EFFICIENT WEB MINING FOR TRAVERSAL PATH PATTERNS

Zhixiang Chen [1], Richard H. Fowler [1], Ada Wai-Chee Fu [2], Chunyue Wang [1]

[1] *Department of Computer Science*
The University of Texas - Pan American
Edinburg, Texas 78539, USA
Email: chen@cs.panam.edu, fowler@panam.edu, cwang@panam.edu

[2] *Department of Computer Science*
The Chinese University of Hong Kong, Hong Kong
Email: adafu@cse.cuhk.edu.hk

Abstract

A maximal forward reference of a Web user is a longest consecutive sequence of Web pages visited by the user in a session without revisiting some previously visited page in the sequence. Efficient Mining of frequent traversal path patterns, i.e., large reference sequences of maximal forward references, from very large Web logs is a fundamental problem in Web mining. This chapter aims at designing algorithms for this problem with the best possible efficiency. First, two optimal linear time algorithms are designed for finding maximal forward references from Web logs. Second, two algorithms for mining frequent traversal path patterns are devised with the help of a fast construction of *"shallow*" generalized suffix trees over a very large alphabet. These two algorithms have respectively provable linear and sublinear time complexity, and their performances are analyzed in comparison with the apriori-like algorithms and the Ukkonen algorithm. It is shown that these two new algorithms are substantially more efficient than the apriori-like algorithms and the Ukkonen algorithm.

Keywords: Web mining, access sessions, maximal forward references, traversal path patterns, suffix trees

INTRODUCTION

Because of its significant theoretical challenges and great application and commercial potentials, Web mining has recently attracted extensive attention (e.g., Buchner, et al., 1998, 1999; Catledge & Pitkow, 1995; Chen, et al., 1998; Cooley, et al., 1997). Surveys of recent research in Web mining can be found in (Kosala & Blockeel, 2000; Srivastava, et al., 2000). One of major concerns in Web mining is to discover user traversal (or navigation) path patterns that are hidden in vast Web logs. Such discovered knowledge can be used to predict where the Web users are going, i.e., what they are seeking for, so that it helps the construction and maintenance of real-time intelligent Web servers that are able to dynamically tailor their designs to satisfy users' needs (Perkowitz & Etzioni, 1998). It has significant potential to reduce, through prefetching and cashing, Web latencies that have been perceived by users year after year (Pitkow & Pirolli, 1999; Padmanabhan & Mogul, 1996). It can also help the administrative personnel to predict the trends of users' needs so that they can adjust their products to attract more users (and customers) now and in the future (Buchner & Mulvenna, 1998).

Traversal path pattern mining is based upon the availability of traversal paths that must be obtained from raw Web logs. A maximal forward reference of a Web user, a longest consecutive sequence of Web pages visited by the user without revisiting some previously visited page in the sequence, is a typical traversal path pattern (Chen, et al., 1998; Cooley, et al., 1999). This chapter studies the problem of efficient mining of frequent traversal path patterns

that are large sequences of maximal forward references. The previously known algorithms for the problem are the two apriori-like algorithms *FullScan* and *SelectiveScan* designed in (Chen, Park & Yu, 1998).

This chapter aims at designing algorithms for mining traversal path patterns with the best possible efficiency. The main contributions are summarized as follows. First, two algorithms ISMFR (Interval Session Maximal Forward References) and GSMFR (Gap Session Maximal Forward References) are given for finding maximal forward references from raw Web logs. The first algorithm is designed for *interval sessions* of user accesses and the second for *gap sessions*. The two algorithms have linear, hence optimal, time complexity, and are substantially more efficient than the sorting based method devised in (Chen, Park & Yu, 1998) for finding maximal forward references. Second, the problem of mining frequent traversal path patterns from maximal forward references is investigated with the help of a fast construction of *"shallow"* generalized suffix trees over a very large alphabet (Chen, at el., 2003a, 2003b). Precisely, a *"shallow"* generalized suffix tree is built for a set of maximal forward references obtained by the algorithm ISMFR (or GSMFR) such that each tree node contains the frequency of the substring represented by that node. Once such a tree is built, a simply traversal of the tree outputs frequent traversal path patterns, i.e., frequent substrings, with respect to some given frequency threshold parameter. Two algorithms SbSfxMiner (Sorting Based Suffix Tree Miner) and HbSfxMiner (Hashing Based Suffix Tree Miner) are designed to overcome some well-understood obstacles of suffix tree construction (Gusfield, 1997, pp. 116-119): the complexity of suffix tree construction is dependent on the size of the underlying alphabet and a suffix tree does not have nice locality properties to support memory paging. These two algorithms SbSfxMiner and HbSfxMiner have

respectively provable sublinear and linear time complexity. Performances of these two algorithms are analyzed in comparison with the two apriori-like algorithms FullScan and SelectiveScan in (Chen, Park & Yu, 1998) and the Ukkonen algorithm for linear time suffix construction (Ukkonen, 1995). It is shown that these two new algorithms are substantially more efficient than FullScan, SelectiveScan and the Ukkonen algorithm as well. In short, algorithm HbSfxMiner has optimal complexity in theory, while algorithm SbSfxMiner has the best empirical performance.

The rest of the chapter is organized as follows. The problem formulation is given in the second section. The related work is reviewed in the third section. Algorithms ISMFR and GSMFR are designed in the fourth section for finding maximal forward references from raw Web logs. Characteristics of maximal forward references are given in the fifth section. In the sixth section, "*shallow*" generalized suffix trees are introduced. In the seventh section, algorithms SbSfxMiner and HbSfxMiner are designed for mining frequent traversal path patterns, and their performances are comparatively analyzed. Finally, conclusions are given in the last section 8.

PROBLEM FORMULATION

The Web has a natural graph structure: pages in general are linked via hyperlinks. When a user surfs the Web, she may move forward along the graph via selecting a hyperlink in the current page. She may also move backward to any page visited earlier in the same session via selecting a "*backward*" icon. A forward reference may be understood as the user looking for her desired information. A backward reference may mean that the user has found her desired

information and is going to looking for something else. A sequence of consecutive forward references may indicate the information for which the user is looking. A maximal forward reference is defined as the longest consecutive sequence of forward references before the first backward reference is made to visit some previously visited page in the same session. Thus, the last reference in a maximal forward sequence indicates a content page (Chen, et al., 1998; Cooley, et al., 1999) that is desired by the user. Under such understanding, when a user searches for desired information, her information needs can be modeled by the set of maximal forward references that occurred during her search process.

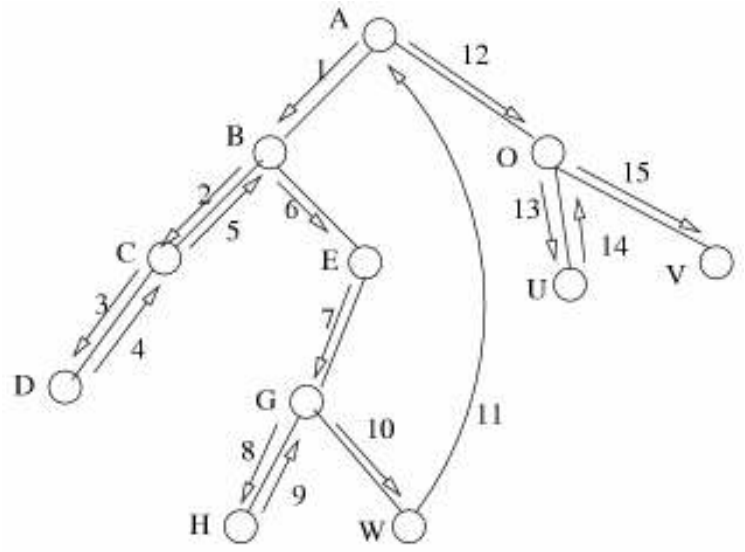


Figure 1. An illustration of traversal path patterns

Suppose that, the Web log contains the traversal path *{A, B, C, D, C, B, E, G, H, G, W, A, O, U, O, V}* for a user as shown in Figure 1. Starting at *A*, one has forward reference sequences *AB, ABC* and *ABCD*. Then, at *D* a backward reference to *C* is made, meaning that *ABCD* is a maximal forward reference. Now, some caution shall be paid to *C*. Here, even though a backward reference is made to *B*, one shall not consider *ABC* as a maximal forward reference, because the current backward reference is the second backward reference (the first is $D \rightarrow C$). At

B, a forward reference is made to *E*, thus begins new forward reference sequences *ABE, ABEG* and *ABEGH*. *ABEGH* is a maximal forward reference, because the first backward reference occurs at *H* after the forward reference *E*. Similarly, one can find the other maximal forward references *ABEGW, AOU* and *AOV*. When *D, H, W, U* and *V* are the pages desired by the user, the set of the maximal forward references *ABCD, ABEGH, ABEGW, AOU, AOV* precisely describes the user's needs and her actual search behaviors as well.

A "*large reference sequence*" with respect to some given frequency threshold parameter α is defined as a consecutive subsequence that occurs at least α many times in a set of maximal forward references. A *"maximal reference sequence"* with respect to α is a large reference that is not contained in any other large reference sequence. In the above example, when the frequency threshold parameter α is 2, one has large reference sequences *A, B, E, G, O, AB, BE, EG, AO, ABE, BEG, ABEG,* and the maximal reference sequences are *ABEG* and *AO*. A maximal reference sequence corresponds to a frequent traversal path pattern, i.e., the *"hot"* access pattern of users. The goal in this chapter is to design algorithms for mining frequent traversal path patterns, i.e., frequent or maximal reference sequences, with the best possible efficiency. Once large reference sequences are determined, maximal references sequences can be obtained in a straightforward manner (Chen, Park & Yu, 1998).

The traversal path *A, B, C, D, C, B, E, G, H, G, W, A, O, U, O, V* of the user in the above example is recorded in the *referrer log*, and so are traversal paths of all the other users. A *referrer log* is a typical configuration of Web logs, where each traversed link is represented as a pair *(source, destination)*. In this chapter, one assumes that Web logs are referrer logs. The high-level design of the approaches to mining frequent traversal path patterns is as follows: first,

one finds maximal forward references from Web logs. Second, one builds a generalized suffix tree for the set of maximal forward references. Last, one traverses the tree to generate large or maximal reference sequences.

RELATED WORK

The problem of mining frequent traversal path patterns resulted in an algorithm for finding frequent traversal path patterns from a Web log (Chen, Park, & Yu, 1998). That algorithm works in two phases. First, the log is sorted according to user IDs to group every user's references as a traversal path. Next, each traversal path is examined to find maximal forward references. The time complexity of the algorithm is $O(N \log N)$, where N denotes the number of records in the log. At the next step, two apriori-like algorithms *FullScan* and *SelectiveScan* were designed for mining large reference sequences from maximal forward references and the performances of these two algorithms were analyzed.

Cooley, et al. (1999) has detailed discussions of various data preparation tasks. The maximal forward references, called *transactions*, were also examined as a finer level characterization of user access sessions. But no explicit algorithms were given to find maximal forward references, and mining frequent traversal path patterns was not studied there. The work in (Xiao & Dunham, 2001) studied the application of generalized suffix trees to mining frequent user access patterns from Web logs. However, the patterns in (Xiao & Dunham, 2001) are not traversal path patterns as considered in the usual settings in much of the literature such as (Chen, et al., 1998; Pitkow & Pirolli, 1999; Su, et al., 2000; Padmanabhan & Mogul, 1996) and in the context of this chapter. The patterns in that paper is in essence frequent consecutive

subsequences of user access sessions, while traversal paths corresponding to the underlying hyperlink structure were not of concerned. Furthermore, that paper relies on the Ukkonen's "*suffix link"* algorithm to construct a suffix tree and hence cannot overcome the well-understood obstacles of suffix tree construction (Gusfield, 1997, pp.116-119), i.e., the complexity dependence on the size of the underlying alphabet and the lack of good locality properties to support memory paging.

Many researchers have been investigating traversal or surfing path patterns in order to model users' behaviors and interests. The uncovered model can be used in many applications such as prefetching and caching to reduce Web latencies (Pitkow & Priolli, 1999; Padmanabhan & Mogul, 1996; Su, et al., 2000). The *N*-gram approaches are used to find most frequent subsequences of the paths, and sometimes the most frequent longest subsequences are the most interested. Once those subsequences are obtained, methods such as first order or higher order Markov-chains can be applied to predict users' behaviors or interests. Other researchers have also considered user access sessions as ordered sequences (Masseglia, et al., 1999). They have also studied how to identify the longest repeating subsequences of the ordered sessions to find associations of those sequences. In all those papers, a user's surfing path is considered as an ordered sequence of consecutive references. In other words, a user's surfing path is the user's access session in order. This is different from the maximal forward references studied in (Chen, et al., 1998; Cooley, et al., 1999) and in the present chapter.

In (Chen, Fu & Tung, 2003), optimal algorithms are designed for finding user access sessions from very large Web logs. Since maximal forward references are finer level characterizations of sessions as carefully examined in (Cooley, et al., 1999), the algorithms in

(Chen, Fu & Tung, 2003) cannot be applied directly to the problem of finding maximal forward references. However, these algorithms were improved in (Chen, Fowler & Fu, 2003) to finding maximal forward references with optimal time complexity.

FINDING MAXIMAL FORWARD REFERENCES

Sessions

The basic idea of finding maximal forward references is as follows. The algorithm reads the Web log once sequentially, and generates sessions on the fly. While a session is generating, maximal forward references within that session are also generated. Although it is easy to understand that a "*session"* or "*visit*" is the group of activities performed by a user from the moment she enters a server site to the moment she leaves the site, yet it is not an easy task to find user sessions from Web logs. Due to security or privacy, a user's identity is often not known except the client machine's IP address or host name. When the log is examined, it is not clear when a session starts or when it ends. Some users access the Web, page after page, at a very fast pace. Others access the Web at a very slow pace, visiting one for a while, doing something else, and then visiting another. Local caching and proxy servers cause other difficulties.

Researchers have proposed cut-off thresholds to approximate the start and the end of a session (Cooley, et al., 1999; Berendt, et al., 2001; Chen, et al., 2003). One may also propose a maximum *30*-minutes gap between any two pages accessed by the user in a session. If the gap limit is exceeded, then a session boundary should be inserted between the two pages. As in (Cooley, et al., 1999; Chen, et al., 2003; Berendt, et al., 2001), the following two types of sessions are considered in this chapter:

α-interval sessions: the duration of a session may not exceed a threshold of α. That is, one assumes that a user should not spend too much time for each session. So a value α is given to limit the time a user can have for a session. Usually, α may be set to 30 minutes.

β-gap sessions: the time between any two consecutively accessed pages may not exceed a threshold of β. That is, it is assumed that a user should not be *"idle"* too much time between any two consecutive accesses in any session. So a value β is given to limit the time a user can be *"idle"* between two consecutive accesses in a session. Usually, β may also be set to 30 minutes.

Certainly, there are exceptions to α and β limits. For example, a person may need to spend a few hours to find a right tour map of Hong Kong over the Web, so the session exceeds the α limit. However, these two kinds of limits are quite accurate in Web usage analysis (Berendt, et al., 2001). A more sophisticated consideration would allow adaptive (or variable) limits for different users in particular applications. This is not trivial due to precise modeling of user behaviors, and is beyond the scope of this chapter.

Data Structures

The proposed algorithms maintain the following data structures. A URL node structure is defined to store the URL and the access time of a user access record and a pointer to point to the next URL node. A user node is also defined to store the following information for the user: the user ID, usually the hostname or IP address in the access record; the start time, i.e., the time of the first access record entered into the user node; the current time, i.e., the time of last access

record entered into the user node; the URL node pointer pointing to the linked list of URL nodes; a forward reference tag indicating whether the last reference of the user is forward reference or not; the counter recording the total number of URL nodes added; and two user node pointers to respectively point to the previous user node and the next. Finally, a head node is defined with two members, one pointing to the beginning of the two-way linked list of user nodes and the other storing the total number of valid records that have been processed so far.

The user nodes and URL linked lists cannot be allowed to exceed the limit of RAM. This difficulty is overcome with the sorted property of the web log as follows: it is not a concern how many user nodes there may be. When an access record of a new user arrives, a user node is created for her. When a new access record of some existing user arrives, the time of this new record and start time (or the current time) of the user node is checked to determine whether the α threshold is exceeded for interval sessions (or the β threshold is exceeded for gap sessions). To do so requires knowing how to control the size of the URL linked list for the user. After a certain number of valid user access records have been processed, all those user nodes with start (or current) times beyond the α (or β) threshold in comparison with the time of the most recent record processed are purged, thus the number of user nodes is under control.

Finding Maximal Forward References from Interval Sessions

At the high level design, the algorithm repeats the following tasks until the end of the Web log: read a user access record *r* from disk to RAM. Test whether *r* is valid or not; if *r* is valid then add *r* to its corresponding user node *D* in the user node linked list. Compare the start time of *D* with the time $t(r)$ to see whether a new session of the user begins or not. If yes then

output the URLs in *D*'s URL linked list in order as a maximal forward reference sequence, and reset *D* with information in *r,* otherwise use *r* to search *D*'s URL linked list to find maximal forward reference sequences. Finally, check if a γ number of valid records have been processed so far, where γ is tunable parameter that is dependent on the size of the underlying RAM. If yes, then purge all the old user nodes and output their sessions. The formal description of the algorithm, called ISMFR, is given as follows.

Algorithm ISMFR (Interval Session Maximal Forward References):

```
   input:
      infile: input web log file
      outfile: output user access session file
      α: threshold for defining interval sessions
      γ: threshold for removing old user nodes
   begin
1.    open infile and outfile
2.    createHeadNode(S); S.head = null; S.counter=0
3.    while (infile is not empty)
4.          readRecord(infile,r)
5.          if (isRecordValid(r))
6.                n = findRecord(S,r)
7.           if (n is null)
8.                    addRecord(S, r)
9.                     else if  (t(r) − n.startTime ≤ α)
10.                    n.currentTime = t(r)
11.                    findMaxFRS(n.urlListPtr, n.forwardReferenceTag, r)
12.          else
13.               writeMaxFRSAndReset(outfile, n, r)
14.           S.counter=S.counter+1
15.           if (S.counter > γ )
16.                purgeNodes(outfile, S, t(r)); S.counter=1
17.           cleanList(outfile,S);
18.           close infile and outfile
    end
```

Finding Maximal Forward Reference Sequences from Gap Sessions

The algorithm, called GSMFR, is similar to algorithm ISMFR. They differ at steps 9 and 16. At step 9 GSMFR checks, to decide whether a new session begins or not, and whether the

"time gap" between the current record and the last record of the same user is beyond the threshold β or not. At step 16, algorithm GSMFR will call function *purge2Nodes(outfile, S, t(r))* to check, for every user node in the user linked list *S.head*, whether the *"time gap"* between the current record and the last record of the user node is beyond the threshold β or not. If so, it purges the user node in a similar manner as algorithm ISMFR. The description of GSMFR is given in the following.

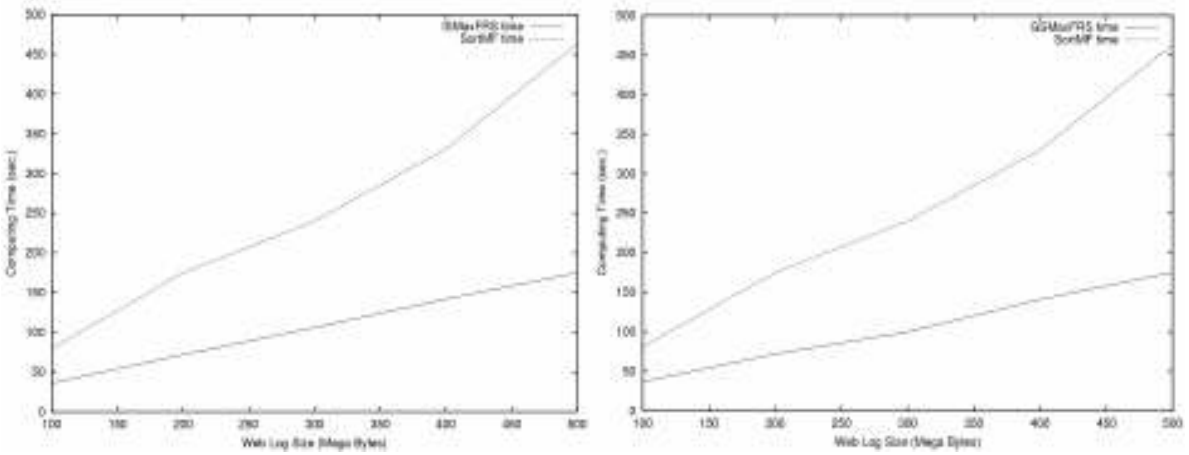
Algorithm GSMFR (Gap Session Maximal Forward References):
input: the same as ISMFR except
$\beta > 0$ is used to replace α
The body is also the same as ISMFR except that
Line 9 is replaced by
9. else if $(t(r) - n.cuurentTime \le \beta)$
and Line 16 is replaced by
16. purge2Nodes(outfile, S, t(r)); S.counter=1

Performance Analysis

When the parameters α, β and γ are given, it is quite easy to see that both algorithms ISMFR and GSMFR have $\theta(N)$ optimal time complexity, where N denoted the number of user access records in the Web log. It should be pointed out that different thresholds α, β and γ may be needed for different applications. The only explicitly reported algorithm for finding maximal forward references from Web logs is the sorting-based algorithm SMF (Chen, Park & Yu, 1998).

To compare the two algorithms, ISMFR and GSMFR, with the SMF algorithm uses five web logs *L100MB, L200MB, L300MB, L400MB* and *L500MB* with 100 to 500 mega bytes of user access records collected from the Web server of the Department of Computer Science, the University of Texas - Pan American. The computing environment is a Gateway Dell E-5400 PC with 512 mega bytes of RAM and 20 GB bytes of hard disk. The *"sort"* command in the DOS

environment, which supports standard external file sorting, is utilized. All programs were implemented in Microsoft Visual C++ 6.0. Setting $\alpha = \beta = 30$ minutes and $\gamma = 500$, and using the same *isRecordValid( )* function for the three algorithms. Performance comparisons are illustrated in Figure 2. The empirical analysis shows that both algorithms ISMFR and GSMFR have almost the same performance, and are substantially much faster than the sorting based algorithm SMF.



(a) ISMFR vs. SMF (b) GSMFR vs. SMF

Figure 2. Performances of Algorithms ISMFR and GSMFR

PROPERTIES OF MAXIMAL FORWARD REFERENCES

The properties of maximal forward references of the five logs *L100MB, L200MB, L300MB, L400MB* and *L500MB* have been examined. When the parameters α and β are set to 30 minutes to define interval sessions and gap sessions, it is obvious that the length of a maximal forward reference, or the number of URLs in it, is small. (Users on average won't make too many clicks along a path, though Web indexers may.) Distributions, accumulative distributions and average lengths of maximal forward references are shown in Figure 3. Part (a) of Figure 3 shows the number of maximal forward references for each given length on X-axis. Part (b) shows the accumulative number of maximal forward references with length less than or equal to a given

value on X-axis. Part (c) shows average lengths of maximal forward references in Web logs of sizes on X-axis. Part (d) shows sizes of sets of maximal forward references from Web logs of sizes on X-axis.

In summary, when the parameters α and β are set to 30 minutes to define interval sessions and gap sessions, maximal forward references in the five Web logs have the following three properties:

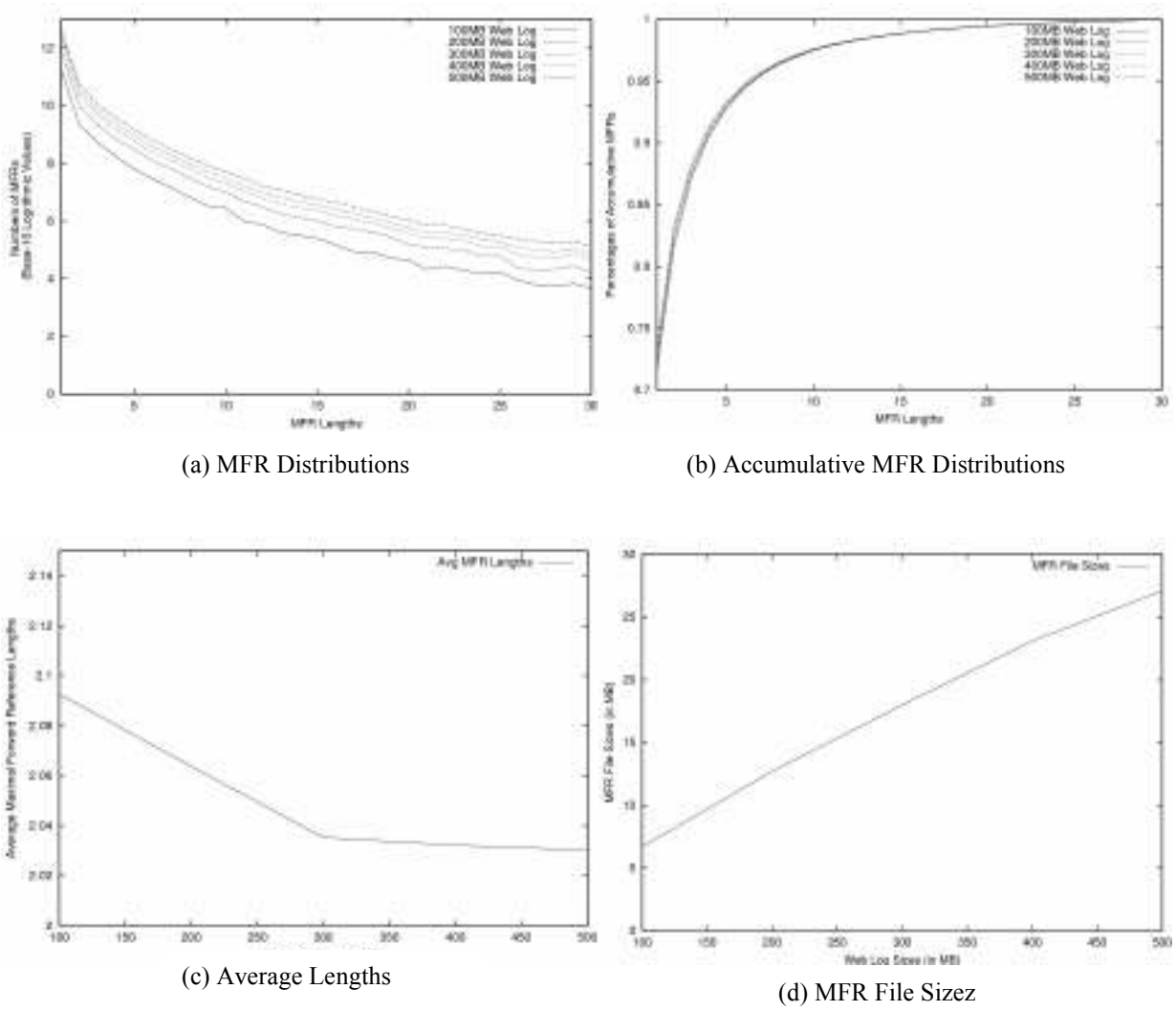
1. Almost all maximal forward references have a length ≤ 30.
2. More than 90% maximal forward references have lengths less than or equal to 4, and the average length is about 2.
3. The number of unique URLs is 11,926.

In addition, the sizes of five maximal forward reference files corresponding to the five Web logs range from 6.7 mega bytes to 27.1 mega bytes. One must caution readers that the above three properties are not typical to the Web logs considered here.

"Shallow" Generalized Suffix Trees Over a Very Large Alphabet

Suffix trees are old data structures that become new again and have found many applications in data mining and knowledge discovery as well as bio-informatics. The first linear-time algorithm for constructing suffix trees was given by Weiner (1973). A different but more space efficient algorithm was given by McCreight (1976). Almost twenty years later Ukkonen (1995) gave a conceptually different linear time algorithm that allows on-line construction of a suffix tree and is much easier to understand. These algorithms build, in their original design, a suffix tree for a single string S over a given alphabet Σ. However, for any set of strings $\{S_1, S_2, ..., S_n\}$ over Σ, those algorithms can be easily extended to build a tree to represent all

suffixes in the set of strings in linear time. Such a tree that represents all suffixes in strings $S_1, S_2, ..., S_n$ is called a "*generalized*" suffix tree. Usually, a set of strings $S_1, S_2, ..., S_n$ is represented as $S_1\$S_2\$\cdots\$S_n\$$.



(a) MFR Distributions

(b) Accumulative MFR Distributions

(c) Average Lengths

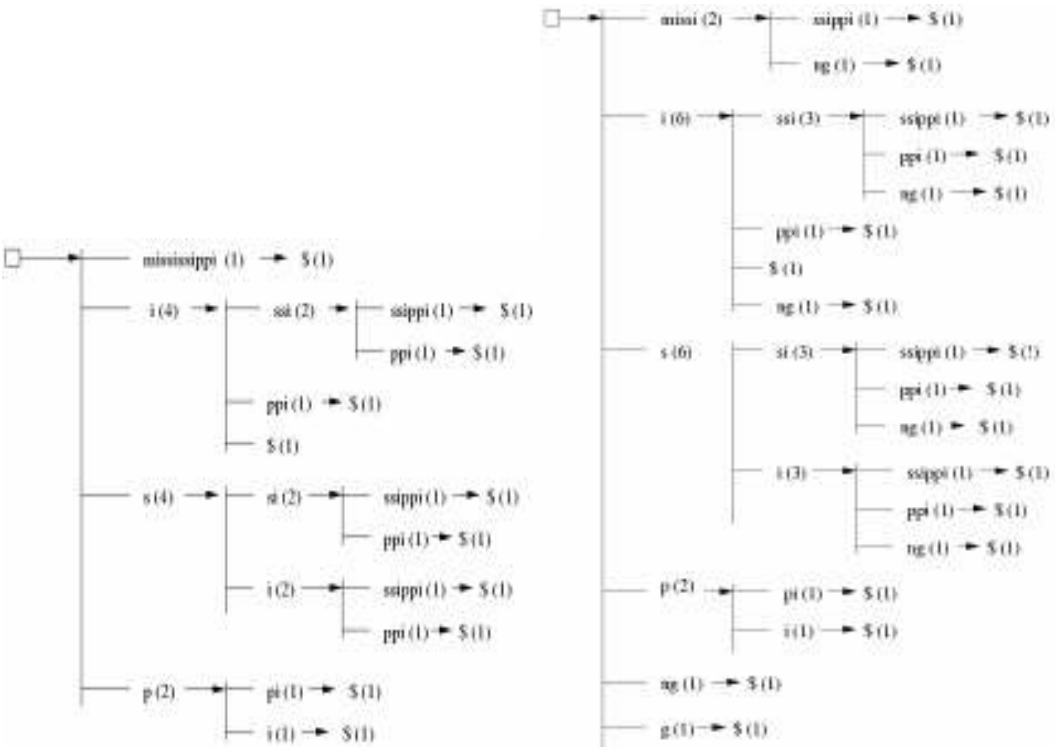
(d) MFR File Sizez

Figure 3. Properties of Maximal Forward References (MFRs)

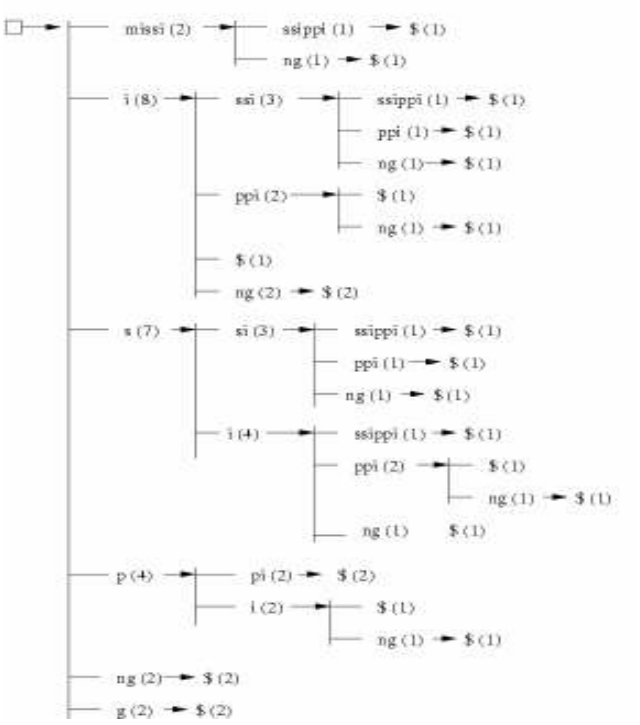
One typical application of generalized suffix trees is the identification of frequent (or longest frequent) substrings in a set of strings. This means that generalized suffix trees can be used to find frequent traversal path patterns of maximal forward references, simply because such patterns are frequent (or longest frequent) substrings in the set of maximal forward references,

when maximal forward references are understood as strings of URLs. Figure 4 illustrates a suffix tree for the string "*mississippi"*, and generalized suffixed trees for "*mississippi$missing$"* and *"mississippi$missing$sipping$"*. Notice that in the generalized suffix trees, a counter is used at each internal node to indicate the frequency of the substring labeling the edge pointing to the node.

For any given set of strings $S_1, S_2, ..., S_n$ over an alphabet Σ, let $m = \sum_{i=1}^{n} | S_i |$. It is well understood (Gusfield, 1997, pp. 116-119) that the linear time (or space) complexity of Weiner, Ukkonen, and McCreight algorithms have all ignored the size of the alphabet Σ, and that memory paging was not considered for large trees and hence cannot be stored in main memory. Notice that suffix trees or generalized suffix trees do not have nice locality properties to support memory paging. When both m and Σ are very large, the time complexity of those classical algorithms is $O(m | \Sigma |)$, and the space complexity is $\theta(m | \Sigma |)$. In particular, as pointed out in the fifth section, the alphabet Σ in the case of mining frequent traversal path patterns is 11,926 (the number of unique URLs) which is too big to be ignored, and m ranges from 6.7 mega bytes to 27.1 mega bytes and hence the $\theta(m | \Sigma |)$ space requirement is far beyond the main memory limit of a reasonable computer. Therefore, the challenges of applying generalized suffix trees to efficient mining of frequent traversal path patterns are how to overcome the efficiency dependence on the size of an alphabet in building a generalized suffix tree for a set of maximal forward references, and how to find innovative ways to scale down the space requirement. As one has learned from the fifth section, in the case of mining frequent traversal path patterns, on the one hand the size of the alphabet Σ (the set of all URLs) is very large, and files of maximal forward



(a) "Mississippi$ (b) "Mississippi$missing$



(c) "Mississippi$missing$sipping$

Figure 4. A Suffix Tree and Two Generalized Suffix Trees

references are also very large. On the other hand, each maximal forward reference has a short length, less than or equal to some small constant (when a threshold is used to delimit sessions); the vast majority have just several URLs; and the average length is even smaller. Thus, the generalized suffix tree for the sequence of maximal forward references in a Web log is "*shallow*" and very "*flat".* In the case of the logs in the fifth section, the width of the first level of the tree is $s=|\Sigma|=$ 11,926; the height of each of the s subtrees of the root is no more than 30; more than 90% of those subtrees have a height no greater than 4; and the average height is just slightly more than 2. In general, if a generalized suffix tree has a depth bounded by some small constant, then we call it a "*shallow*" generalized suffix tree.

MINING FREQUENT TRAVERSAL PATH PATTERNS

It is assumed in this section that either algorithm ISMFR or GSMFR has obtained maximal forward references. Fast construction of shallow generalized suffix trees over a very large alphabet is studied (Chen, et al., 2003a). The techniques developed there will be used here to design algorithms with the best possible efficiency for mining frequent traversal path patterns. Let Σ be the alphabet of all URLs in maximal forward references. For any maximal forward reference S over Σ with a length of n (i.e., S has n URLs in it), $S[1:n]$ denotes S, where $S[i]$ is the i-th URL in S. One uses RstUkkonen(SuffixTree sft, NewString str) to denote the restricted version of Ukkonen algorithm for building a generalized suffix as follows: it takes an existing generalized suffix tree *sft* and a new string $str[1:n]$ as input and adds the suffix $str[1:n]$ to *sft* but ignores all the suffixes $str[2:n], str[3:n], \ldots, str[n:n]$. When no confusion arises, letters means URLs, and strings means maximal forward references.

Consider a generalized suffix tree *sft* of a set of strings. For each subtree *t* of *sft* such that the root of *t* is a child of the root of *sft,* let *t.first()* denote the first letter occurred at *t*. Here the subtree *t* contains exactly all suffixes starting with the letter *t.first(),* and any two distinct subtrees must have distinct first letters (Chen, et al., 2003a). Furthermore, a simple traversal of the subtree *t* can generate all the frequent (or longest frequent) substrings starting with the letter *t.first().* E.g., the first letter occurred at the top subtree in Figure 4(c) is *m*, and all suffixes starting with *m* in the strings *mississippi$missing$sipping$* are contained in that subtree. This critical observation leads to the designs of the new algorithms: the strategy is to organize all suffixes starting with the same letter in to a group and build a subtree for each of such group. A more or less related strategy has been devised in (Hunt, et al., 2001), but the strings considered there are over a small alphabet and the method used to build subtrees is of quadratic time complexity.

In the following two subsections, N is used to denote the number of strings in a given set of strings, and a percentage parameter *pct* is used to determine the frequency threshold $pct * N$.

Algorithm SbSfxMiner

The key idea for designing algorithm SbSfxMiner is as follows. Read strings sequentially from an input file, and for every string *s[1:n]* output its *n* suffixes to a temporary file. Then sort the temporary file to group all the suffixes starting with the same letter together. Next, build a generalized suffix tree for each group of such suffixes. (Precisely, the tree here is a subtree of the conventional generalized suffix tree.) Finally, traverse the tree to output all frequent (or longest frequent) substrings. It follows from the properties of maximal forward references in the fifth section that the size of the temporary file is about $O(N)$. Sorting this file takes $O(N \log N)$ time.

By the property of the Ukkonen algorithm, building a generalized suffix tree for a group of strings takes *O(N)* time. Hence, the whole process is of sublinear time. The description of SbSfxMiner is given in the following.

```
Algorithm SbSfxMiner:
    input:
        infile: maximal forward reference file
        tmpfile: temperary file
        pct: frequencey threshold = pct * N
        outfile: frequenct maximal forward reference pattern file
    Begin
1.      infile.open(); tmpfile.open()
2.      while (infile  is not empty)
3.          readMFR(infile, s)
4.          for (i = 1, i ≤ n; i++)
5.              tmpfile.append(s[i:n])
6.      infile.close();outfile.open()
7.      sort(tmpfile);  createSuffixTree(sft)
8.      while (tmpfile is not empty)
9.          readMFR(tmpfile, s)
10.         if (sft.empty() or sft.first() == s[1])
11.             RstUkkonen(sft, s)
12.         else if (sft.first()  ≠ s[1])
13.             traverseOutput(sft, outfile, pct)
14.     createSuffixTree(sft)
15.     traverseOutput(sft,outfile, pct)
16.     tmpfile.close(); outfile.close()
    end
```

Algorithm HbSfxMiner

The key idea of designing algorithm HbSfxMiner is to eliminate the sorting process to group all suffixes with the same starting letter together. The new approach is to design a fast function to map each letter to a unique integer, and to use this function to map suffixes with the same starting letter into a group. Because URLs have a nice directory path structure, such a function can be designed with constant time complexity. In the general case, one may use a hashing function to replace the function so that it will be more efficient in computing hashing

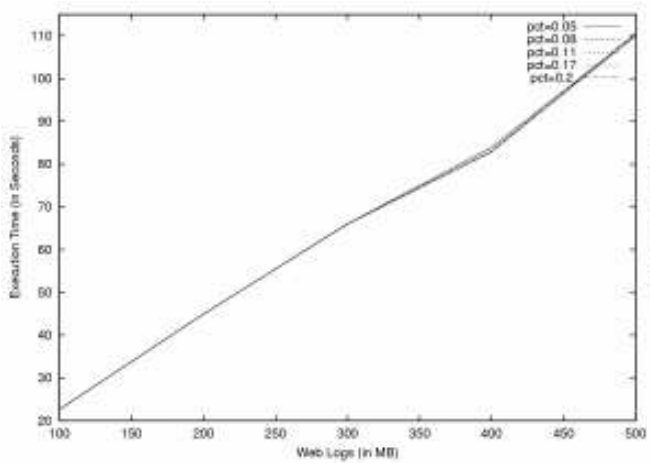
values of letters. Because sorting is not required, the time complexity of the new algorithm is linear. The following is the detailed description of the algorithm.

```
Algorithm HbSfxMiner:
    input:
        infile: maximal forward reference file
        f: a function from letter to integer
        pct: frequencey threshold =pct*N
        outfile: frequenct maximal forward reference pattern file
    Begin
1.      infile.open(); outfile.open()
2.      createSuffixTrees(sft, size)
3.      while (infile is not empty)
4.      readMFR(infile, s)
5.      for (i = 1; i ≤ n; i++)
6.          RstUkkonen(sft[f(s[i])], s[i:n])
7.      for (i = 1; i < size; i++)
8.          traverseOutput(sft[i], outfile, pct)
9.      tmpfile.close(); outfile.close()
    end
```

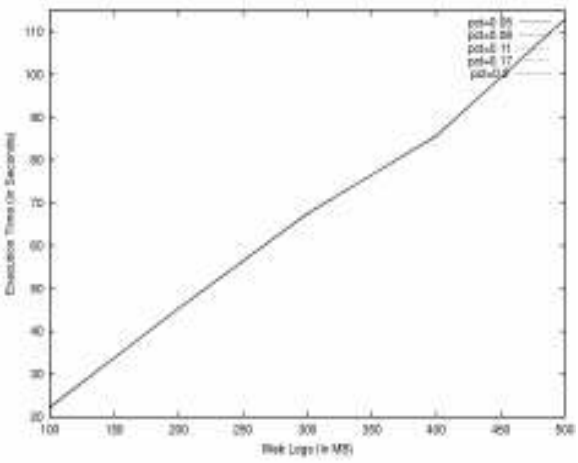
Performance Analysis

Using the five Web logs *L100MB, L200MB, L300MB, L400MB* and *L500MB* as mentioned in the fifth section, the algorithms SbSfxMiner and HbSfxMiner are compared with the Ukkonen algorithm (Ukkonen, 1995) and the SelectiveScan algorithm (Chen, Park & Yu, 1998). Since it was known that SelectiveScan is more efficient than FullSacn (Chen, Park & Yu, 1998), it is not necessary to compare the new algorithms with FullScan. For SelectiveScan, only times for *pct=0.*2 are shown. It has much worse performance for smaller values of *pct*. The computing environment is the same that in Session 4, and all programs were also implemented using Microsoft Visual C++ 6.0. The empirical results are shown in Figure 5.

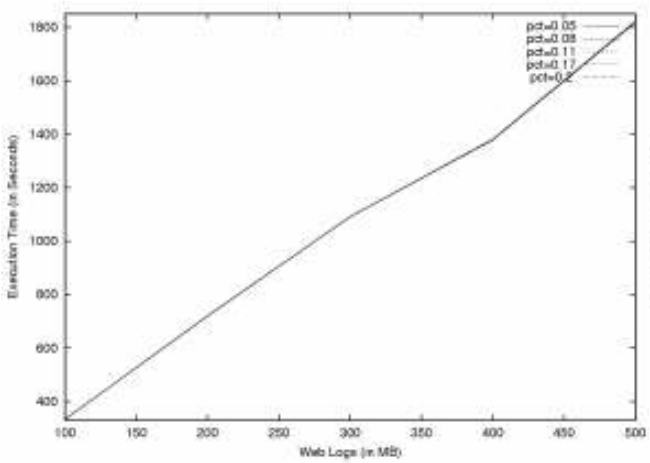
It is clear that the new algorithms have the best performance. When carefully examined, algorithm SbSfxMiner performs even better than HbSfxMiner. The reason is that the computing of the underlying mapping function, though a constant time for each computation, exceeds the sorting time when accumulated for so many substrings in the file.



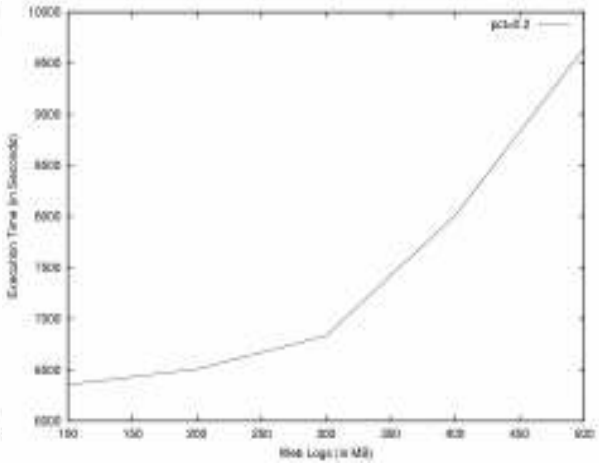
(a) Algorithm SbSfxMiner



(b) Algorithm HbSfxMiner



(c) Ukkonen Algorithm



(b) Algorithm SelectiveScan

Figure 5. Performance of Four Algorithms

CONCLUSIONS

The problem of mining frequent traversal path patterns from very large Web logs is fundamental in Web Mining. In this chapter, two algorithms ISMFR and GSMFR are designed for finding maximal forward references from very large Web logs, and two other algorithms SbSfxMiner and HbSfxMiner are devised for mining frequent traversal path patterns from large sets of maximal forward references. When SbSfxMiner and HbSfxMiner are respectively combined with algorithms ISMFR or GSMFR, two algorithms are obtained for mining frequent traversal path patterns from Web logs directly. These two algorithms have respectively linear and sublinear complexity, and have substantially better performance than the two apriori-like algorithms in (Chen, Park & Yu, 1998) and the Ukkonen algorithm (Ukkonen, 1995) in the case of mining frequent traversal patterns.

Acknowledgment. The authors thank the two anonymous referees and Professor Anthony Scime for their valuable comments on the draft of this chapter. The work of the first two and the last authors is supported in part by the Computing and Information Technology Center of the University of Texas-Pan American.

REFERENCES

Berendt, B., Mobasher, B., Spiliopoulou, M., & Wiltshire, J. (2001). Measuring the accuracy of sessionizers for web usage analysis. Proc. of the Workshop on Web Mining at the First SIAM International Conference on Data Mining. 7-14.

Borges, J., & Levene, M. (1999). Data mining of user navigation patterns. Proc. of the WEBKDD'99 Workshop on Web Usage Analysis and User Profiling. 31-39.

Buchner, A.G., Baumgarten, M., & Anand, S.S. (1999). Navigation pattern discovery from internet data. Proc. of the WEBKDD'99 Workshop on Web Usage Analysis and User Profiling. 25-30.

Buchner, A.G., & Mulvenna, M.D. (1998). Discovering internet marketing intelligence through online analytical web usage mining. ACM SIGMOD RECORD. 54-61.

Catledge, L., & Pitkow, J. (1995). Characterizing browsing behaviors on the world wide web. Computer Networks and ISDN Systems 27(6), 1065-1073.

Chen, Z., Fowler, R.H., & Fu, A. (2003). Linear time algorithms for finding maximal forward references. Proc. of the IEEE Intl. Conf. on Info. Tech.: Coding & computing. 160-164.

Chen, Z., Fowler, R.H., Fu, A., & Wang, C. (2003a). Fast construction of generalized suffix trees over a very large alphabet. Proc. of the Ninth Intl. Computing and Combinatorics Conference, Lecture Notes in Computer Science LNCS 2697, 284-293.

Chen, Z., Fowler, R.H., Fu, A., & Wang, C. (2003b). Linear and sublinear time algorithms for mining frequent traversal path patterns from very large Web logs. Proc. of the Seventh Intl. Database Engineering & Applications Symposium.

Chen, Z., Fu, A.& Tung, F. (2003).Optimal algorithms for finding user access sessions from very large Web logs. World Wide Web 6(3), 259-279.

Chen, M.S., Park, J.S. & Yu, P.S. (1998). Efficient data mining for path traversal patterns. IEEE Transactions on Knowledge and Data Engineering. 10(2), 209-221.

Cooley, R., Mobasher, B. & Srivastava, J. (1997). Web mining: Information and pattern discovery on the world wide web. Proc. of the IEEE Intl. Conference Tools with AI. 558-567.

Cooley, R., Mobasher, B. & Srivastava, J. (1999). Data preparation for mining world wide web browsing patterns. Journal of Knowledge and Information Systems}, 1(1), 5-32.

Gusfield, D. (1997). Algorithms on Strings, Trees, and Sequences. Cambridge University Press.

Kosala, R. & Blockeel, H. (2000). Web mining research: A survey. SIGKDD Explorations, 2(1), 1-15.

Hunt, E., Atkinson, M.P. & Irving, R.W. (2001). A database index to large biological sequences, Proc. of the 27th Intl. Conf. on Very Large Data Bases. 139-148.

Masseglia, F., Poncelet, P. & Cicchetti, R. (1999). An efficient algorithm for Web usage mining. Networking and Information Systems Journal, 2(5-6), 571-603.

McCreight, E.M. (1976). A space-economical suffix tree construction algorithm, Journal of Algorithms. 23(2), 262-272.

Padmanabhan, V.N. & Mogul, J.C. (1996). Using predictive prefetching to improve World Wide Web latency, Computer Communication Review, 26, 22-36.

Perkowitz, M. & Etzioni, O. (1998). Adaptive web pages: Automatically synthesizing web pages. Proc. of AAAI/IAAI'98. 727-732.

Pitkow, J. & Pirolli, P. (1999). Mining longest repeating subsequences to predict World Wide Web Surfing. Proc. of the Second USENIX Symposium on Internet Technologies & Systems. 11-14.

Srivastava, J., Cooley, R., Desphande, M. & Tan, P. (2000). Wen usage mining: Discovery and applications of usage patterns from Web data. SIGKDD Exploration. 1(2), 12-23.

Su, Z., Yang, Q., Lu, Y.& Zhang, H. (2000). WhatNext: A prediction system for Web requests using N-gram sequence models. Proc. of the First International Conference on Web Information Systems Engineering. 200-207.

Ukkonen, E. (1995). On-line construction of suffix trees. Algorithmica. 14(3), 249-260.

Weiner, P. (1973). Linear pattern matching algorithms. Proc. of the 14th IEEE Annual Symp. on Switching and Automata Theory. 1-11.

Xiao, Y. & Dunham, M. (2001). Efficient mining of traversal patterns. Data & Knowledge Engineering, 39, 191-214.