Proc. of the 9th Intl Comput and Combinatorics Conf (COCOON03)

Fast Construction of Generalized Suffix Trees Over a Very Large Alphabet

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

[1] Department of Computer Science, University of Texas-Pan American, Edinburg TX 78539 USA. chen@cs.panam.edu, fowler@panam.edu, cwang@panam.edu
[2] Department of Computer Science, Chinese University of Hong Kong, Shatin, N.T., Hong Kong. adafu@cse.cuhk.edu.hk

Abstract. The work in this paper is motivated by the real-world problems such as mining frequent traversal path patterns from very large Web logs. Generalized suffix trees over a very large alphabet can be used to solve such problems. However, traditional algorithms such as the Weiner, Ukkonen and McCreight algorithms are not sufficient assurance of practicality because of large magnitudes of the alphabet and the set of strings in those real-world problems. Two new algorithms are designed for fast construction of generalized suffix trees over a very large alphabet, and their performance is analyzed in comparison with the well-known Ukkonen algorithm. It is shown that these two algorithms have better performance, and can deal with large alphabets and large string sets well.

1 Introduction and Problem Formulation

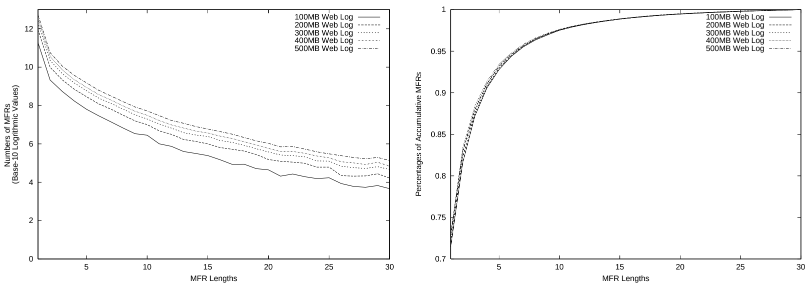
1.1 Introduction

Recently, suffix trees have found many applications in bio-informatics, data mining and knowledge discovery. The first linear-time algorithm for constructing suffix trees was given by Weiner in [17] in 1973. A different but more space efficient algorithm was given by McCreight in [13] 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 [16]. 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.

Typical applications of generalized suffix trees include the identification of frequent (or longest frequent) substrings in a set of strings. One particular example of such applications is the mining of frequent traversal path patterns of

Web users from very large Web logs [7, 5], because such patterns are frequent (or longest frequent) substrings in the set of maximal forward references of Web users when maximal forward references are understood as strings of URLs. Such discovered patterns (or 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 [7]. It has significant potential to reduce, through prefetching and caching, Web latencies that have been perceived by users year after year [12]. It can also help the administrative personnel to predict the trends of the users' needs so that they can adjust their products to attract more users (and customers) now and in the future [2]. Other examples include document clustering, where a short summary of a document is viewed as a string of keywords and a generalized suffix tree is built for a set of such strings to group documents into different clusters.

In the mining of frequent traversal path patterns, specific properties exist for the data set. As investigated in [5], maximal forward references (MFRs) of Web logs exhibit properties as shown in Figure 1. In summary, the following properties hold: (1) The size of the set of strings (or maximal forward references) is very large, ranging from megabyte magnitude to gigabyte magnitude. (2) The size of the alphabet (or the number of unique URLs) is very large, ranging from thousands to tens of thousands or more. (3) All strings have a length ≤ 30 (derived from the given parameter setting of Web log sessionization). (4) More than 90% strings have lengths less than or equal to 4, and the average length is about 2.04.



(a) MFR Distributions (b) Accumulative MFR Distributions

Fig. 1. Properties of Maximal Forward References (MFRs)

1.2 Problem Formulation

Throughout this paper, we use Σ to denote an alphabet, and let $|\Sigma|$ denote the size of Σ. For any string $s \in \Sigma^*$, let $|s|$ denote its size, i.e., the number of all occurrences of letters in s. For any set of strings $\mathcal{S}$, let $|\mathcal{S}| = \sum_{s \in \mathcal{S}} |s|$. When

$\mathcal{S}$ is stored as a file, we also refer $|\mathcal{S}|$ as the number of bytes of $\mathcal{S}$. Motivated by real world problems such as mining frequent traversal path patterns, in this paper we study fast construction of generalized suffix trees for a set $\mathcal{S}$ of strings over an alphabet Σ under the following conditions:

Conditions.
1. $|\Sigma|$ is very large, ranging from thousands to tens of thousands or more.
2. $|\mathcal{S}| = \sum_{s\in\mathcal{S}} |s|$, the size of the set of of strings, is very large, ranging from megabyte magnitudes to gigabytes magnitudes or more.
3. For each string $s \in \mathcal{S}$, $|s| \leq \alpha$, where α is a small constant.

We shall point out that depending on concrete applications, more restrictions can be added to the third condition. For example, in the case of mining frequent traversal path patterns, we can further require that 90% of strings in $\mathcal{S}$ have a length ≤ 4 and the average string length in $\mathcal{S}$ is about 2.04.

We now give several formal definitions. Unlike traditional suffix trees, in this paper we additionally require that counting information of substrings are recorded at internal nodes and leaves as well.

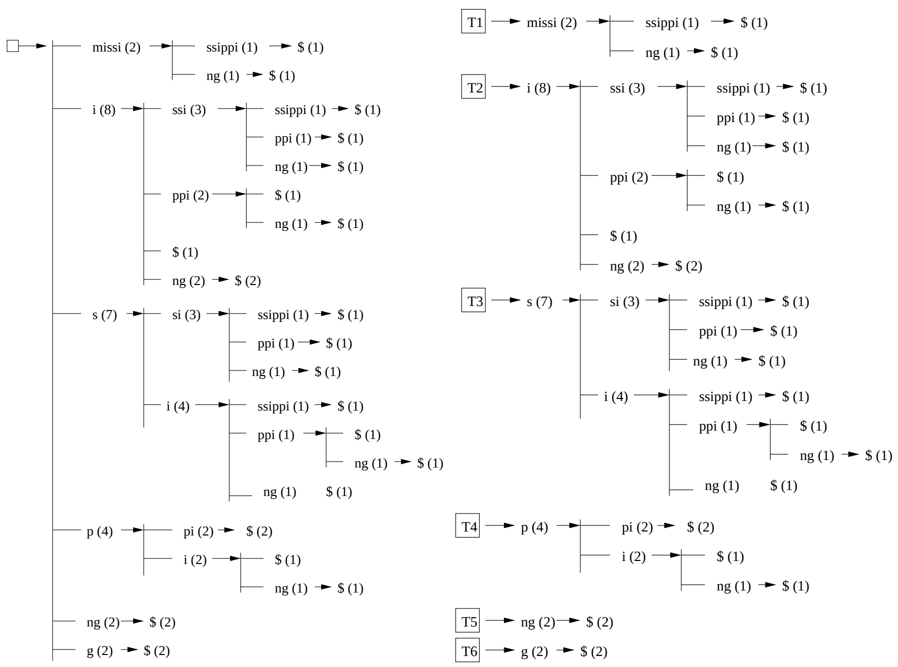
Definition 1. *For any string $s \in \Sigma$, we also denote $s = s[1..n]$ where $n = |s|$. For every i, j with $1 \leq i \leq j \leq n$, $s[i]$ is the i-th letter in s, and $s[i..j]$ is the substring from the i-th letter to the j-th letter. Note that $s[i..n]$ is a suffix starting at the i-th letter.*

Definition 2. *A suffix tree $\mathcal{T}$ for a string $s[1..n]$ over a given alphabet Σ is a rooted directed tree with exactly n leaves. Each internal node other than the root has at least two children and each edge is labeled with a nonempty substring of s. No two edges out of a node can have edge labels starting with the same letter. Each internal node or leaf has a counter to indicate the number of times the concatenation of the edge labels on the path from the root to the node or leaf occurs in the string s. The key feature of the suffix tree is that the concatenation of the edge labels on each path from the root to one of the n leaves represents exactly one of n suffixes of s.*

Definition 3. *Given a set of strings $\mathcal{S} = \{s_1, s_1, \ldots, s_m\}$ over an alphabet Σ, a suffix tree $\mathcal{T}$ for first string s_1 can be generalized to represent all suffixes and to record the counting information of substrings in the set of strings. The key feature of such a tree is that the concatenation of the edge labels on each path from the root to one of the leaves represents exactly a distinct suffixes in $\mathcal{S}$, and every suffix in $\mathcal{S}$ is represented by exactly one of such concatenations. Such a tree is called a "generalized" suffix tree. Usually, we assume that $\$ \notin \mathcal{S}$, and $\mathcal{S}$ is represented as $s_1\$s_2\$\ldots\$s_m\$$.*

In Figure 2(a), we illustrate a generalized suffix tree for *"mississippi\$missing\$sipping\$"*.

The goal of this paper is to design algorithms for fast construction of generalized suffix trees under Conditions (1) to (3). A sorting-based algorithm SbSfxTree (Sorting-based Suffix Tree) and a hashing-based algorithm HbSfxTree



(a) *"mississippi$missing$sipping$"* (b) Subtrees of the big tree in (a)

Fig. 2. A Generalized Suffix Tree and Its Subtrees

(Hashing-based Suffix Tree) will be devised and their performance will be analyzed comparatively. It is shown that algorithms SbSfxTree and HbSfxTree are substantially faster than Ukkonen's algorithm. Furthermore, these two algorithms have superior space scalable performance and can be easily tuned to parallel or distributed algorithms.

The rest of the paper is organized as follows. In Section 2, we will review practical implementation challenges of suffix tree construction. Some properties of suffix trees are given in Section 3. Algorithms SbSfxTree and HbSfxTree are devised in Section 4. Performance analysis is given in Section 5. Finally, we conclude the paper in Section 6.

2 Practical Implementation Challenges

As well discussed in Gusfield [8] (pages 116 to 119), the Weiner, McCreight, and Ukkonen algorithms [17, 13, 16] have ignored the size of the alphabet Σ, and have not considered memory paging when trees are large and hence cannot be stored in RAM. When the size of Σ is too large to be ignored, those three algorithms all require $\theta(|\mathcal{S}| \cdot |\Sigma|)$ space, or the linear time bound $O(|\mathcal{S}|)$ should be replaced with $min\{O(|\mathcal{S}| \cdot \log |\mathcal{S}|), O(|\mathcal{S}| \cdot \log |\Sigma|)\}$.

The main design issues in all the three well known algorithms [17, 13, 16] are how to represent and search the branches out of the nodes of the tree. For example, in the Ukkonen algorithm, in order to achieve linear space complexity, array indexes are used to represent substrings labeling tree edges under the implicit assumption that the whole string (or the set of strings) is kept in RAM and represented as an array; and in order to achieve linear time complexity, suffix links are used to allow quick walks from one part of the tree to another part under the implicit assumption that the entire tree is kept in RAM. Those design techniques are great for theoretical time/space bounds, but are inadequate for paging if the string (or the set of strings) or the entire tree cannot be stored in RAM. Because of those algorithms' dependence on the availability of the entire string (or the set of strings) and the entire tree in RAM, and because of the tree's lack of nice locality properties, those algorithms cannot support parallel or distributed construction of the tree. Therefore, new techniques are needed for implementing generalized suffix trees for very large sets of strings over a very large alphabet.

Gusfield [8] summaries four basic alternative techniques for represent branches in order to balance the constraints of space against the need for speed. The first one is to use an array of size $\theta(|\Sigma|)$ at each non-leaf node to represent branches to children nodes. The second is to use linked list to replace array in the first technique. The third is to replace the linked list at each non-leaf node with some balanced tree. Finally, the last is to use hashing at each non-leaf node to facilitate branch search. However, all the above alternative techniques fail to overcome the dependence on the availability of the entire string (or the set of strings) and the entire tree in RAM. The first three also increase the burden of space demand when the alphabet is very large. The challenge for the last one is to find a hashing scheme to balance space with speed. These techniques cannot facilitate the parallel or distributed construction of the trees.

3 Some Properties

Let $\mathcal{T}$ be a generalized suffix tree for a set of strings $\mathcal{S}$ over the alphabet Σ. For each child node v of the root of $\mathcal{T}$, we can obtain a subtree for v by simply removing all other children nodes and their descendants as well as their edges. E.g., Six subtrees are shown in Fig.2(b) for the tree in Fig.2(a). For each t of such subtrees, it is obvious that the root of t has exactly one edge leading to its only child node or to its only leaf. Let $t[1]$ denote the first letter in the string on the edge out of the root. We say a string s is contained in a subtree t if s is the concatenation of all edge-labels on a path from the root of t to some leaf of t.

Lemma 1. *Let W be the number of distinct letters that appear in the set of strings $\mathcal{S}$. $\mathcal{T}$ has exactly W many subtrees. Moreover, for any two distinct subtrees t' and t'' of $\mathcal{T}$, $t'[1] \neq t''[1]$.*

Proof Sketch. Directly from Definitions 2 and 3.

We may assume without loss of generality that $\mathcal{S}$ contains every letter in Σ (otherwise, a smaller Σ can be used). Lemma 1 means that $\mathcal{T}$ has exactly $|\Sigma|$ many subtrees, each of which starts with a letter in Σ.

Lemma 2. *For any subtree t of $\mathcal{T}$, and for any suffix s in any string of $\mathcal{S}$, t contains s if and only if $s[1] = t[1]$.*

Proof Sketch. If t contains s, then s is the concatenation of edge labels on a path from the root of t to some leaf of t, thus we have $s[1] = t[1]$. By Definitions 2 and 3, s is the concatenation of edge labels on a path from the root of $\mathcal{T}$ to one of T's leaves. Let t' be the subtree of $\mathcal{T}$ that has the path representing s, then we have $s[1] = t'[1]$. If $s[1] = t[1]$, then $t[1] = t'[1]$, hence it follows from Lemma 1 that $t = t'$, i.e., t contains s.

Corollary 1. *Let t be any subtree of $\mathcal{T}$, and s be any substring in a string of $\mathcal{S}$. Then, s is the concatenation of edge labels on a path from the root of t to some internal node (or leaf) of t, and the frequency of s is recorded at the node (or leaf) counter, if and only if $s[1] = t[1]$.*

Proof Sketch. By Definitions 2 and 3, Lemma 2 and the fact that any substring in a string of $\mathcal{S}$ is a prefix of some suffix in a string of $\mathcal{S}$.

4 New Algorithms

4.1 The Strategy

Lemma 2 and Corollary 1 combined imply a new way of fast construction of generalized suffix trees. The strategy is to organize all suffixes starting with the same letter into a group and build a subtree for each of such groups. A more or less related strategy has been devised in [9], but the strings considered there are over a small alphabet and the method used to build subtrees is of quadratic time complexity.

The task of grouping can be done by means of sorting or hashing. We shall point out that the hashing here is substantially different from other hashing techniques used to improve the performance of suffix tree construction [8]. We use hashing here for the purpose of *"divide-and-conquer"*, while others use hashing to speed up searching the branches out of the nodes. The task of constructing a subtree can be done easily, say, with one phrase execution of the Ukkonen Algorithm. Recall that the Ukkonen algorithm builds a suffix tree for a string $s[1:n]$ in n phrases with the i-th phrase adding the i-th suffix $s[i:n]$ to the existing (but partially built) suffix tree. Let *ResUkkonen(SuffixTree t, NewString s)* denote the one phrase execution of the Ukkonen Algorithm to add the only suffix $s[1:n]$ to t. We additionally require that ResUkkonen records frequencies of substrings at internal nodes and leaves, which can be done easily by tuning the Ukkonen algorithm.

4.2 Algorithm SbSfxTree

The key idea 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. Sort the temporary file to group all the suffixes starting with the same letter together. Finally, build a subtree for each group of such suffixes.

```
input:
    infile: a set of strings
    tmpfile: a set of suffixes
    outfile: a set of subtrees
Begin
    while (infile is not empty)
        readString(infile, s[1:n])
        for (i = 1; i ≤ n; i++)
            tmpfile.append(s[i : n])
    sort(tmpfile); createSuffixTree(t)
    while (tmpfile is not empty)
        readString(tmpfile, s)
        if (t.empty() or t[1] == s[1])
            RstUkkonen(sft, s)
        else if (t[1]≠ s[1])
            Output(t,outfile), ResetTree(t)
    Output(t,outfile)
end
```

Figure 3: Algorithm SbSfxTree

4.3 Algorithm HbSfxMiner

The key idea is to replace sorting with hashing to group all suffixes with the same starting letter together.

```
input:
    infile: a set of strings
    f: a hashing function from letters to integers
    outfile: a set of subtrees
Begin
    create subtrees t_1, ..., t_|Σ|;
    while (infile is not empty)
        readString(infile, s[1:n])
        for (i = 1; i ≤ n; i++)
            RstUkkonen(t_f(s[i]),s[i:n])
    for (i = 0; i < |Σ|; i++)
        Output(t_i,outfile)
end
```

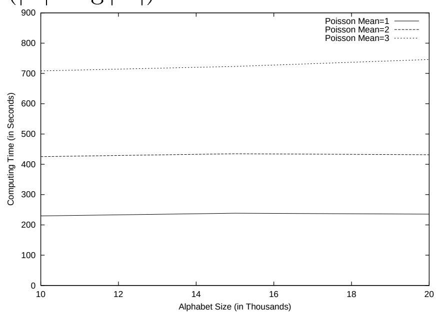
Figure 4: Algorithm HbSfxTree

5 Performance Analysis

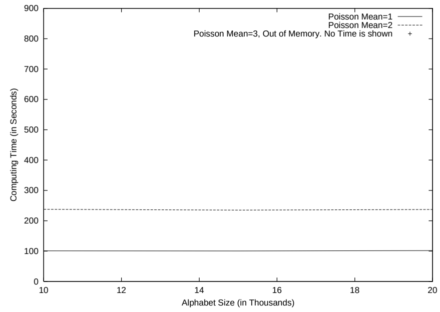
Due to space limit, we only give complexity bounds for algorithm SbSfxTree and would like to point out that similar bounds can be given to algorithm HbSfxTree. We will also present experimental results for both algorithms

Theorem 1. *Let $\mathcal{T}$ be a generalized suffix tree of a set of strings $\mathcal{S}$ over an alphabet Σ satisfying Conditions(1), (2) and (3). Assume the size of each subtree t of $\mathcal{T}$ is $O(|\mathcal{S}|/|\Sigma|)$. Algorithm SbSfxTree builds $\mathcal{T}$ (via building all its subtrees) in time $O(|\mathcal{S}| \cdot \log|\mathcal{S}| + |\mathcal{S}| \cdot \log|\Sigma|)$ and in space $O(|\mathcal{S}|/|\Sigma|)$.*

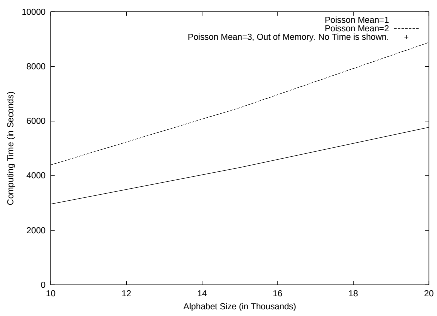
Proof Sketch. By Conditions (1), (2) and (3), each string $s \in \mathcal{S}$ has at most α suffixes. Hence, the size of the suffix file is at most $\alpha|\mathcal{S}|$, this means that sorting to group suffixes is of $O|\mathcal{S}| \cdot \log|\mathcal{S}|)$ time complexity and of $O(|\mathcal{S}|/|\Sigma|)$ space complexity when a buffer of $O(|\mathcal{S}|/|\Sigma|)$ size is used. It follows from the Ukkonen algorithm that building a subtree requires $O(|\mathcal{S}|/|\Sigma|)$ space and $O((|\mathcal{S}|/|\Sigma|) \cdot \log|\Sigma|)$ time. This means that the total space for building all the subtrees is still $O(|\mathcal{S}|/|\Sigma|)$, but the total time is by Lemma 1 $O((|\Sigma| \cdot |\mathcal{S}|/|\Sigma|) \cdot \log|\Sigma|) = O(|\mathcal{S}| \cdot \log|\Sigma|)$.



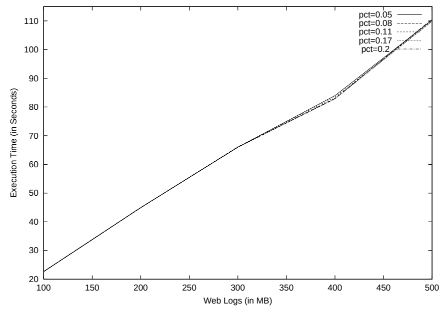
(a) Algorithm SbSfxTree



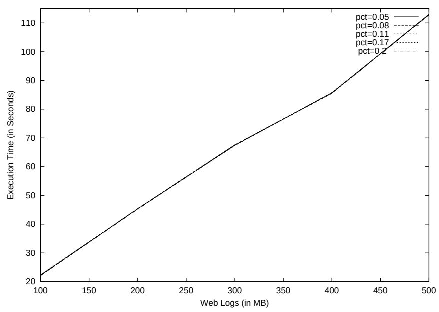
(b) Algorithm HbSfxTree



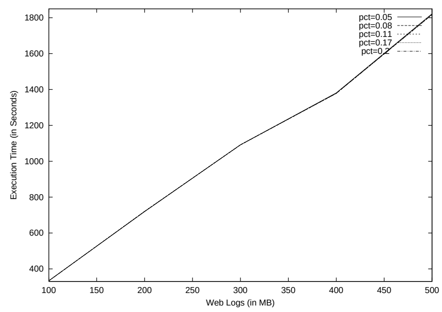
(c) Ukkonen Algorithm



(d) Algorithm SbSfxTree Based Miner



(e) Algorithm HbSfxTree Based Miner



(f) Ukkonen Algorithm Based Miner

Fig. 5. Performance of the Algorithms

In theory, algorithm SbSfxTree has better space complexity, while it has almost the same time complexity bound as the Weiner, Ukkonen, and McCreight algorithms. In Fig.5(a,b,c), we report experimental analysis of SbSfxTree and HbSfxTree in comparison with the Ukkonen algorithm. The computing environment is a Dell PWS 340 with a 1.5 GHz P4 Processor, 512 MB RAM and 18 GB memory. In those experiments, we used alphabets $\Sigma_i, i = 1, 2, 3$, such that $|\Sigma_1| = 10,000$, $|\Sigma_2| = 15,000$ and $|\Sigma_3| = 20,000$. For each Σ_i, we generated three sets of 1 million strings such that sizes of strings follow Poisson distributions with means values of 1, 2 and 3, respectively. It is clear that algorithms SbSfxTree and HbSfxTree have substantially better performance than the Ukkonen algorithm. For the set of strings following Poisson distribution of means value 3, the Ukkonen algorithm ran out of memory. The current version of algorithm HbSfxTree also ran out memory, because all the subtrees were stored in RAM. We shall point out that this can be improved through paging subtrees in the next stage of implementation.

In [5], we has applied algorithms SbSfxTree and HbSfxTree to the mining of frequent traversal path patterns from very large Web logs. Fig.5(d,e,f) shows performance of SbSfxTree and HbSfxTree based mining in comparison with the Ukkonen algorithm based mining. It is clear that SbSfxTree and HbSfxTree are far superior to the Ukkonen algorithm. It is also shown [5] that SbSfxTree and HbSfxTree are far superior to the apriori-like algorithms within the context of mining frequent traversal path patterns.

Remark 1. By Lemma 2, the construction of one subtree has no dependence on any other subtrees. This means that both algorithms SbSfxTree and HbSfxTree can be easily revised to allow parallel or distributed construction of generalized suffix tree.

6 Conclusions

The work in this paper is motivated by the real-world problems such as mining frequent traversal path patterns from very large Web logs. Generalized suffix trees over a very large alphabet can be used to solve such problems. However, due to large magnitudes of the underlying alphabet and the set of strings, traditional algorithms such as the Weiner, Ukkonen and McCreight algorithms are not sufficient assurance of practicality. We have designed two algorithms for fast construction of generalized suffix trees over very alphabet. We have shown that the two algorithms are efficient in theory and in practice, and applied them to solve the problem of mining frequent traversal path patterns.

ACKNOWLEDGMENT. Thank Prof. Ukkonen for sending his work [16] to us. Thank Yavuz Tor for helping us on experiments in Fig.5(a,b,c). 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. The work of the third author is supported by the CUHK RGC Research Grant Direct Allocation ID 2050279.

References

1. J. Borges and M. Levene. Data mining of user navigation patterns. *MS99*, 1999.
2. A.G. Buchner and M.D. Mulvenna. Discovering internet marketing intelligence through online analytical web usage mining. *ACM SIGMOD RECORD*, pages 54-61, Dec. 1998.
3. L. Catledge and J. Pitkow. Characterizing browsing behaviors on the world wide web. *Computer Networks and ISDN Systems*, 27, 1995.
4. 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.
5. Z. Chen, R. Fowler, A. Fu, and C. Wang, Linear and sublinear time algorithms for mining frequent traversal path patterns from very large Web logs, Proceeding of the Seventh International Database Engineering and Applications Symposium, 2003.
6. 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. (Full version will appear in Journal of World Wide Web: Internet and Information Systems, 2003.)
7. 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.
8. D. Gusfield, *Algorithms on Strings, Trees, and Sequences*, Cambridge University Press, 1997.
9. E. Hunt, M.P. Atkinson and R.W. Irving, A database index to large biological sequences, Proceedings of the 27th International Conference on Very Large Data Bases, pages 139-148, 2001.
10. R. Kosala and H. Blockeel, Web mining research: A survey, SIGKDD Explorations, 2(1), pages 1 - 15, 2000.
11. 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.
12. 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.
13. E.M. McCreight, A space-economical suffix tree construction algorithm, Journal of Algorithms, 23(2),pages 262-272, 1976.
14. C. Shababi, A.M. Zarkesh, J. Abidi, and V. Shah. Knowledge discovery from user's web page navigation. *Proceedings of the Seventh IEEE Intl. Workshop on Research Issues in Data Engineering (RIDE)*, pages 20-29, 1997.
15. 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.
16. E. Ukkonen, On-line construction of suffix trees, Algorithmica, 14(3), pages 249-260, 1995.
17. P. Weiner, Linear pattern matching algorithms, Proc. of the 14th IEEE Annual Symp. on Switching and Automata Theory, pages 1-11, 1973.