

Efficient Mining of Traversal Patterns *

Yongqiao Xiao, Margaret H. Dunham
Department of Computer Science and Engineering
Southern Methodist University
Dallas, Texas 75275-0122
{xiao,mhd}@seas.smu.edu

Abstract

A new problem of mining traversal patterns from Web access logs is introduced. The traversal patterns are defined to keep duplicates as well as consecutive ordering in the sessions. Then an efficient algorithm is proposed. The algorithm is online, which allows the user to see the incremental results with respect to the scanned part of the database. The algorithm also adapts to large databases through dynamic compressions and effective pruning. Finally the algorithm is evaluated through experiments with real Web logs.

1 Introduction

Mining of Web access patterns made by Web users has recently attracted much research interest [5, 10, 7, 4, 19]. Mining of this *clickstream* data can be based on the access made by one person's usage or could be performed by a site examining how all users access the pages there. The former technique could be used to develop user profiles which can assist in improving the precision of search engines. The second function can be used to assist Web masters in creating a more user friendly set of Web pages and to improve the Web server performance by prefetching or pushing. Since we are primarily interested in this type of Web mining, we illustrate this with Example 1.

Example 1 *The Web master at ABC Corp. finds out that a high percentage of users have the following pattern of reference to pages $\langle A, B, A, C \rangle$. This means that a user accesses page A, then page B, then back to page A, and finally to page C. Based on this observation, he determines that a link is needed directly to page C from page B. He then adds this link.*

Notice that in this example, we are interested in sequences of page references including any backward traversals. Here a backward traversal is a reference of a page earlier visited. We are also interested in contiguous page references, not just a set of pages that are visited. Such page references can be obtained from the raw Web logs. A typical log on an Apache Web server contains the source IP address, timestamp, the page accessed, the status and the size of the page, which is shown below.

```
216.250.143.112 - - [29/Feb/2000:02:05:00 -0600] "GET /cse/ HTTP/1.1" 200 3043
```

From the raw web logs, we can extract access sessions, which consist of the list of pages visited by the users in a time interval. Traversal patterns which are accessed frequently by the users are then mined from the access sessions. More details about data preparation for traversal pattern mining are given in Section 6 of this paper.

In this paper we investigate techniques to discover frequently used contiguous sequences of page references, which we call *Maximal Frequent Sequences (MFS)*. Although there has been previous research into the mining

*This material is based upon work supported by the National Science Foundation under Grant No. IIS-9820841

of traversal patterns, the mining of this type of pattern is new. In this paper we first define the problem and then relate it to previous research. In section 3 we provide an overview of our algorithm, called OAT(Online Adaptive Traversal pattern mining), to mine maximal frequent sequences. Two major features of this algorithm, i.e., being both online and adaptive, are described in sections 4 and 5 respectively. In section 6 we investigate the performance of the algorithm as implemented against a real set of traversal data. We then conclude the paper.

2 Problem Statement and Related Work

In this section we first provide definitions needed to more formally state the problem. We then discuss how this problem relates to previous research in the area of mining traversal patterns.

2.1 Problem Statement

Let $E = \{e_1, e_2, \dots, e_m\}$ be a set of literals, called *pages* or *clicks*. A *session* S is an ordered list of pages accessed by a user, i.e., $S = \langle (p_1, t_1), (p_2, t_2), \dots, (p_n, t_n) \rangle$, where $p_i \in E$, t_i is the time when the page is accessed, and $t_i \leq t_{i+1}$ for $i = 1, 2, \dots, n - 1$. Since only the ordering of the access is of our main interest, the access time is omitted, that is, we write a session S as $\langle p_1, p_2, \dots, p_n \rangle$. Associated with each session is a unique identifier, called session ID. The length of a session S is the number of pages in it, which is denoted as $len(S)$. Let database D be a set of such sessions, and the total length of D be $len(D) = \sum_{S \in D} len(S)$.

A *reference sequence*(or *sequence* for short) is a list of consecutive pages in some session. A k -sequence is a reference sequence with k pages. A sequence $X = \langle q_1, q_2, \dots, q_k \rangle$ is a *subsequence* of another sequence $Y = \langle p_1, p_2, \dots, p_n \rangle$, if $k \leq n$ and $\exists i(1 \leq i \leq n - k + 1)$ such that $p_i = q_1, p_{i+1} = q_2, \dots, p_{i+k-1} = q_k$. For example, sequence $\langle A, B \rangle$ is a subsequence of sequence $\langle A, B, C, D \rangle$, while $\langle A, C \rangle$ is not, since A and C are not consecutive in $\langle A, B, C, D \rangle$. A sequence X *occurs in* a session S if the sequence is a subsequence of the session. If X is a subsequence of Y , we also say that Y is a *supersequence* of X .

The *frequency* of a sequence X is the total number of occurrences of X in all sessions in D , which is denoted as $freq_D(X)$. When the database D is obvious from the context, we write the frequency simply as $freq(X)$. The percentage of the frequency, called *support*, is defined as the ratio of the frequency to the total length of D , i.e., $sup_D(X) = \frac{freq(X)}{len(D)}$. Here the *length* of the database is the sum of each specific page occurrence listed in all sessions. Alternatively it can be thought of as the number of clicks or pages (including duplicates) visited. Note that we don't define the support as the ratio of the frequency to the total number of sessions in D , because a sequence may occur more than once in a session due to backward traversals and/or reloads(refreshes). This definition of support also has the downward closure property as for itemsets[2], that is, $sup(X) \leq sup(Y)$, if Y is a subsequence of X .

A sequence X is called *frequent* if $sup(X) \geq s_{min}$, where s_{min} is the minimum support threshold given by the user. A frequent sequence X is *maximal*, if it is not a subsequence of any other frequent sequence. Each such maximal frequent sequence represents a traversal pattern. Our objective of mining traversal patterns is to find all maximal frequent sequences in the database given some minimum support threshold. Example 2 illustrates the concepts of maximal and frequent.

Example 2 Given database $D = \{\langle A, B, C, D, E, D, C, F \rangle, \langle A, A, B, C, D, E \rangle, \langle B, G, H, U, V \rangle, \langle G, H, W \rangle\}$. The first session has backward traversals, and the second session has a reload/refresh on page A. Here $len(D) = 22$. Let the minimum support be: $s_{min} = 0.09$. This means that we are looking at finding sequences that occur at least 2 times. There are two maximal frequent sequences: $\langle A, B, C, D, E \rangle$ and $\langle G, H \rangle$. Notice that both sequences occur two times.

2.2 Related Work

Although there has been much previous work investigating the mining of Web traversal data, the type of patterns uncovered in our approach is new. The three major features of a maximal frequent sequence

are: duplicates(backward traversals and refreshes/reloads), contiguous, and maximal. We do not remove duplicate page references in the sequence as the existence of these duplicate references is itself meaningful as shown in Example 1. We are only interested in contiguous page references as they can be useful for prefetching and caching purposes. Suppose we wanted to predict what pages would be referenced next. In this case we need to have the exact sequence of previous pages (not a truncated or partial sequence). The maximal property is used primarily to reduce the number of meaningful patterns discovered. We do not intend to imply that patterns which remove duplicates or which look at sequences of page references which are not contiguous are not important. Rather, by looking at these features we can detect useful information not found if they are not used. However, the algorithm which we develop in this paper could be similarly used for uncovering patterns without duplicates and where page references are not contiguous.

The work closest to ours is the maximal frequent forward sequences proposed in [5]. The difference lies in that it first transforms each raw session into forward sequences(i.e., removes the backward traversals and reloads/refreshes), from which the traversal patterns are then mined using improved level-wise algorithms[16]. For example, for the session $\langle A, B, A, C \rangle$ in Example 1, the resulting forward sequences are $\langle A, B \rangle$ and $\langle A, C \rangle$. After such transformation, however, some useful information may be lost, e.g., from the two forward sequences, we could not tell whether a direct link to page C from page B is needed, as shown in Example 1.

Association rules, which were originally proposed for market basket data[1, 2], have also been applied to Web access logs [10, 7]. A page is regarded as an item, and a session is regarded as a transaction. As required by association rules, both duplicates and ordering are ignored. For example, the session $\langle A, B, A, C \rangle$ is simply seen as $\{A, B, C\}$.

Sequential patterns[20] have also been applied to Web access logs [4, 19, 17]. The sessions are ordered by the user id(corresponds to customer id), and the patterns are across the sessions. As with association rules, the duplicates are ignored.

Episodes, which were originally proposed for telecommunication alarm analysis [12], can also be applied to Web logs[13]. All pages(corresponds to events) are ordered by their access time, and usually the users need not be identified(i.e., no sessions).

A detailed comparison is shown in Table 1.

	Ordering	Duplicates	Consecutive	Maximal	Support
Association Rules	N	N	N	N	$\frac{freq(X)}{\# transactions}$
Episodes	Y ¹	N	N	N	$\frac{freq(X)}{\# time windows}$
Sequential Patterns	Y	N	N	Y	$\frac{freq(X)}{\# customers}$
Forward Sequences	Y	N	Y	Y	$\frac{freq(X)}{\# forward sequences}$
Maximal Frequent Sequences	Y	Y	Y	Y	$\frac{freq(X)}{\# clicks}$

Table 1: A Comparison

3 Overview of Algorithm

We propose an online adaptive algorithm, OAT, to detect maximal frequent sequences. The online property ensures that the sessions can be examined and mined incrementally as they arrive. We assume that the data is coming continuously and as soon as sessions are detected they are fed into the algorithm. To be online, the algorithm adopts a suffix tree data structure. The adaptive property allows the algorithm to use the available main memory efficiently. If there is not enough main memory then the algorithm reduces its usage requirements and continues. To be adaptive, the algorithm uses two pruning techniques and compresses the suffix tree.

3.1 Suffix Tree Introduction

A *suffix tree* is a trie-like data structure representing all suffixes of a sequence, and the nodes having a single child are collapsed. A suffix tree has the following characteristics[14]:

- Each internal node except the root has at least two children.
- Each edge represents a nonempty subsequence.
- The subsequences represented by sibling edges begin with different symbols.

A suffix tree can be constructed from a sequence in time and space linear in the length of the sequence[24, 14]. With the help of a suffix tree, not only is it efficient to find any subsequence in a sequence, but also efficient to find the common subsequences among multiple sequences. Suffix trees are used extensively in string matching[24, 14, 22] and biological sequence matching[23, 3].

To build a suffix tree for multiple sequences(e.g., the sessions in a database), a unique symbol is appended to each session and the database is regarded as one big sequence. The resulting suffix tree is called a *generalized suffix tree(GST)* [9, 23, 3].

Example 3 The complete suffix tree for the database of sessions in Example 2 is shown in Figure 1. The unique symbol \$ is appended to each session. To differentiate \$ from the symbols in a sequence(pages in a session), \$ can be implemented as negative numbers, while the pages as positive numbers. Leaf nodes are represented by circles, and internal nodes by rectangles. The subsequence represented by each edge is shown by two numbers $x : y$, where x is the position of the beginning symbol of the subsequence in the big sequence, and y is the length of the subsequence (or the position of the ending symbol of the subsequence in the big sequence for simplicity). Each internal node represents a sequence of symbols that start at the root. A suffix link is found at each internal node. The suffix link at an internal node u points to the node that represents the longest suffix of the subsequence represented by note u , i.e., if there is a suffix link from node u to v , and the subsequence represented by u is $\langle p_1, p_2, \dots, p_n \rangle$, then the subsequence represented by v must be $\langle p_2, \dots, p_n \rangle$. Suffix links are usually used to facilitate the construction of a suffix tree. Suffix links are shown as dashed curves, and the suffix links which point to the root are omitted.

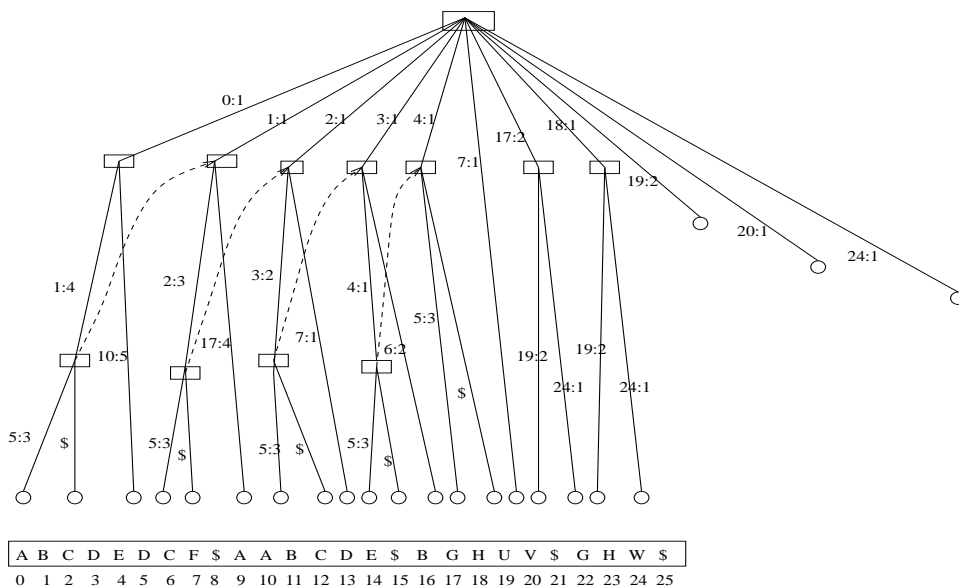


Figure 1: The Full Suffix Tree

D	the database of sessions
n	number of sessions in D
N	total length of all sessions in D
I	number of internal nodes in the suffix tree
α	average load factor of hashtables on internal nodes

Table 2: Notations for suffix tree

It is well known that the suffix tree takes linear main memory with respect to the total length of the big sequence. Here we analyze the main memory requirement in bytes on an ordinary machine. We assume that an integer takes four bytes on the machine as does a pointer. Using the notations in Table 2, we have the following result.

Lemma 1 *The suffix tree for a database D of sessions requires $8I + \frac{10(I+N-1)}{\alpha} + 4(N+n) = (8 + \frac{10}{\alpha})I + (4 + \frac{10}{\alpha})N + 4n - \frac{10}{\alpha}$ bytes of main memory.*

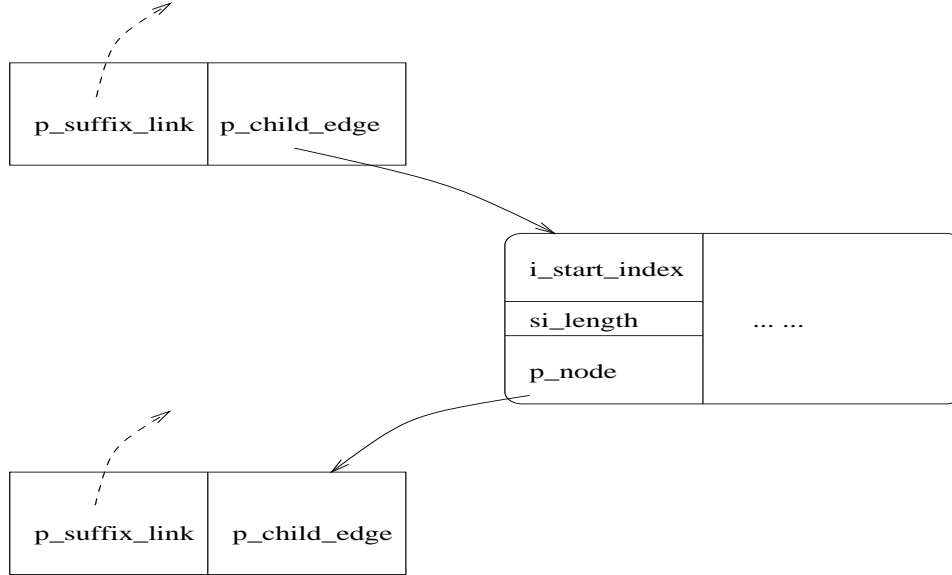


Figure 2: Data Structure for Suffix Tree

Proof. We use the data structure shown in Figure 2 to represent the suffix tree. There are three parts of main memory requirement: nodes, edges, and database.

- database. As described earlier in this section, a unique symbol is appended to each session in the database. The actual length of the database stored in main memory is then $N + n$, shown in the lower part of Figure 1. Without loss of generality, the pages in the sessions are represented as integers. Thus the main memory requirement for the database is $4(N + n)$.
- nodes. Each internal node (shown as a rectangle in Figure 1) requires 8 bytes: 4 bytes for the suffix link pointer, and 4 bytes for the pointer to the hashtable for the edges from this node to its children (the pointer is required because the number of child nodes can not be determined beforehand and thus the hashtable has to be dynamic). We don't need to explicitly represent the leaf nodes (shown as circles in Figure 1), since they have neither suffix link pointers nor child nodes. Therefore all nodes require $8I$ bytes of main memory.

- edges. Each edge requires 10 bytes: 4 bytes for the integer index to the database array at which is the beginning page on the edge, 2 bytes for the number of pages on the edge, and 4 bytes for the pointer to the child node. The number of edges is $I + N - 1^2$. Since the edges starting from each internal node are stored in a hashtable, let α be the average load factor of all hashtables on the nodes (a load factor of a hashtable indicates how full the hashtable is). Therefore the edges take $\frac{10(I+N-1)}{\alpha}$ bytes of main memory. Note that (1) an integer index is enough if the database can be put in the main memory. For a database longer than 2^{32} , an integer index should still be enough if the compression is done as describe in the next section. (2) two bytes are used for the length of sequence on an edge, which should be enough even for agents or robots. If the user is not interested in such possibly too long sequences, one byte could also be used to limit the maximum length of sequences to be found.

□

Since in a suffix tree each internal node (except the root) must have at least two children, we have $I \leq N$. Let $\alpha = 0.8$, which allows a reasonable tradeoff between time and space costs for hashtables. The main memory requirement for the suffix tree is bounded by $20I + 16N + 4n < 36N + 4n$. So on an ordinary PC with 64M bytes of main memory, the algorithm can handle a database of length 1.5 million clicks. This is the worst case. Usually we expect $I \ll N$, since a frequent sequence shares the same internal nodes.

However, for large databases which have more than millions (or even billions) of clicks, there will be insufficient main memory to hold the whole suffix tree. We propose two pruning techniques and compression of the suffix tree to handle such large databases. They are described in the Section 5.

3.2 The Algorithm

The algorithm OAT incrementally maintains the suffix tree with inclusion of a new session. When the user interrupts, the algorithm outputs the current maximal frequent sequences to him or her (the outputs are actually a superset of the current maximal frequent sequences due to compressions). When there is insufficient main memory, the suffix tree is compressed, and then the algorithm continues as usual. When all sessions are scanned once, the suffix tree consists of all potentially frequent sequences. The algorithm then checks whether there are sequences that are not completely counted. If all sequences are counted, the algorithm outputs the maximal frequent sequences and finishes. Otherwise, an additional scan is needed to get the exact counts of all sequences. Also some sequences may eventually turn out to be infrequent and are then pruned. The algorithm is sketched in Algorithm 1. The online and the adaptive features are described in the following two sections, respectively.

Algorithm 1

Input:

S_1, S_2, \dots, S_n : sessions.
 s_{min} : minimum support threshold.
 M : main memory size.

Output:

all maximal frequent sequences (MFSs)

Method:

- (1) $ST =$ an empty suffix tree;
//first scan
- (2) **for** i from 1 to n **do**
//if insufficient main memory with inclusion of S_i , compress the suffix tree using frequent sequences.
- (3) **if** $mem(ST \cup S_i) > M$ **then**
- (4) $ST = OAT_compress(ST)$;
- (5) **endif**
//update the suffix tree with inclusion of S_i

²in a tree, the number of edges is one less than the number of nodes, and in the suffix tree, there are I internal nodes and N leaf nodes

```

(6)   ST = update(ST, Si);
(7)   if interrupted by the user then
      //do a depth-first traversal of ST and output the MFSs.
(8)   MFS_output_depth_first(ST.root);
(9)   endif
(10)  endfor
      //second scan
(11)  if there are sequences not completely counted then
(12)   count them in an additional scan.
(13)  endif
(14)  output the MFSs in the suffix tree.

```

4 Online Pattern Mining

The online feature of the algorithm includes online construction of the suffix tree and online extraction of patterns, which are described in the following two subsections.

4.1 Online Suffix Tree Construction

More recently, the construction of suffix trees are made online[22], that is, the online algorithm processes the big sequence(database) symbol by symbol from left to right, and always has the suffix tree for the scanned part of the big sequence(database) ready to show to the user if needed. For the database in Example 2, if the sessions are read in the order as listed, the suffix trees after scanning one session/two sessions/three sessions are shown in Figure 4.1(a), b) and c), respectively.

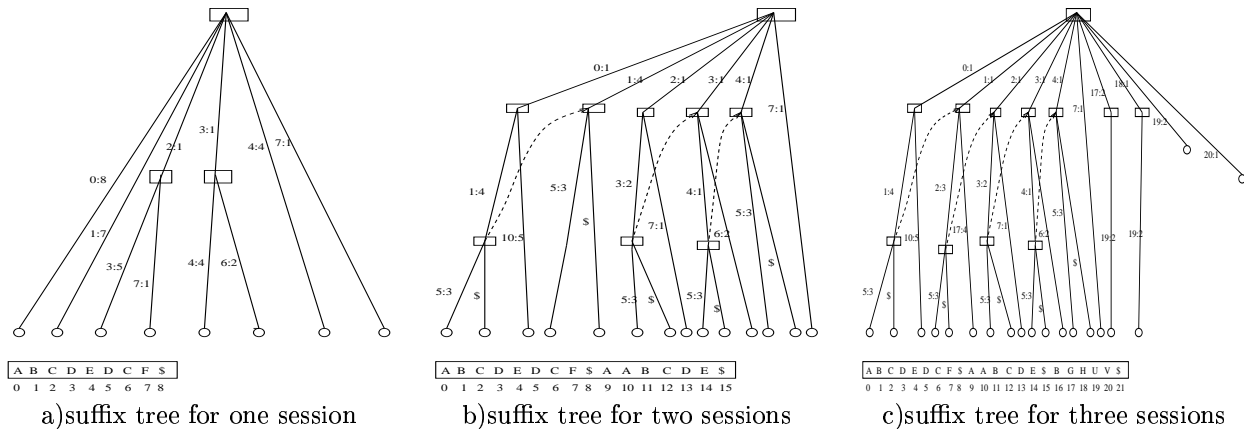


Figure 3: Online Construction of Suffix Tree

The linear construction algorithm requires constant transition time from one node to one of its children nodes in the suffix tree. If the alphabet(i.e., the set of literals E) is small, an array of fixed size (equal to the alphabet size) can be used on each node, e.g., for DNA sequence matching, the alphabet is $\{A, C, G, T\}$, an array of size 4 can be used on each node. However, this fixed array method does not work for the clickstream data, since the number of distinct pages on a Web server is usually very large and may even not be fixed. To overcome this, we use a dynamic hashtable on each internal node for transition to its children.

Here we assume that there is always enough main memory to hold the suffix tree and the database. For large databases, this may not be true. We propose two pruning techniques and compression of the suffix tree,

which will be described in the next section. With such compressions and pruning techniques, it is shown in Lemma 3 that the online construction algorithm still works.

4.2 Online Pattern Extraction

During the construction of the suffix tree, the user may want to see the current MFSs with respect to the sessions scanned. Let the current session scanned be S_i and the scanned part of the database be $D^i = \cup_{j=1}^i S_j$. A sequence X is frequent with respect to D^i if $freq_{D^i}(X) \geq len(D^i) \times s_{min}$.

The straightforward way has three steps: first the frequencies of all sequences are obtained by a traversal of the suffix tree. Note that the frequency of the sequence corresponding to a node is the total number of leaf nodes under the node, i.e., the number of leaf nodes in the sub-tree with the node as the root. Then the frequent sequences are found. Last the maximal frequent sequences are found from the frequent sequences. There are obvious drawbacks about this approach: (1) all frequent sequences need to be explicitly found and stored, which requires additional main memory; (2) to find the maximal frequent sequences from the frequent sequences requires quite a bit overhead.

The above approach can be improved by using a depth-first traversal. During the traversal over each node, we accumulate the frequency on the node. If any of its children nodes corresponds to a frequent sequence, then we don't need to output the sequences corresponding to this node and its ancestor nodes. The approach is sketched in Function 1. The function actually outputs all the MFSs and their suffixes. Note that it is possible to output only the MFSs by traversing the child nodes in the order of their depths (the root has depth 0, and the depth of any child node is one more than that of its parent), but this will require additional space, e.g., for the heights. As analyzed later the suffix tree already requires a lot of main memory, so we choose to simply output the frequent sequences by a simple depth-first traversal.

Function 1

Function MFS_output_depth_first(suffix tree node: nd):frequency at the node

```

    global variables:  $s_{min}$  and  $len(D^i)$ 
(1)   $freqChild = freqNode = 0$ ;
(2)   $maximal = true$ ;
(3)  for each child node  $c$  of  $nd$  do begin
(4)    if  $c$  is a leaf node then
(5)       $freqNode ++$ ;
(6)    continue; //for next child node
(7)    endif
(8)     $freqChild = MFS\_output\_depth\_first(c)$ ;
(9)     $freqNode = freqNode + freqChild$ ;
(10)   if  $freqChild \geq s_{min} \times len(D^i)$  then
(11)      $maximal = false$ ;
(12)   endif
(13) endfor
(14) if  $maximal == true$  and  $freqNode \geq s_{min} \times len(D^i)$  then
(15)   print the pattern starting from the root to node
(16) endif
(17) return  $freqNode$ ;

```

If the user wants to see the MFSs after each session as in [8], the depth-first traversal may not be efficient, since it traverses the entire suffix tree each time. In this case, we run the algorithm in a different mode, which automatically outputs the MFSs after each session. Instead of computing the MFSs from scratch after each session, we keep the previous MFSs in main memory and update them incrementally with inclusion of a new session. Let the current session be S_i and MFS_{i-1} be the MFSs with respect to the first $i - 1$ sessions, i.e., D^{i-1} . The task is to compute MFS_i from MFS_{i-1} with inclusion of S_i .

There are two cases: (1) some sequences in MFS_{i-1} are kept in MFS_i . This happens when their new support is not less than the threshold. These sequences can be easily identified by checking their new frequency (old frequency in the first $i - 1$ session plus the number of occurrences in S_i). For those sequences that become infrequent, however, we still need to check whether their subsequences are $MFSs$, and include them in MFS_i if they are $MFSs$. (3) some infrequent sequences become $MFSs$. This could happen only when the infrequent sequences occur in S_i . Therefore, we only need to check these infrequent sequences that are subsequence of S_i . This can be done during the updating of the suffix tree as follows: whenever a leaf node is added, the frequency of all nodes from the root to the leaf node increases by one. We start from the parent node of the leaf node and check whether the sequence spelled out by each node on the path was infrequent but becomes frequent. We stop when such node is found or reach a frequent node (the root is always frequent). Whenever we find such a sequence, we also need to delete from MFS_{i-1} that are subsequences of this sequence.

Note that we are not actually restricted to output the $MFSs$ only at the end of a session. Instead the algorithm could produce the $MFSs$ in the middle of a session, since (1) the suffix tree construction is online and (2) the support is defined as the ratio to the length of the database. Sometimes it may be beneficial to allow outputting the $MFSs$ in the middle of a session. For example, if a session is dynamically tracked using cookies, the session may last for a while before it times out. Instead of waiting until the session ends, we could feed the available part of the session to the mining algorithm and see the $MFSs$. However, there are usually more than one session outstanding on a Web server. To feed multiple sessions to the algorithm at the same time requires concurrent construction of suffix trees, which is of our future interest.

5 Adaptive Pattern Mining

Whenever there is insufficient main memory, the algorithm reduces its main memory requirement as shown in function OATCompress in Algorithm 1. The reduction is achieved in three ways: local pruning, cumulative pruning, and compression of the suffix tree. We call each such main memory reduction as a compression. We also call the scanned part of the database between two consecutive compressions as a partition. The notations in Table 3 are used in the following description.

D	the database of sessions
m	number of partitions (compressions)
P_i	the i -th partition ($1 \leq i \leq m$)
D^i	the first i partitions, i.e., $D^i = \sup_{j=1}^i P_j$
FP_i	the frequent patterns in the i -th partition, i.e., $\{X freq_{P_i}(X) \geq s_{min} \times len(P_i)\}$
CP^i	the candidate patterns kept in the suffix tree after the i -th compression
GP	the frequent patterns in D , i.e., $GP \supseteq MFS$

Table 3: Notations for traversal patterns

5.1 Local Pruning

By local pruning, we prune away from the suffix tree those sequences that are not frequent in partition P_i . Local pruning is based on a similar property as in the PARTITION[18] algorithm for association rules. The property states that a frequent sequence in D must be frequent in at least one partition, i.e., $\sup_{i=1}^m FP_i \supseteq GP$. Using this property, we can prune away the sequences that are not frequent in the current partition P_i , since these sequences will be added back into the suffix tree if they are eventually frequent.

The local pruning favors a uniform distribution of the database partitions, i.e., the frequencies of the sequences in each partition are evenly distributed. If the database is skewed, however, the local pruning may not be very powerful. For example, suppose each partition has a distinct set of frequent sequences. In this extreme case, the number of sequences remaining in the suffix tree keeps increasing after each compression,

and most of the sequences may not turn out to be frequent in the whole database. More severely, given a fixed amount of main memory, with the size of the suffix tree increasing, the length for the next partition will decrease. This results in more frequent sequences in the next partition added into the suffix tree, which in turn causes the next next partition to be even smaller. This is the partition size problem. Generally a partition should be large enough to capture enough frequency for the sequences to be frequent in the partition, which is much like the sample size problem[21](a sample can be viewed as a partition). Note that unlike the database partitions of equal size in [18], the length of database partitions in our case varies depending on the available main memory.

The Web log data, unfortunately, tend to be skewed. For example, on a news Web site, the hot topics(represented by traversal patterns) change all the time. On the other hand, an e-retail store may have seasonal patterns due to that the customers usually have seasonal buying patterns. To handle the skew and partition size problem, we introduce a more powerful pruning technique, cumulative pruning, which is described in the next subsection.

5.2 Cumulative Pruning

Cumulative pruning uses the cumulative frequency in the scanned part of the database for pruning. By cumulative pruning, we prune away from the suffix tree the sequences that are not frequent in the first i partitions, i.e., $freq_{D^i}(X) < s_{min} \times len(D^i)$.

To employ the cumulative pruning, we need to add one more integer field to each internal node of the suffix tree(leaf nodes always have frequency equal to 1, and thus can be omitted). The field, $freqCumu$, is to keep track of the cumulative frequency of the sequence corresponding to each node. Note that the field is not updated during the construction of the suffix tree. Instead it is updated only when a compression occurs. However, $freqCumu$ may not be directly available for some sequences. For example, suppose a sequence X is infrequent in partition 1, but frequent in partition 2. Since X is infrequent in partition 1, it will be pruned at the first compression. Thus during the 2nd compression, we do not have the cumulative frequency for X . Note that X will be kept in the suffix tree if only local pruning is used, since X is frequent in partition 2.

In the case that $freqCumu$ is not directly available, we estimate an upper bound for the frequency in the missing part of the database. For the above example, since we know that X is infrequent in partition 1, the upper bound for its frequency in partition i can be simply estimated as $S_{min} \times len(P_1) - 1$. In general, for a sequence X that is infrequent in the first i partitions, the upper bound for the frequency of X in D^i is simply $ubound_{D^i}(X) = s_{min} \times len(D^i) - 1$. The upper bound is estimated when a sequence is first determined to be frequent and added into the suffix tree. The cumulative frequency is then the sum of the upper bound and the frequency in the later partitions.

The cumulative pruning using the simple upper bound estimation works similarly to the First Global Anti-Skew in [11]. Yet it is biased towards the beginning partitions of the database. Consider the extreme case again, i.e., each partition has distinct sets of frequent sequences. The frequent sequences in the beginning partitions go through the remaining partitions, and are more likely to be pruned by the cumulative pruning. Thus, the errors of the simple upper bound estimation have less impact on the actual cumulative frequencies. While for the frequent sequences in the ending partitions, the impact of the estimation errors are huge because there are not enough partitions to go through, especially for those added in the last partition no cumulative pruning can be employed on them.

To alleviate the bias for the ending partitions, we improve the upper bound estimation by storing more information in main memory. Instead of pruning away from the suffix tree all infrequent sequences either by local pruning or accumulative pruning, we keep the edges starting from the root to its children even if the corresponding sequences are infrequent. Let $X = \langle p_1, p_2, \dots, p_m \rangle$ be a sequence that is infrequent in the first i partitions. The upper bound for its frequency in D^i can be refined as: $ubound_{D^i}(X) = \min\{s_{min} \times len(D^i) - 1, freq_{D^i}(\langle p_1 \rangle), \dots, freq_{D^i}(\langle p_m \rangle)\}$. The above formula follows directly from the downward closure property of the support of sequences. Then if the cumulative frequency of X is less than the required minimum frequency, X can be pruned away. The new upper bound is usually tighter than the simple one. For the above example, let the minimum frequency required for P_1 be 50. Table 4 shows the estimated upper bounds for three sequences. Note that frequencies of A and B are stored in the suffix tree,

since they are both children of the root.

sequence	real freq in P_1	simple ubound	refined ubound
A	1	49	1
B	2	49	2
$\langle A, B \rangle$	1	49	1

Table 4: Upper Bound Estimation

It is worth mentioning that cumulative pruning can not be used alone, instead cumulative pruning should always be used together with local pruning. Example 4 illustrates how they are used together.

Example 4 Suppose there are two partitions. For a sequence X , there are four cases regarding whether it is frequent in partition 1 or partition 2. For each case, different actions are taken during the compressions shown in Table 4. An interesting one is case 2, i.e., X is infrequent in partition 1 but frequent in partition 2. During the first compression, it is pruned away from the suffix tree by local pruning (suppose X does not correspond to a child node of the root). During the second compression, the frequency of X in partition 1 is estimated using the refined upper bound. If the cumulative frequency in both partitions is less than the required minimum frequency, X is pruned away by cumulative pruning. Otherwise, X is still kept in the suffix tree.

	partition 1		partition 2	
	frequent	action	frequent	action
case 1	No	local pruning	No	local pruning
case 2	No	local pruning	Yes	cumulative pruning
case 3	Yes	keep	No	cumulative pruning
case 4	Yes	keep	Yes	keep

Table 5: Upper Bound Estimation

Lemma 2 shows that combining cumulative pruning and local pruning guarantees that no MFSs will be missing at the end of the first database scan.

Lemma 2 Let m be the number of partitions (compressions), FP_i be the frequent patterns in the i -th partition, CP^i be the candidate patterns kept in the suffix tree after the i -th compression, GP be the frequent patterns in D . We have $GP \subseteq CP^m$, if local pruning and cumulative pruning are used together.

Proof. By induction on the number of partitions. If there is only one partition, it is obviously true. Suppose the lemma holds for $(m - 1)$ partitions, i.e., $GP^{m-1} \subseteq CP^{m-1}$, where GP^{m-1} is the set of frequent patterns in D^{m-1} . For the whole database of m partitions, we can regard it as one big partition (D^{m-1}) and the other partition P_m . Let the two partitions be $P_{(1)}$ and $P_{(2)}$, and the frequent patterns in the two partitions be $FP_{(1)}$ and $FP_{(2)}$, respectively. Note that $FP_{(1)} = GP^{m-1}$. For any frequent pattern X in GP , X must be in either $FP_{(1)}$ or $FP_{(2)}$ or both. The three cases correspond to cases 3, 2, 4 in Table 5, respectively. If $X \in FP_{(1)}$ (cases 3 and 4), X will also be in CP^{m-1} from the induction hypothesis. Since $freq_{CumuD}(X)$ is an upper bound for its real frequency $freq_D(X)$, X will not be pruned away from the suffix tree by local pruning or cumulative pruning, i.e., $X \in CP^m$. Similarly, if $X \in FP_{(2)}$, X will also be in CP^m . Therefore, we have $GP \subseteq CP^m$. \square

We can also take advantage of the bias toward the beginning partitions. Since the frequent sequences in the beginning partitions will go through the remaining partitions, we can set a lower support, say $0.9 \times s_{min}$,

for the beginning partitions to allow more sequences to go through. Using a lower support for the beginning partitions will also help to alleviate the swapping effect (pruned in one partition and added in the later partition).

5.3 Compression of the Suffix Tree

The compression function first prunes away from the suffix tree the sequences that can be pruned by local pruning or cumulative pruning. This can be done similarly as the function MFS-output-depth-first through a depth-first traversal over the suffix tree. Note that the edges starting from the root are kept for the cumulative pruning.

Then the compression function tries to compress the scanned part of the database D^i in the database buffer. The idea is that since only the sequences that are likely to be frequent are kept in the suffix tree, we don't need to keep the infrequent sequences in D^i . Likewise, a frequent sequence occurs in multiple sessions, but only one occurrence needs to be kept in D^i . To compress D^i using the frequent sequences, we first find out the positions in D^i that have corresponding frequent sequences in the suffix tree by traversing the suffix tree. Then we compute the offset for each such position that needs to shift left. Last we traverse the suffix tree again to update the edges, and shift the data in D^i to the left by the corresponding offsets. Note that additional main memory is required to store the bitmap and the offsets, both of which are of length of the database buffer.

Function 2

Function OAT-compress(suffix tree: ST_i)

global variables: s_{min} and $len(D^i)$

additional main memory required for bitmap and offsets.

//prune away from the suffix tree the patterns that are unlikely to be frequent with respect to D^i .

(1) *OAT-prune-depth-first($ST_i.root$);*

(2) *clear the bitmap and offsets array.*

//traverse the remaining ST_i , set the bitmap for the positions that have corresponding edges in ST_i .

(3) **for** each edge e in ST_i **do**

(4) **for** i from $e.i_start_index$ to $e.si_length$ **do**

(5) $bitmap[i] = 1$;

//compute the offsets for each position with the bit set that needs to

//shift left, and save them in the integer array.

(6) $count=0$;

(7) **for** i from 0 to buffer sizedo

(8) **if** $bitmap[i] = 0$ **then**

(9) $count++$;

(10) **else**

(11) $offsets[i] = count$;

(11) **endif**

(12) **endfor**

//traverse the suffix tree ST_i again to update the field $char_start_index$ for each edge using the above offsets.

(13) **for** each edge e in ST_i **do**

(14) $e.i_start_index = offsets[e.i_start_index]$;

//shift the data in the buffer to the left using the offsets.

(16) **for** i from 0 to buffer sizedo

(17) **if** $offsets[i] > 0$ **then**

(18) $buffer[i-offsets[i]] = buffer[i]$;

The compressed suffix tree of the complete suffix tree in Figure 1 is shown in Figure 5.3a). After compressions, the construction of the suffix tree with inclusion of new sessions can work the same as without compressions, shown in Lemma 3. Note that after compressions some internal nodes may have only one

child, which seems to violate one property of suffix trees. Actually this does not impact the correctness of the construction algorithm as shown below.

Lemma 3 *After compressions as above, the online suffix tree construction algorithm works the same with new sessions included.*

Proof. First we observe that the remaining suffix links after compressions are well maintained, i.e., all suffix links in the pruned suffix tree point to the right node. This is due to the downward closure property, that is, all subsequences of a frequent (or potentially frequent) sequence are frequent (or potentially frequent), and in the suffix tree after compressions are all the frequent (or potentially frequent) sequences. Note that the child nodes of the root are kept for cumulative pruning, and the suffix links on these nodes all point to the root, thus they are still valid. Second, the online construction algorithm [22] starts from the root whenever it scans a new session due to the unique identifier appended to each session. In addition, the algorithm relies on the suffix links to create new edges or split existing edges. \square

If there is still not enough main memory, we may keep only the current *MFSs* in the suffix tree. The compressed suffix tree using *MFSs* is shown in Figure 5.3b), which is much smaller than the compressed with all frequent sequences. However, after compressing with *MFSs* some suffix links may become invalid, since the nodes corresponding to frequent but not maximal sequences are removed, and some edges may be merged as shown in Figure 5.3b). Therefore, in order for the construction algorithm to work properly with new sessions included, the suffix links should be adjusted accordingly. Another problems with such compression using *MFSs* is that when one of the current *MFSs* become infrequent we should check whether their subsequences are *MFSs*, while the frequencies of their subsequences may not be available in the suffix tree. For example, when $\langle A, B, C, D, E \rangle$ becomes infrequent, we need to check whether its subsequences $\langle A, B, C, D \rangle$ or $\langle B, C, D, E \rangle$ are frequent. However their frequencies are not kept in the suffix tree. Even though we can estimate the frequencies using the upper bound as above, there may be too much overhead, since we need to continue checking its sub-subsequences and so on if the subsequences are found to be infrequent.

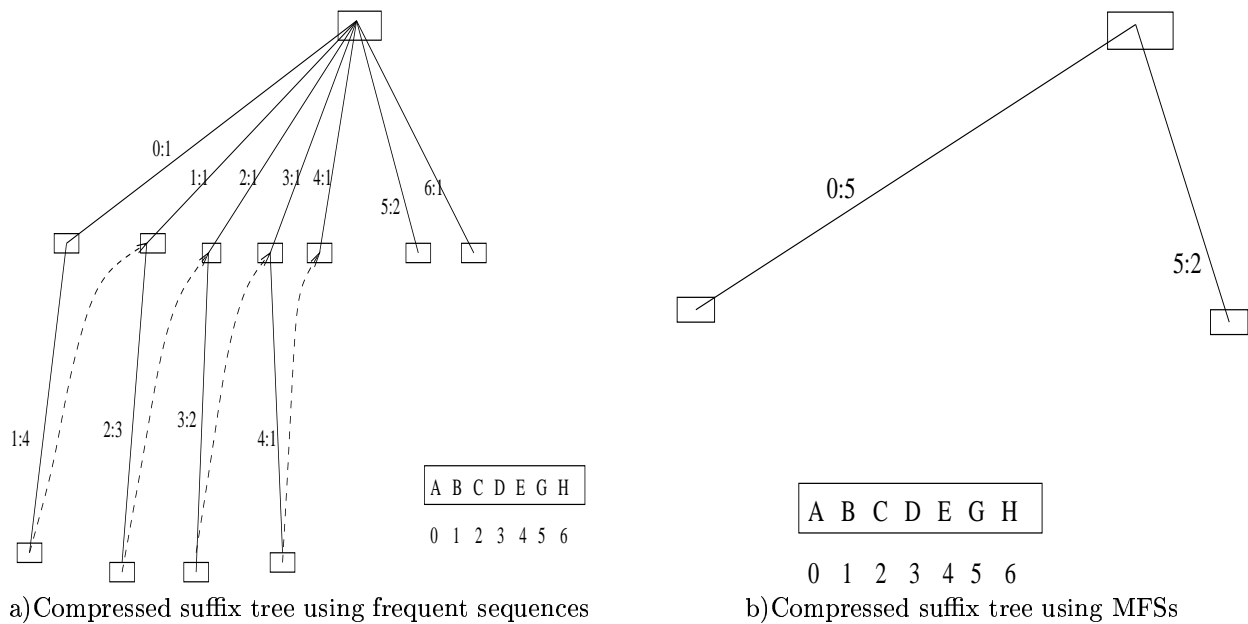


Figure 4: Compression of Suffix Tree

6 Experimental Results

We report on the experiments using the Web logs collected on our School of Engineering Web server. We first describe the data preparation, then report the response time and the size of suffix trees. All experiments were conducted on a Dell PC with a 650MHz Pentium III processor and 128 MB of RAM and running Redhat Linux 6.2. OAT was implemented in C++.

6.1 Data Preparation

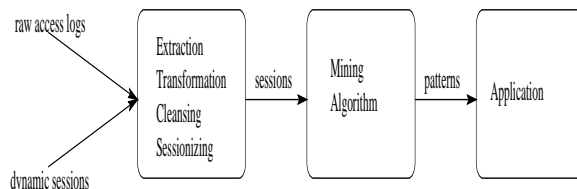


Figure 5: Traversal Pattern Mining Process

The raw Web logs collected on the Web server are not directly used for pattern mining. Instead we obtain sessions from the raw logs by extraction, cleansing, transformation and sessionization as shown in Figure 5. We process the raw logs as follows:

- extraction. We extract from the raw logs only the related fields, i.e., source IP address, timestamp and the page accessed.
- cleansing. Not all logs are used for pattern mining. For example, the graphics files(e.g., gif files) are filtered, since they are usually embedded in the HTML page and are accessed at the same time as the HTML page. In the experiments, we focus on the static HTML pages, and the dynamic pages using CGI(Common Gateway Interface), JSP(Java Server Pages), etc., are ignored. It is also interesting to identify the HTML pages accessed by agents/robots(e.g., robots for internet search engines)³. and cleanse them, since their patterns are expected to be different from these of human beings. For simplicity, in the experiments we keep all static HTML pages, regardless of whether they are accessed by agents or human beings.
- transformation. After cleansing, the remaining logs are transformed, e.g., pages(URLs) are transformed into page ids(integers), source IP addresses into source ids(integers). These transformations are not only necessary for efficiency(both in space and time), but also for privacy.
- sessionization. The transformed logs are then sessionized based on the source IP⁴ and a time window parameter. The pages accessed from the same source IP are sorted in increasing order of their timestamps. For each source IP, two pages are in the same session if their access time gap is within the time window. For example, consider the page list accessed from a source IP, $\langle (A, 0), (A, 1), (B, 2), (C, 5), (D, 10), (E, 12) \rangle$. If the time window is set to 6, all pages are within one session. For a time window of 4, there are two sessions: $\langle A, A, B, C \rangle$ and $\langle D, E \rangle$. We can see that the sessionization depends on the time window parameter. With our new definition of support(ratio of the frequency to the total number of clicks), however, the traversal patterns found are less sensitive to different time windows, since the total number of clicks keep unchanged. Note that due to caching either on the Web server side or on the browser side, some backward traversals may not be recorded in the Web logs. The inference technique introduced in[6] can be used to infer some backward traversals.

³It is itself a nontrivial problem

⁴sessionization based on the source IP may have problems when there are proxy servers and multi-user workstations, since they all have the same source IP for different users. In these cases, dynamic sessionization using cookies or URL rewriting[15] is needed

The raw Web logs were collected on our engineering school Web server from February 1 to June 30 2000. The original size is about 1.3G bytes. After extraction, cleansing and transformation, the size of the data set is about 40M bytes, and there are 1,629,492 clicks on 30,151 distinct pages from 195,924 different sources.

data set	number clicks	number sessions	max. length of sessions	avg. length of sessions
Sall.5m	1,629,492	530,122	1,061	3.07
Sall.15m	1,629,492	455,227	1,320	3.60
S100K.5m	312,782	100K	619	2.72
S100K.15m	312,782	100K	1,257	3.13

Table 6: The Session Data Sets

We sessionize the transformed logs using two different time windows, 5 minutes and 15 minutes, which results in two data sets, represented by Sall.5m and Sall.15 respectively. Two additional data sets with 100K sessions randomly selected are also used in our experiments, which are represented by S100K.5m and S100K.15m, respectively. The characteristics of these data sets are summarized in Table 6.

It can be seen that there is a big variance in the length of the sessions, i.e., a lot of sessions only 1 or 2 clicks, while some sessions are extremely long(near a thousand of clicks). One reason for such extremely long sessions is that most machines in our engineering school are multi-user workstations, so that different users on the same machine have the same source IP address. Another reason may be due to robots, since they usually search the pages thoroughly on a Web server.

6.2 Execution Time

Since all four data sets can be completely hold in main memory, we are able to run the program without any compression. The results with minimum support $s_{min} = 0.0001$ are shown in Table 7. We don't interpret the traversal patterns due to the problem of sessionization using source IP and privacy concerns. The longest frequent sequence actually consists of reloads of the same page. This can be explained by the fact that the students in a lab access the professor's homepage from the same machine during the same period. Similar results were obtained with different minimum supports.

data set	execution time(seconds)	number MFSs	max. length of MFSs	number edges in suffix tree
Sall.5m	42.32	1878	79	2,555,353
Sall.15m	50.91	1923	236	2,540,400
S100K.5m	3.58	2172	71	431,325
S100K.15m	4.27	2173	166	488,819

Table 7: The Results

We can see that the execution time of OAT is linear to the total length of data. We can also see that the execution time of OAT is not impacted by the big variance of session lengths. This is due to that the suffix tree construction only depends on the total length of data.

The execution times with different main memory constraints(shown as number of partitions) are shown in Figures 6 and 7. To compare the effect of the two pruning techniques(local pruning and cumulative pruning), the program is run with two modes: one is with local pruning only and the other with cumulative pruning and local pruning. The results for both modes are shown.

It can be seen that for the larger data sets Sall.5m and Sall.15m, the execution time of OAT decreases with more partitions. This is because the suffix tree is smaller due to more compressions. With a much smaller suffix tree, many unnecessary transitions during the construction are avoided. For the small data sets

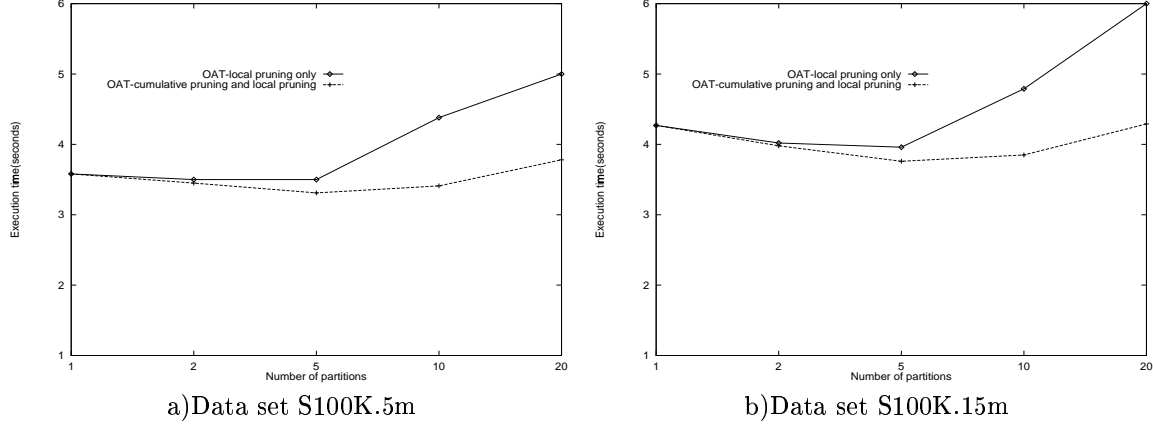


Figure 6: Execution time of OAT($s_{min}=0.0001$)

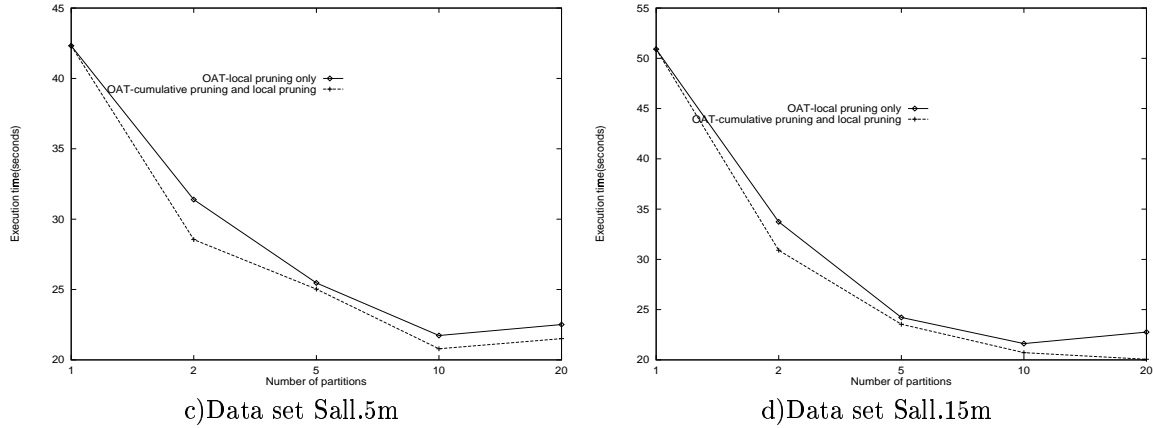


Figure 7: Execution time of OAT($s_{min}=0.0001$)

S100K.5m and S100K.15m, the execution time of OAT first increases and then decreases. This is because the overhead for the compressions increases and becomes more significant for the smaller data sets.

We can also see that with cumulative pruning and local pruning combined the algorithm is faster. This can be explained by that fact that cumulative pruning is more effective in pruning candidate patterns, which is shown in the next subsection.

6.3 Effect of Compression

Shown in Figures 8 and 9 is the number of candidate patterns (correspond to candidate itemsets for association rules) after the first scan for both modes of OAT. We can see that the number of candidate patterns for cumulative pruning increases only slightly with more partitions, while for local pruning the number increases almost linearly to the number of partitions. This demonstrates that cumulative pruning with the refined upper bound estimation is effective in pruning false candidate patterns.

The result for the data set Sall.15m with varying minimum support is shown in Figure 10. The number of partitions is set to 20. The execution time of both modes increases with a lower support, because there are more candidate patterns. The difference between the two pruning techniques also becomes more significant.

Recall that there are three kinds of compressions: suffix tree pruning (leave only the frequent sequences in the tree), suffix tree compression (leave only the MFSs in the tree) and data compression. We do not measure the suffix tree compression here, since the other two are powerful enough as shown below. The size of suffix trees is measured by the number of edges, while size of the data buffer by the number of clicks.

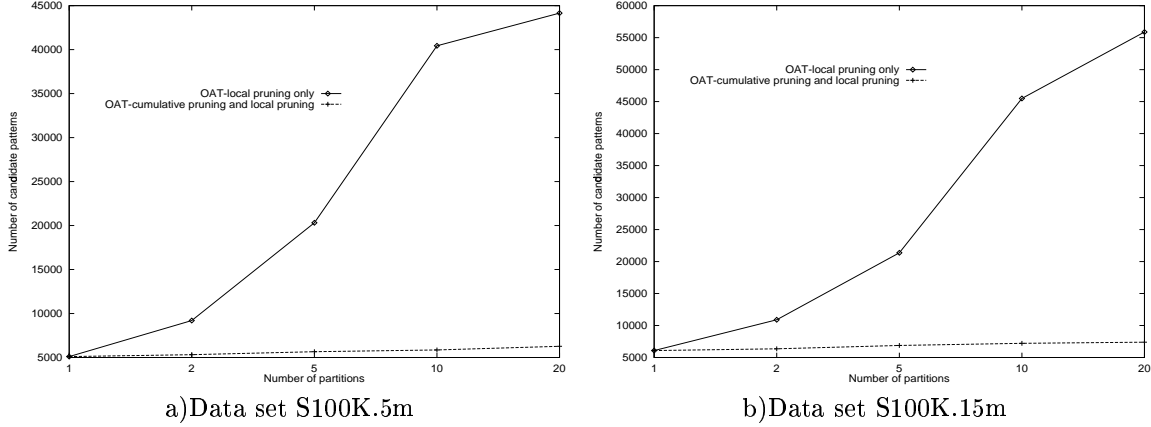


Figure 8: Number of candidate patterns($s_{min}=0.0001$)

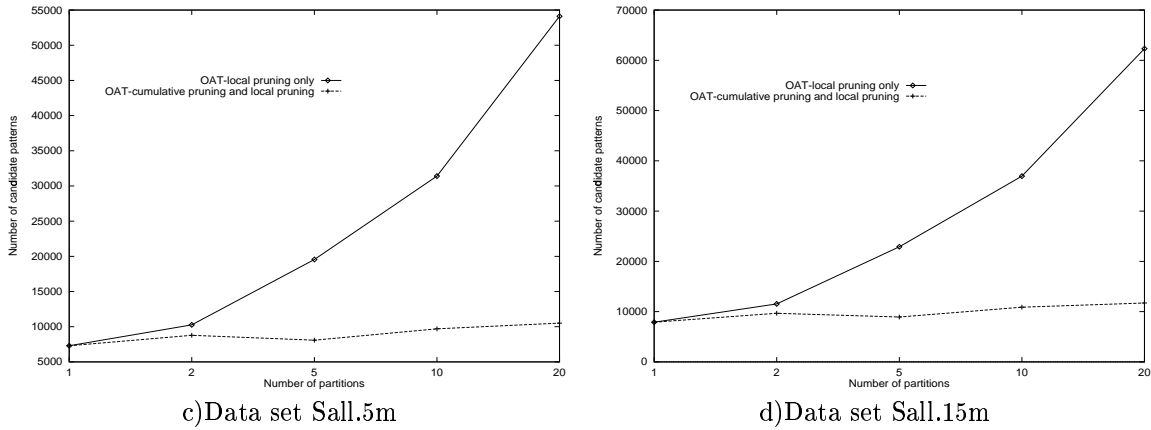


Figure 9: Number of candidate patterns($s_{min}=0.0001$)

Shown in Figure 11 are the curves for both of them (data set is Sall.15m, $s_{min} = 0.0001$ and number of partitions is 10). Note that only the sizes before the compressions and after the compressions are shown. For suffix tree pruning, the number of edges is about 3.2% of that before compressions. For data compression, the number of clicks in the data buffer is about 3.5% of that before compressions.

6.4 Scalability

Since the above four data sets are not large enough for scalability experiments, and we could not obtain any other larger data sets at this moment, we created larger data sets by duplicating the data set Sall.15m multiple times⁵. The data set Sall.15m is chosen because it contains the other three data sets. The new larger data sets are named as Sall.15m.x, where x represents the number of times Sall.15m is duplicated. All new data sets are summarized in Table 8. The largest data set Sall.15m.64 has more than 100 million clicks and about 30 million sessions, and the file size is about 0.7 giga bytes.

The execution time for two minimum supports 0.001 and 0.0001 is shown in Figure 12a) and b), respectively. The data buffer is set to be about 100,000 clicks for all cases (20 partitions for the base data set Sall.15m). Since the data buffer is fixed, the number of partitions (compressions) for each data set is different (there are up to 640 compressions for the largest data set Sall.15m.64). It can be seen that the algorithm adapts to large data sets very well, since the execution time increases linearly to the size of the data sets. With larger data sets, the difference between the two pruning techniques becomes more significant. For the

⁵Doing experiments with real data sets is still of our interest, and will be continuing

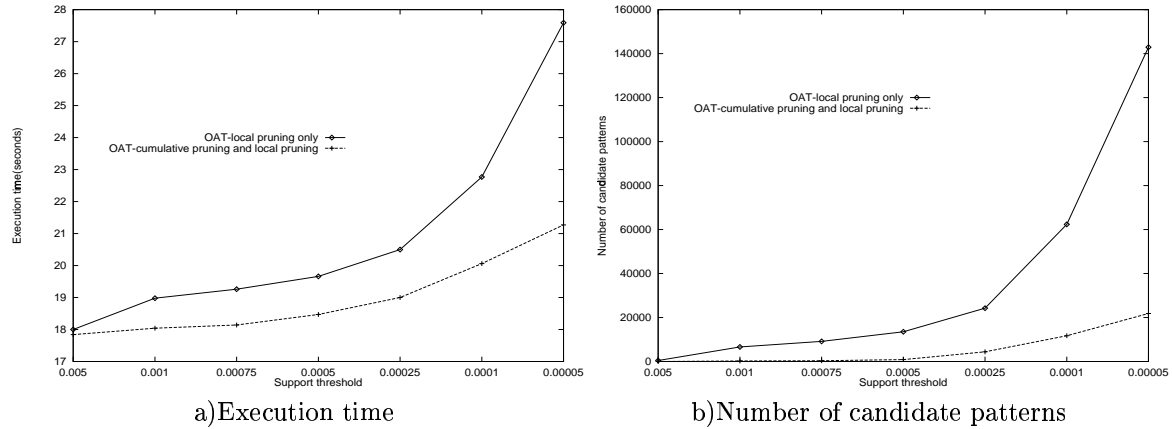


Figure 10: Result for Sall.15m with varying minimum support

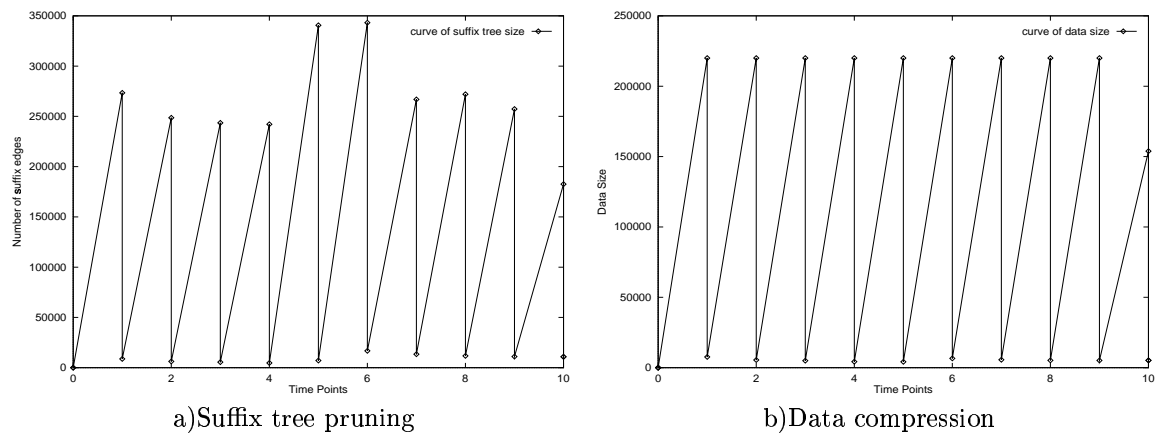


Figure 11: Effect of compression

largest data set Sall.15m.64, OAT with cumulative pruning and local pruning is about 3.2% faster than with local pruning only. Even with local pruning only, OAT can finish in less than half an hour for the large data set.

6.5 Summary of Performance

Real Web logs are used for the experiments. The two pruning techniques are shown to be effective in reducing the main memory requirement. The algorithm is also shown to adapt to large data sets very well given a limited amount of main memory.

We don't include other algorithms for comparison in the experiments because of the difference of the problem as shown in Table 1. The closest work to ours is the FS algorithm [5]. They remove all backward/duplicate traversals to get maximal forward sequences and the level-wise algorithm requires database

data set	Sall.15m.2	Sall.15m.4	Sall.15m.8	Sall.15m.16	Sall.15m.32	Sall.15m.64
number clicks	3,258,984	6,517,968	13,035,936	26,071,872	52,143,744	104,287,488
number sessions	910,454	1,820,908	3,641,816	7,283,632	14,567,264	29,134,528

Table 8: The larger data sets

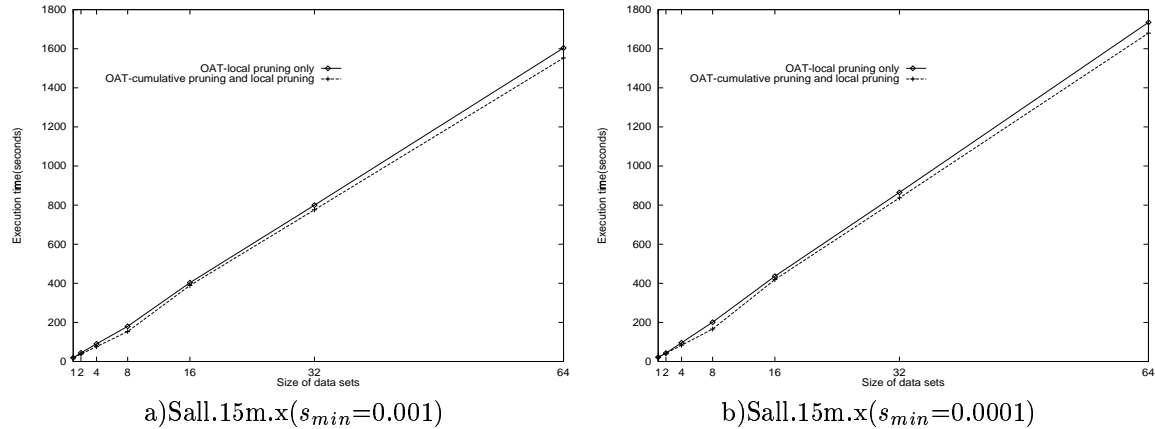


Figure 12: Scalability results

scans equal to the maximal length of frequency sequences, while we keep all traversals in sessions, and our algorithm requires at most two database scans. Our technique used in OAT can also be applied to the maximal forward sequences, and we expect that OAT will be much faster than the level-wise algorithms for these data sets with very long patterns.

7 Conclusion

We introduced a new problem of mining traversal patterns, which keep duplicates as well as consecutive ordering in the sessions. We then proposed an online and adaptive algorithm for this. Suffix trees are used for efficient counting. The adaptability to large databases is achieved through dynamic compressions and effective pruning. Experiments with real web logs demonstrated the effectiveness of our approach.

Future research will include looking at applying the OAT technique to uncover different types of patterns.

References

- [1] Rakesh Agrawal, Tomasz Imielinski, and Arun N. Swami. Mining association rules between sets of items in large databases. In Peter Buneman and Sushil Jajodia, editors, *Proceedings of the 1993 ACM SIGMOD International Conference on Management of Data*, pages 207–216, Washington, D.C., 26–28 May 1993.
- [2] Rakesh Agrawal and Ramakrishnan Srikant. Fast algorithms for mining association rules in large databases. In *Proceedings of the Twentieth International Conference on Very Large Databases*, pages 487–499, Santiago, Chile, 1994.
- [3] Paul Bieganski, John Riedl, and John Carlis. Generalized suffix trees for biological sequence data: Applications and implementation. In *available from <http://www.cbc.med.umn.edu/VirtLibrary/Bieganski/htree/htree-paper.ss.html>*, 1994.
- [4] A.G. Buchner, M. Baumgarten, S.S. Anand, M.D. Mulvenna, and J.G. Hughes. Navigation pattern discovery from internet data. In *Workshop on Web Usage Analysis and User Profiling (WEBKDD-99)*, August 1999.
- [5] Ming-Syan Chen, Jong Soo Park, and Philip S. Yu. Efficient data mining for path traversal patterns. *IEEE Transactions on Knowledge and Data Engineering*, 10(2):209–221, 1998.
- [6] Robert Cooley, Bamshad Mobasher, and Jaideep Srivastava. Data preparation for mining world wide web browsing patterns. *Knowledge and Information Systems*, 1(1), 1999.

- [7] Robert Cooley, Pang-Ning Tan, and Jaideep Srivastava. WebsIFT: The web site information filter system. In *Workshop on Web Usage Analysis and User Profiling(WEBKDD-99)*, August 1999.
- [8] Christian Hidber. Online association rule mining. In *Proceedings of the 1999 ACM SIGMOD International Conference on Management of Data*, pages 145–156, 1999.
- [9] Lucas Chi Kwong Hui. Color set size problem with application to string matching. In *Proceedings of Combinatorial Pattern Matching, Third Annual Symposium*, pages 230–243. Lecture Notes in Computer Science, Vol. 644, Springer, 1992.
- [10] Bin Lan, Stephane Bressan, and Beng Chin Ooi. Making web servers pushier. In *Workshop on Web Usage Analysis and User Profiling(WEBKDD-99)*, August 1999.
- [11] Jun-Lin Lin and Margaret H. Dunham. Mining association rules: Anti-skew algorithms. In *Proceedings of the 14th International Conference on Data Engineering*, Orlando, Florida, February 1998. IEEE Computer Society Press.
- [12] Heikki Mannila, Hannu Toivonen, and A. Inkeri Verkamo. Discovering frequent episodes in sequences. In *Proceedings of the first International Conference on Knowledge Discovery and Data Mining (KDD-95)*, pages 210–215, 1995.
- [13] Heikki Mannila, Hannu Toivonen, and A. Inkeri Verkamo. Discovering frequent episodes in event sequences. Technical Report C-1997-15, Dept. of Computer Science, Univ. of Helsinki, 1997.
- [14] E. M. McCreight. A space-economical suffix tree construction algorithm. *Journal of the ACM*, 23(1):262–272, 1976.
- [15] Karl Moss. *Java Servlets*. McGraw-Hill Book Co., 1999.
- [16] Jong Soo Park, Ming-Syan Chen, and Philip S. Yu. An effective hash based algorithm for mining association rules. In Michael J. Carey and Donovan A. Schneider, editors, *Proceedings of the 1995 ACM SIGMOD International Conference on Management of Data*, pages 175–186, San Jose, California, 22–25 May 1995.
- [17] Jian Pei, Jiawei Han, Behzad Mortazavi-Asl, and Hua Zhu. Mining access patterns efficiently from web logs. In *Proc. 2000 Pacific-Asia Conf. on Knowledge Discovery and Data Mining (PAKDD'00)*, page 592, Kyoto, Japan, April 2000.
- [18] Ashoka Savasere, Edward Omiecinski, and Shamkant B. Navathe. An efficient algorithm for mining association rules in large databases. In *Proceedings of the 21nd International Conference on Very Large Databases*, pages 432–444, Zurich, Switzerland, 1995.
- [19] Myra Spiliopoulou. Web usage mining for web site evaluation. *Communications of the ACM*, 43(8):127–134, 2000.
- [20] Ramakrishnan Srikant and Rakesh Agrawal. Mining generalized association rules. In *Proceedings of the 21nd International Conference on Very Large Databases*, pages 407–419, Zurich, Switzerland, 1995.
- [21] Hannu Toivonen. Sampling large databases for association rules. In *Proceedings of the 22nd International Conference on Very Large Databases*, pages 134–145, Mumbai, India, 1996.
- [22] Esko Ukkonen. On-line construction of suffix trees. *Algorithmica*, 14(3):249–260, September 1995.
- [23] Jason Tsong-Li Wang, Gung-Wei Chirn, Thomas G. Marr, Bruce A. Shapiro, Dennis Shasha, and Kaizhong Zhang. Combinatorial pattern discovery for scientific data: Some preliminary results. In Richard T. Snodgrass and Marianne Winslett, editors, *Proceedings of the 1994 ACM SIGMOD International Conference on Management of Data, Minneapolis, Minnesota, May 24-27, 1994*, pages 115–125. ACM Press, 1994.
- [24] Peter Weiner. Linear pattern matching algorithms. In *14th Annual Symposium on Switching and Automata Theory*, pages 1–11, 1973.