algorithms WAP [17] and GSP [3] that PLWAP was compared with. Extensive experimental studies were conducted on the three implemented algorithms using IBM quest synthetic datasets. From the experiments, it was discovered that the PLWAP algorithm, which mines a pre-order linked, position coded version of the WAP tree, always outperforms the other two algorithms and is much more efficient. PLWAP improves on the performance of the efficient tree-based WAP tree algorithm by mining with position codes and their suffix tree root sets, rather than storing intermediate WAP trees recursively. Thus, it saves on processing time and more so when the number of frequent patterns increases and the minimum support threshold is low. PLWAP's performance seems to degrade some with very long sequences having sequence length more than 20 because of the increase in the size of position code that it needs to build and process for very deep PLWAP tree. For mining sequential patterns from web logs or databases, the following aspects may be considered for future work. The PLWAP algorithm could be extended to handle sequential pattern mining in large traditional databases to handle concurrency of events. The position code features of the PLWAP tree provides a mechanism for concisely storing small items not represented in the tree for future incremental refreshment of mined patterns. It also enables easy multilevel mining of frequent patterns at detailed (e.g., item level like city level or word level) to more generalized calss level (e.g., country level or book level) through rollup mining with the same tree by providing levelwise pre-order header linkages. This may make it useful in several sequence oriented applications like frequent word usages in collections of documents (books), cell phone call sequence record data, intrusion detection and sensor network mining as well as biological sequence data. A more efficient implementation of the position code management for long sequences would make the PLWAP more scalable.

6. ACKNOWLEDGMENTS

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

7. REFERENCES

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

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

[3] 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*, pages 3–17, 1996.

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

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

[6] C. Ezeife and M. Chen. Mining web sequential patterns incrementally with revised plwap tree. In *Proceedings of the fifth International Conference on Web-Age Information Management (WAIM 2004)*, pages 539–548. Springer Verlag, July 2003.

[7] C. Ezeife and Y. Lu. Mining web log sequential patterns with position coded pre-order linked wap-tree. *International Journal of Data Mining and Knowledge Discovery (DMKD) Kluwer Publishers*, 10(1):5–38, 2005.

[8] J. Han, J. Pei, B. Mortazavi-Asl, Q. Chen, U. Dayal, and M. Hsu. Freespan: Frequent pattern-projected sequential pattern mining. In *Proceedings of the 2000 Int. Conference on Knowledge Discovery and Data Mining (KDD'00)*, pages 355–359, 2000.

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

[10] Y. Lu and C. Ezeife. Position coded pre-order linked wap-tree for web log sequential pattern mining. In *Proceedings of the 7th Pacific-Asia Conference on Knowledge Discovery and Data Mining (PAKDD 2003)*, pages 337–349. Springer, May 2003.

[11] S. Madria, S. Bhowmick, W. Ng, and E. Lim. Sampling large databases for finding association rules. In *Proceedings of the first International Data Warehousing and Knowledge Discovery DaWak99*, 1999.

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

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

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

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

[16] J. Pei, J. Han, B. Mortazavi-Asl, and H. Pinto. Prefixspan: Mining sequential patterns efficiently by prefix-projected pattern growth. In *Proceedings of the 001 International Conference on Data Engineering (ICDE 01)*, pages 214–224, 2001.

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

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

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