Linear and Sublinear Time Algorithms for Mining Frequent Traversal Path Patterns From Very Large Web Logs

Zhixiang Chen* Richard H. Fowler* Ada Wai-Chee Fu† Chunyue Wang*
*Department of Computer Science, University of Texas-Pan American, Texas, USA
chen@cs.panam.edu, fowler@panam.edu, cwang@panam.edu
† Department of Computer Science, Chinese University of Hong Kong, Hong Kong
adafu@cse.cuhk.edu.hk

Abstract

This paper aims for designing algorithms for the problem of mining frequent traversal path patterns from very large Web logs with best possible efficiency. We devise two algorithms for this problem with the help of fast construction of "shallow" generalized suffix trees over a very large alphabet. These two algorithms have respectively provable linear time and sublinear complexity, and their performance is analyzed in comparison with the two apriori-like algorithms in [4] and the well-known Ukkonen algorithm for on-line suffix tree construction [13]. It is shown that these two algorithms are substantially efficient than the two apriori-like algorithms and the Ukkonen algorithm. The linear time algorithm has optimal performance in theory, while the sublinear time algorithm has better empirical performance.

1. The Problem Formulation

The Web has a natural graph structure: pages are in general 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 that the user is 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 what the user is looking for. 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. As thus, the last reference in a maximal forward sequence indicates a content page [4, 5] that is desired by the user. Under such understanding, when a user searches for desired information, her information needs can be modeled by a set of maximal forward references occurred during her search process.

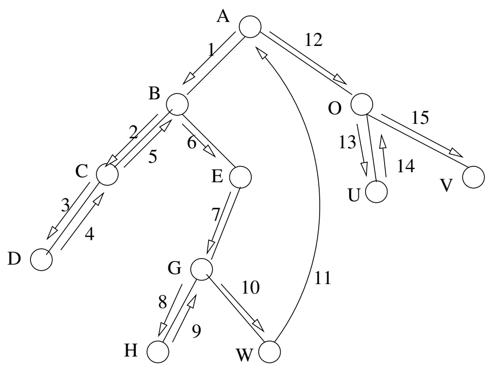


Figure 1. An illustration of traversal path patterns

Suppose, like in [4], 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*, we have 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*, we 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, we have 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., a *"hot"* access patterns of users. Our goal in this paper is to design algorithms for mining frequent traversal path patterns, i.e., frequent or maximal reference sequences, with best possible efficiency. As pointed out in [4], once large reference sequences are determined, maximal reference sequences can be obtained in a straightforward manner.

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 paper, we assume that our Web logs are referrer logs.

2. The Related Work

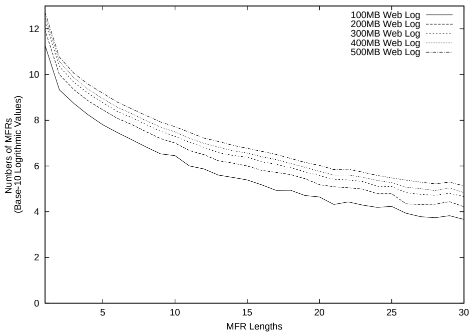
The problem of mining frequent traversal path patterns was initially investigated in [4]. As the first step, an algorithm for finding maximal forward references from a Web log was designed in that paper. That algorithm works in two phrases. First, the log is sorted according to user IDs to group every 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 performance of these two algorithms were analyzed. [5] has detailed discussions of various data preparation tasks. The maximal forward references, called *"transactions"*, was 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 [15] studied the application of generalized suffix trees to mining frequent user access patterns from Web logs, however the patterns in [15] are not traversal path patterns as considered in the usual settings in many literatures such as [4, 9, 12, 8, 11] and the context of this paper. The patterns in that paper are in essence frequent consecutive subsequences of user access sessions, while traversal paths corresponding to the underlying hyperlink structure were not concerned with. For the example illustrated in Figure 1, that paper concerns with frequent consecutive subsequences of $\{$*A,B,C,D,C,B,E,G,H,G,W,A,O,U,O,V*$\}$, but not frequent consecutive sequences of the maximal forward references $\{$*ABCD, ABEGH, ABEGW, AOU, AOV*$\}$. In other words, it does not concern with the underlying traversal paths as shown in Figure 1. Furthermore, that paper has not studied how to overcome the well-understood obstacles of suffix tree construction (see, e.g., [6], pages 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. Examples of such work include [9, 8, 11, 12]. The N-gram approaches were used to find most frequent subsequences of the paths, and sometimes the most frequent longest subsequences are the most interested. Once those subsequences were 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, such as in [7]. They also studied how to identify the longest repeating subsequences of the ordered sessions so as to find associations of those sequences. In all those papers, a user's surfing path is considered as an ordered sequences of consecutive references. In other words, a user's surfing path is the user's access session with order. This is different from the maximal forward references studied in [4, 5] and in the present paper.

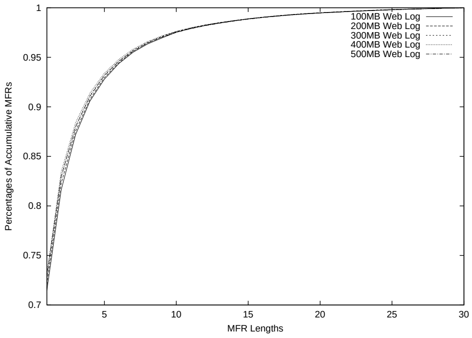
In [3], optimal algorithms were designed for finding user access sessions from very large Web logs. Since maximal forward references are finer level characterizations of sessions as carefully examined in [5], the algorithms in [3] can not be applied directly to the problem of finding maximal forward references.

3. Properties of Maximal Forward References

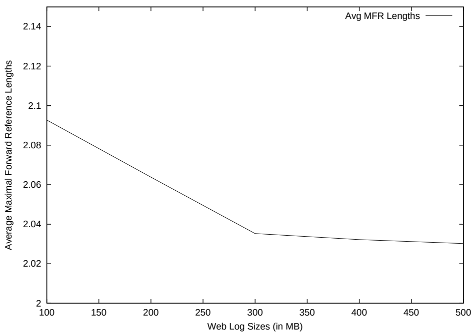
We have examined properties of maximal forward references of the five logs *L100MB, L200MB, L300MB, L400MB* and *L500MB*. When the parameters α and β are set to 30 minutes to define interval sessions and gap sessions [3], it is obvious that the length of a maximal forward reference, or the number of URLs in it, is small. Statistical properties of maximal forward references are shown in Figure 2. In summary, we have observed that (1) almost all maximal forward references have a length ≤ 30; more than 90% max-



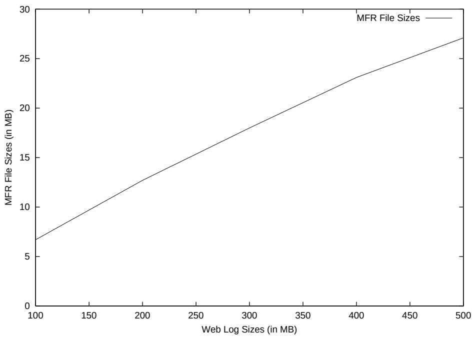
(a) MFR Distributions



(b) Accumulative MFR Distributions



(c) Average Lengths



(d) MFR File Sizes

Figure 2. Properties of Maximal Forward References (MFRs)

imal forward references have lengths less than or equal to 4, and the average length is about 2; and (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 megabytes to 27.1 megabytes. we must caution readers that the above three properties are not typical to our Web logs. These are common characteristics of large Web logs. That is, similar properties hold for other large Web logs.

4. "Shallow" Generalized Suffix Trees Over a Very Large Alphabet

Suffix tree are old data structures that become new again and have found many applications in data mining and bioinformatics as well. The first linear-time algorithm for constructing suffix trees was given by Weiner in [14] in 1973. A different but more space efficient algorithm was given by McCreight in [10] in 1976. Almost twenty years later Ukkonen 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 $\{S_1, S_2, \ldots, S_n\}$ 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 $S_1, S_2, \ldots, S_n$, is called a *"generalized"* suffix tree. Usually, a set of strings $S_1, S_2, \ldots,$ and S_n, is represented as $S_1\$S_2\$\ldots\$S_n\$$.

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 a maximal forward references when maximal forward references are understood as strings of URLs. In Figure 3, we illustrate a generalized suffixed tree *"mississippi$missing$sipping$"*. Notice that in our 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 $S_1, S_2, \ldots, S_n$ over an alphabet Σ, let $m = \sum_{i=1}^{n} |S_i|$. It is well-understood (see, e.g., [6], pages 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 were not considered when trees are large 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 $|\Sigma|$ are very large, the time complexity of those classical algo-

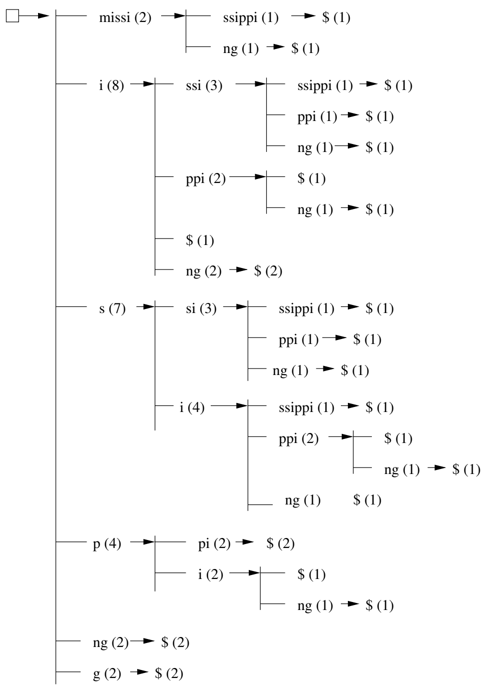


Figure 3. A Generalized Suffix Tree

rithms is $O(m|\Sigma|)$, and the space complexity is $\theta(m|\Sigma|)$. In particular, as pointed in Section 3, the alphabet Σ in our case of mining frequent traversal path pattern is 11,926 (the number of unique URLs) which is too big to be ignored, and m ranges from 6.7 megabytes to 27.1 megabytes and hence the $O(m|\Sigma|)$ space requirement are 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 we have learned from Section 3, in our 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 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 small. 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 our logs, the width of the first level of the tree is $|\Sigma| = 11,926$; the height of each subtree t_i, $1 \leq i \leq |\Sigma|$, is no more than 30; more than 90% of subtrees have a height ≤ 4; and the average height is just 2. In general, if a generalized suffix tree has a depth bounded by some constant, then we call it a *"shallow"* generalized suffix tree.

5. Mining Frequent Traversal Path Patterns

In [1], we have designed two linear time algorithms for finding maximal forward references from very large Web logs. We assume in this section that maximal forward references are obtained by either of the two algorithms in [1]. In [2], we have studied fast construction of shallow generalized suffix trees over a very large alphabet. We shall use our techniques developed in [2] to design algorithms with 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), we use $S[1:n]$ to denote S, where $s[i]$ is the ith URL in S. We use 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, we use letters for URLs, and substrings for maximal forward references.

```
input: infile, tmpfile, pct, outfile
while (infile is not empty)
        readMFR(infile, s)
        for (i = 1; i ≤ n; i++)
                tmpfile.append(s[i : n])
sort(tmpfile); createSuffixTree(sft)
while (tmpfile is not empty)
        readMFR(tmpfile, s)
        if (sft.empty() or sft.first() == s[1])
                RstUkkonen(sft, s)
        else if (sft.first() ≠ s[1])
                traverseOutput(sft,outfile,pct)
                createSuffixTree(sft);
traverseOutput(sft,outfile,pct)
```

Figure 4: Algorithm SbSfxMiner

Consider a generalized suffix tree *sfx* of a set of strings. For each subtree t of *sfx*, let $t.first()$ denote the first letter occurred at t. As carefully examined in [2], t contains exactly all suffixes starting with the letter $t.first()$, and any two distinct subtrees must have distinct first letters. 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 3 is m, and all suffixes starting with m in the strings "$mississippi\$missing\$sipping\$$" are contained in this subtree. This critical observation leads to the designs of our new algorithms.

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

5.1. Algorithm SbSfxMiner

In the algorithm, *infile* is the MFR file, *tmpfile* a temporary file, *pct* a frequency threshold parameter, and *outfile* the frequent MFR pattern file. The key idea is as follows. We read strings sequentially from an input file, and for every string $s[1 : n]$ we output its n suffixes to a temporary file. We then sort the temporary file to group all the suffixes starting with the same letter together. Next, we build a generalized suffix tree for each group of such suffixes (precisely, our tree here is a subtree of the conventional generalized suffix tree). Finally, we traverse the tree to output all frequent (or frequent longest) substrings. It follows from the properties of maximal forward reference in Section 3 that the size of the temporary file is about $O(N)$. Sorting this file takes $O(N \log N)$ time. By the property of Ukkonen algorithm, building a generalized suffix tree for a group of strings takes $O(N)$. Hence, the whole process is of sublinear time. The description of SbSfxMiner is given in Figure 4.

```
input: infile, f, pct, outfile
createSuffixTrees(sft, size)
while (infile is not empty)
        readMFR(infile, s)
        for (i = 1; i ≤ n; i++)
                RstUkkonen(sft[f(s[i])], s[i:n])
for (i = 0; i < size; i++)
        traverseOutput(sft[i],outfile,pct)
```

Figure 5: Algorithm HbSfxMiner

5.2. Algorithm HbSfxMiner

Here, *infile*, *pct*, and *outfile* are the same as those in algorithm SbSfxMiner, and f is a hash function. The key idea 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 some nice directory path structure, such a function can be designed with constant time complexity. In the general case, we may use a hashing function to replace the function so that it will be more efficient in computing hashing values of letters. The detailed description of the algorithm is given in Figure 5. Because sorting is not required, the time complexity of this algorithm is linear.

5.3. Performance Analysis

We used the five Web logs *L100MB, L200MB, L300MB, L400MB* and*L500MB* to analyze the performance of algorithms SbSfxMiner and HbSfxMiner in comparison with the Ukkonen algorithm [13] and the SelectiveScan algorithm [4]. Since it was known that SelectiveScan is more efficient than FullSacn [4], we decide not to compare our algorithms with FullScan. For SelectiveScan, only times for $pct = 0.2$ is shown. It has much worse performance for other smaller values of pct. The empirical results are shown in Figure 6.

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

6. Conclusions

The problem of mining frequent traversal path patterns from very large Web logs is fundamental in Web Mining. In this paper we design two algorithms for this problem. These two algorithms have respectively linear and sublinear complexity, and have substantially better performance than the two apriori-like algorithms in [4] and the well-known Ukkonen algorithm [13] in the case of mining frequent traversal patterns.

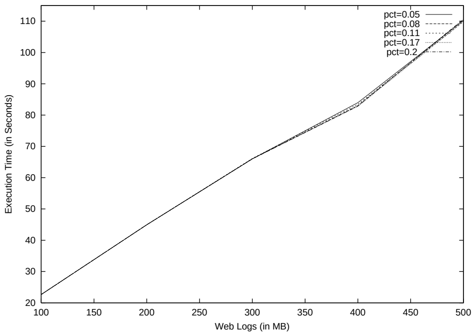
ACKNOWLEDGMENT. The work of the first two and the last authors is supported in part by the the UTPA Computing and Information Technology Center. The work of the third author is supported by the CUHK RGC Research Grant Direct Allocation ID 2050279.

References

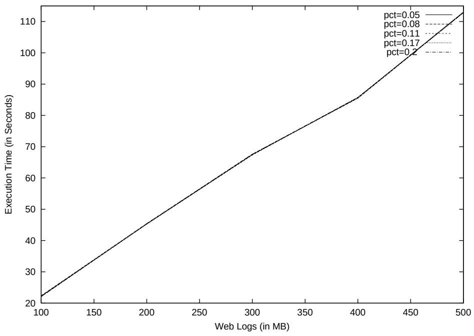
[1] Z. Chen, R. Fowler, and A. Fu, Linear time algorithms for finding maximal forward references, Proc. of the IEEE Intl. Conf. on Info. Tech.: Coding & computing (ITCC 2003), 2003.

[2] Z. Chen, R. Fowler, A. Fu, and C. Wang, Fast construction of generalized suffix trees over a very large alphabet, Proceedings of the Intl. Computing and Combinatorics Conf. (COCOON03), 2003.

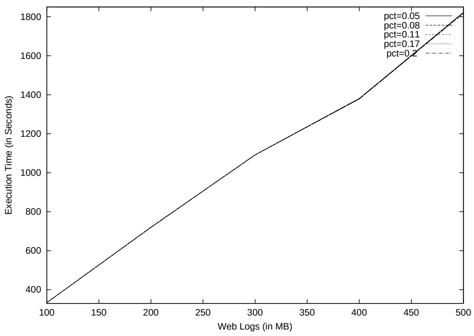
[3] Z. Chen, A. Fu, and F. Tong, Optimal algorithms for finding user access sessions from very large Web logs, Advances in Knowledge Discovery and Data Mining/PAKDD'02, Lecture Notes in Computer Science 2336, pages 290-296, 2002. (The full version will apear in WWW Journal: Internet and Information Systems, 2003.)



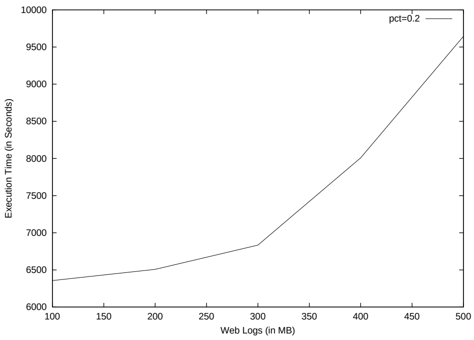
(a) Algorithm SbSfxMiner



(b) Algorithm HbSfxMiner



(c) Ukkonen Algorithm



(d) Algorithm SelectiveScan

Figure 6. Performance of Four Algorithms

[4] M.S. Chen, J.S. Park, and P.S. Yu. Efficient data mining for path traversal patterns. *IEEE Transactions on Knowledge and Data Engineering*, 10:2:209-221, 1998.

[5] R. Cooley, B. Mobasher, and J. Srivastava. Data preparation for mining world wide web browsing patterns. *Journal of Knowledge and Information Systems*, 1:1, 1999.

[6] D. Gusfield, *Algorithms on Strings, Trees, and Sequences*, Cambridge University Press, 1997.

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

[8] V.N. Padmanabhan and J.C. Mogul, Using predictive prefetching to improve World Wide Web latency, Computer Communication Review, 26, 22-36, 1996.

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

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

[11] S. Schetcher, M. Krishnan, and M.D. Smith, Using path profiles to predict HTTP requests, Proc. of the Seventh International World Wide Web Conference, Computer Networks, 30(1-7), 457-467, 1998.

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

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

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

[15] Y. Xiao and M.H. Dunham, Efficient mining of traversal patterns, Data & Knowledge Engineering, 39, 191-214, 2001.

[16] O. Zaïane, M. Xin, and J. Han. Discovering web access patterns and trends by applying olap and data mining technology on web logs. *Advances in Digital Libraries*, pages 19-29, April, 1998.