# Efficient Data Mining for Path Traversal Patterns

Ming-Syan Chen*

Elect. Eng. Department
National Taiwan Univ.
Taipei, Taiwan, ROC
mschen@cc.ee.ntu.edu.tw

Jong Soo Park†

Department of Comput. Sci.
Sungshin Women's Univ.
Seoul, Korea
jpark@cs.sungshin.ac.kr

Philip S. Yu

IBM T.J. Watson Res. Ctr.
P.O.Box 704
Yorktown, NY 10598, U.S.A.
psyu@watson.ibm.com

**Abstract**

In this paper, we explore a new data mining capability which involves mining path traversal patterns in a distributed information providing environment where documents or objects are linked together to facilitate interactive access. Our solution procedure consists of two steps. First, we derive an algorithm to convert the original sequence of log data into a set of maximal forward references. By doing so, we can filter out the effect of some backward references which are mainly made for ease of traveling and concentrate on mining meaningful user access sequences. Second, we derive algorithms to determine the frequent traversal patterns, i.e., large reference sequences, from the maximal forward references obtained. Two algorithms are devised for determining large reference sequences: one is based on some hashing and pruning techniques, and the other is further improved with the option of determining large reference sequences in batch so as to reduce the number of database scans required. Performance of these two methods is comparatively analyzed. It is shown that the option of selective scan is very advantageous and can lead to prominent performance improvement. Sensitivity analysis on various parameters is conducted.

*Index Terms* — Data mining, traversal patterns, distributed information system, World Wide Web, performance analysis.

*M.-S. Chen is in part supported by National Science Council, Project No. NSC 86-2621-E-002-023-T, Taiwan, ROC.

†J. S. Park is supported by the Grants for Professors of Sungshin Women's University in 1997, Korea.

# 1 Introduction

Due to the increasing use of computing for various applications, the importance of database mining is growing at a rapid pace recently. Progress in bar-code technology has made it possible for retail organizations to collect and store massive amounts of sales data. Catalog companies can also collect sales data from the orders they received. It is noted that analysis of past transaction data can provide very valuable information on customer buying behavior, and thus improve the quality of business decisions (such as what to put on sale, which merchandises to be placed together on shelves, how to customize marketing programs, to name a few). It is essential to collect a sufficient amount of sales data before any meaningful conclusion can be drawn therefrom. As a result, the amount of these processed data tends to be huge. It is hence important to devise efficient algorithms to conduct mining on these data.

Note that various data mining capabilities have been explored in the literature. One of the most important data mining problems is mining association rules [3, 4, 13, 15]. For example, given a database of sales transactions, it is desirable to discover all associations among items such that the presence of some items in a transaction will imply the presence of other items in the same transaction. Also, mining classification is an approach of trying to develop rules to group data tuples together based on certain common features. This has been explored both in the AI domain [16, 17] and in the context of databases [2, 6, 12]. Mining in spatial databases was conducted in [14]. Another source of data mining is on ordered data, such as stock market and point of sales data. Interesting aspects to explore from these ordered data include searching for similar sequences [1, 19], e.g., stocks with similar movement in stock prices, and sequential patterns [5], e.g., grocery items bought over a set of visits in sequence. It is noted that data mining is a very application-dependent issue and different applications explored will require different mining techniques to cope with. Proper problem identification and formulation is therefore a very important part of the whole knowledge discovery process.

In this paper, we shall explore a new data mining capability which involves mining access patterns in a distributed information providing environment where documents or objects are linked together to facilitate interactive access. Examples for such information providing environments include World Wide Web (WWW) [11] and on-line services, such as Prodigy, CompuServe and America Online, where users, when seeking for information of interest, travel from one object to another via the corresponding facilities (i.e., hyperlinks) provided. Clearly, understanding user access patterns in such environments will not only help improve the system design (e.g., provide efficient access between highly correlated objects, better authoring design for pages, etc.) but also be able to lead to better marketing decisions (e.g., putting advertisements in proper places, better customer/user classification and behavior analysis, etc.). Capturing user access patterns in such environments is referred to as *mining traversal patterns* in this paper. Note that although some efforts have elaborated upon analyzing the user behavior [8, 9, 10], there is little result reported on dealing with the algorithmic aspects to improve the execution of traversal pattern mining. This can be in part explained by the reason that these information providing services, though with great potential, are mostly in their infancy and their customer analysis may still remain in a coarser level such as user occupation/age study. In addition, it is important to note that, as pointed out in [8], since users are traveling along the information providing services to search for the desired information, some objects are visited because of their locations rather than their content, showing the very difference between the traversal pattern problem and others which are mainly based on customer transactions. This unique feature of the traversal pattern problem unavoidably increases the difficulty of extracting meaningful information from a sequence of traversal data. However, as these information providing services are becoming increasingly popular nowadays, there is a growing demand for capturing user behavior and improving the quality of such services. As a result, the problem of mining traversal patterns has become too important not to address immediately.

Consequently, we shall explore in this paper the problem of mining traversal patterns. Our solution procedure consists of two steps. First, we derive an algorithm, called algorithm $MF$ (standing for maximal forward references), to convert the original sequence of log data into a set of traversal subsequences. As defined in Section 2, each traversal subsequence represents a maximal forward reference from the starting point of a user access. As will be explained later, this step of converting the original log sequence into a set of maximal forward references will filter out the effect of backward references which are mainly made for ease of traveling, and enable us to concentrate on mining meaningful user access sequences. Second, we derive algorithms to determine the frequent traversal patterns, termed *large reference sequences*, from the maximal forward references obtained above, where a large reference sequence is a reference sequence that appeared in a sufficient number of times in the database. Note that the problem of finding large reference sequences is similar to that of finding large itemsets for association rules [3] where a large itemset is a set of items appearing in a sufficient number of transactions. However, they are different from each other in that a reference sequence in mining traversal patterns has to be consecutive references in a maximal forward reference whereas a large itemset in mining association rules is just a combination of items in a transaction. As a consequence, although several schemes for mining association rules have been reported in the literature [3, 4, 15], the very difference between these two problems calls for the design of new algorithms for determining large reference sequences.

Explicitly, we devise two algorithms for determining large reference sequences. The first one, referred to as *full-scan* (FS) algorithm, essentially utilizes some techniques on hashing and pruning while solving the discrepancy between traversal patterns and association rules mentioned above. Although trimming the transaction database as it proceeds to later passes, algorithm FS is required to scan the transaction database in each pass. In contrast, by properly utilizing the candidate reference sequences, the second algorithm devised, referred to as *selective-scan* (SS) algorithm, is able to avoid database scans in some passes so as to reduce the disk I/O cost involved. Specifically, algorithm SS has the option of using a candidate reference set to generate subsequent candidate reference sets, and delaying the determination of large reference sets to a later pass when the database is scanned. Since SS does not scan the database to obtain a large reference set in each pass, some database scans are saved. Experimental studies are conducted by using a synthetic workload that is generated based on referencing some logged traces, and performance of these two methods, FS and SS, is comparatively analyzed. It is shown that the option of selective scan is very advantageous and algorithm SS thereby outperforms algorithm FS in general. Sensitivity analysis on various parameters is also conducted.

This paper is organized as follows. Problem formulation is given in Section 2. Algorithm MF to identify maximal forward references is described in Section 3.1, and two algorithms, FS and SS, for determining large reference sequences are given in Section 3.2. Performance results are presented in Section 4. Section 5 contains the summary.

## 2 Problem Formulation

As pointed out earlier, in an information providing environment where objects are linked together, users are apt to traverse objects back and forth in accordance with the links and icons provided. As a result, some node might be revisited because of its location, rather than its content. For example, in a WWW environment, to reach a sibling node a user is usually inclined to use "backward" icon and then a forward selection, instead of opening a new URL. Consequently, to extract meaningful user access patterns from the original log database, we naturally want to take into consideration the effect of such backward traversals and discover the real access patterns of interest. In view of this, we assume in this paper that a backward reference is mainly made for ease of traveling but not for browsing, and concentrate on the discovery of forward reference patterns. Specifically, a backward reference means revisiting a previously visited object by the same user access. When backward references occur, a forward reference path terminates. This resulting forward reference path is termed a *maximal forward reference*. After a maximal forward reference is obtained, we back track to the starting point of the forward referencing and resume another forward reference path. In addition, the occurrence of a null source node also indicates the termination of an ongoing forward reference path and the beginning of a new one.

While deferring the formal description of the algorithm to determine maximal forward references (i.e., algorithm MF) to Section 3.1, we give an illustrative example for maximal forward references below. Suppose the traversal log contains the following traversal path for a user: $\{A, B, C, D, C, B, E, G, H, G, W, A, O, U, O, V\}$, as shown in Figure 1. Then, it can be verified by algorithm MF that the set of maximal forward references for this user is $\{ABCD, ABEGH, ABEGW, AOU, AOV\}$. After maximal forward references for all users are obtained, we then map the problem of finding frequent traversal patterns into the one of finding frequent occurring consecutive subsequences among all maximal forward references. A *large reference sequence* is a reference sequence that appeared in a sufficient number of times. In a set of maximal forward references, the number of times a reference sequence has to appear in order to be qualified as a large reference sequence is called the minimal *support*. A large $k$-reference is a large reference sequence with $k$ elements. We denote the set of large $k$-references as $L_k$ and its candidate set as $C_k$, where $C_k$, as obtained from $L_{k-1}$ [4], contains those $k$-references that may appear in $L_k$. Explicitly, $C_k$ is a superset of $L_k$.

It is worth mentioning that after large reference sequences are determined, *maximal reference sequences* can then be obtained in a straightforward manner. A maximal reference sequence is a large reference sequence that is not contained in any other maximal reference sequence. For example, suppose that $\{AB, BE, AD, CG, GH, BG\}$ is the set of large 2-references (i.e., $L_2$) and $\{ABE, CGH\}$ is the set of large 3-references (i.e., $L_3$). Then, the resulting maximal reference sequences are $AD, BG, ABE$, and $CGH$. A maximal reference sequence corresponds to a "hot" access pattern in an information providing service. In all, the entire procedure for mining traversal patterns can be summarized as follows.

Procedure for mining traversal patterns:

**Step 1:** Determine maximal forward references from the original log data.

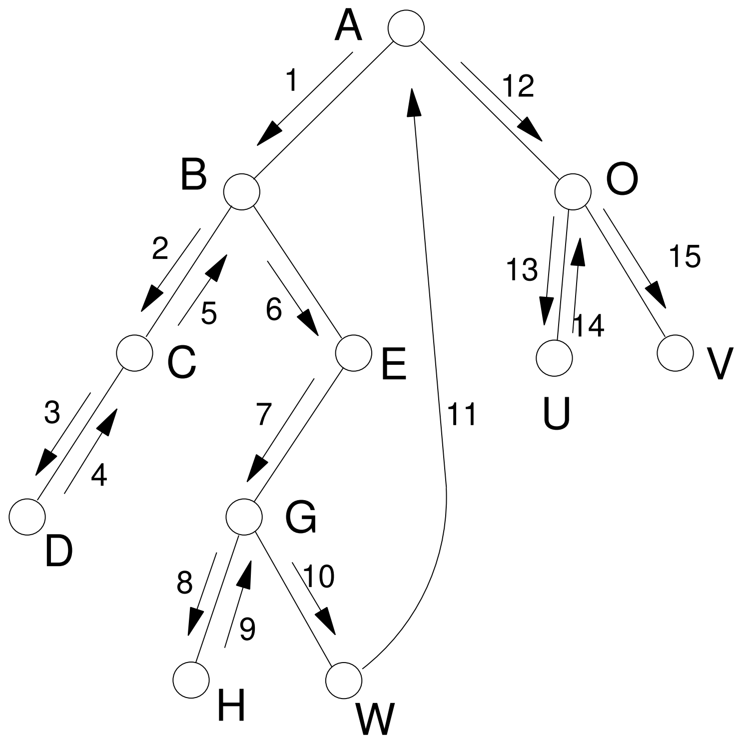


Figure 1: An illustrative example for traversal patterns.

**Step 2:** Determine large reference sequences (i.e., $L_k$, $k \geq 1$) from the set of maximal forward references.

**Step 3:** Determine maximal reference sequences from large reference sequences.

Since the extraction of maximal reference sequences from large reference sequences (i.e., Step 3) is straightforward, we shall henceforth focus on Steps 1 and 2, and devise algorithms for the efficient determination of large reference sequences.

# 3 Algorithm for Traversal Pattern

We shall describe in Section 3.1 algorithm MF which converts the original traversal sequence into a set of maximal forward references. Then, by mapping the problem of finding frequent traversal patterns into the one of finding frequent consecutive subsequences, we develop two algorithms, called full-scan (FS) and selective-scan (SS), for mining traversal patterns.

## 3.1 Identifying Maximal Forward References

In general, a traversal log database contains, for each link traversed, a pair of (source, destination). This part of log database is called *referer log* [7]. For the beginning of a new path, which is not linked to the previous traversal, the source node is null. Given a traversal sequence $\{(s_1, d_1), (s_2, d_2), ..., (s_n, d_n)\}$ of a user, we shall map it into multiple subsequences, each of which represents a maximal forward reference. The algorithm for finding all maximal forward references is given as follows. First, the traversal log database is sorted by user id's, resulting in a traversal path, $\{(s_1, d_1), (s_2, d_2), ..., (s_n, d_n)\}$, for each user, where pairs of $(s_i, d_i)$ are ordered by time. Algorithm $MF$ is then applied to each user path to determine all of its maximal forward references. Let $D_F$ denote the database to store all the resulting maximal forward references obtained.

**Algorithm $MF$: An algorithm to find maximal forward references.**

**Step 1:** Set $i = 1$ and string $Y$ to null for initialization, where string $Y$ is used to store the current forward reference path. Also, set the flag $F = 1$ to indicate a forward traversal.

**Step 2:** Let $A = s_i$ and $B = d_i$.
If $A$ is equal to null then
/* this is the beginning of a new traversal */
begin
Write out the current string $Y$ (if not null) to the database $D_F$;
Set string $Y = B$;
Go to Step 5.
end

**Step 3:** If $B$ is equal to some reference (say the $j$-th reference) in string $Y$ then
/* this is a cross-referencing back to a previous reference */
begin
If $F$ is equal to 1 then write out string $Y$ to database $D_F$;
Discard all the references after the $j$-th one in string $Y$;
$F = 0$;
Go to Step 5.
end

**Step 4:** Otherwise, append $B$ to the end of string $Y$.
/* we are continuing a forward traversal */
If $F$ is equal to 0, set $F = 1$.

**Step 5:** Set $i = i + 1$. If the sequence is not completed scanned then go to Step 2.

Consider the traversal scenario in Figure 1 for example. It can be verified that the first backward reference is encountered in the 4-th move (i.e., from $D$ to $C$). At that point, the maximal forward reference $ABCD$ is written to $D_F$ (by Step 3). In the next move (i.e., from $C$ to $B$), although the first conditional statement in Step 3 is again true, nothing is written to $D_F$ since the flag $F = 0$, meaning that it is in a reverse traversal. The subsequent forward references will put $ABEGH$ into the string $Y$, which is then written to $D_F$ when a reverse reference (from $H$ to $G$)

Table 1: An example execution by algorithm MF.

| move | string $Y$ | output to $D_F$ |
|---|---|---|
| 1 | $AB$ | – |
| 2 | $ABC$ | – |
| 3 | $ABCD$ | – |
| 4 | $ABC$ | $ABCD$ |
| 5 | $AB$ | – |
| 6 | $ABE$ | – |
| 7 | $ABEG$ | – |
| 8 | $ABEGH$ | – |
| 9 | $ABEG$ | $ABEGH$ |
| 10 | $ABEGW$ | – |
| 11 | $A$ | $ABEGW$ |
| 12 | $AO$ | – |
| 13 | $AOU$ | – |
| 14 | $AO$ | $AOU$ |
| 15 | $AOV$ | $AOV$ (end) |

is encountered. The execution scenario by algorithm MF for the input in Figure 1 is given in Table 1.

It is noted that in some cases, the traversal log record obtained only contains the destination references instead of a pair of references. For example, for WWW browsing, the request message may only contain the destination URL. The traversal sequence will then have the form $\{d_1, d_2, ..., d_n\}$ for each user. Even with such an input, we can still convert it into a set of maximal forward references. The only difference is that in this case we cannot identify the breakpoint where the user picks a new URL to begin a new traversal path, meaning that two consecutive maximal forward references. For example, $ABEH$ and $WXYZ$, may be treated as one path, i.e., $ABEHWXYZ$. Certainly, this constraint, i.e., without the id's of source nodes, could increase the computational complexity because the paths considered become longer. However, this constraint should have little effect on identifying frequent reference subsequences. Since there is no logical link between $H$ and $W$, a subsequence containing $HW$ is unlikely to occur frequently. Hence, a reference containing the pattern HW will unlikely emerge as a large reference later. Therefore, algorithm MF can in fact be employed for the cases when the id's of source nodes are not available.

### 3.2 Determining Large Reference Sequences

Once the database containing all maximal forward references for all users, $D_F$, is constructed, we can derive the frequent traversal patterns by identifying the frequent occurring reference sequences in $D_F$ . A sequence $s_1, ...., s_n$ is said to contain $r_1, ...., r_k$ as a consecutive subsequence if there exists an $i$ such that $s_{i+j} = r_j$, for $1 \leq j \leq k$. For example, $BAHPM$ is said to contain $AHP$. A sequence of $k$ references, $r_1, ...., r_k$, is called a *large k-reference sequence*, if there are a sufficient number of users with maximal forward references in $D_F$ containing $r_1, ...., r_k$ as a consecutive subsequence.

As pointed out before, the problem of finding large reference sequences is different from that of finding large itemsets for association rules and thus calls for the design of new algorithms. Consequently, we shall derive in this paper two algorithms for mining traversal patterns. The first one, called full-scan (FS) algorithm, essentially utilizes the concept of DHP [15] (i.e., hashing and pruning) while solving the discrepancy between traversal patterns and association rules. DHP has two major features in determining association rules: one is efficient generation for large itemsets and the other is effective reduction on transaction database size after each scan. Although trimming the transaction database as it proceeds to later passes, FS is required to scan the transaction database in each pass. In contrast, by properly utilizing the candidate reference sequences, the second algorithm, referred to as selective-scan (SS) algorithm, is improved with the option of determining large reference sequences in batch so as to reduce the number of database scans required.

#### 3.2.1 Algorithm on Full Scan (FS)

Algorithm FS utilizes key ideas of the DHP algorithm. The details of DHP can be found in [15]. An example scenario for determining large itemsets and candidate itemsets is given in the Appendix[1]. As shown in [15], by utilizing a hash technique, DHP is very efficient for the generation of candidate itemsets, in particular for the large 2-itemsets, thus greatly improving the performance bottleneck of the whole process. In addition, DHP employs effective pruning techniques to progressively reduce the transaction database size.

Recall that $L_k$ represents the set of all large $k$-references and $C_k$ is a set of candidate $k$-references. $C_k$ is in general a superset of $L_k$. By scanning through $D_F$, FS gets $L_1$ and makes a hash table (i.e., $H_2$) to count the number of occurrences of each 2-reference. Similarly to DHP, starting with $k = 2$, FS generates $C_k$ based on the hash table count obtained in the previous pass, determines the set of

[1] In this example, the technique of hashing, which is employed by DHP to reduce the number of candidate itemsets, is not shown.

large $k$-references, reduces the size of database for the next pass, and makes a hash table to determine the candidate $(k+1)$-references. Note that as in mining association rules, a set of candidate references, $C_k$, can be generated from joining $L_{k-1}$ with itself, denoted by $L_{k-1} * L_{k-1}$.[2] However, due to the difference between traversal patterns and association rules, we modify this approach as follows. For any two distinct reference sequences in $L_{k-1}$, say $r_1, ...., r_{k-1}$ and $s_1, ...., s_{k-1}$, we join them together to form a $k$-reference sequence only if either $r_1, ...., r_{k-1}$ contains $s_1, ...., s_{k-2}$ or $s_1, ...., s_{k-1}$ contains $r_1, ...., r_{k-2}$ (i.e., after dropping the first element in one sequence and the last element in the other sequence, the remaining two $(k-2)$-references are identical). We note that when $k$ is small (especially for the case of $k = 2$), deriving $C_k$ by joining $L_{k-1}$ with itself will result in a very large number of candidate references and the hashing technique is thus very helpful for such a case. As $k$ increases, the size of $L_{k-1} * L_{k-1}$ can decrease significantly. Same as in [15], we found that it is generally beneficial for FS to generate $C_k$ directly from $L_{k-1} * L_{k-1}$ (i.e., without using hashing) once $k \geq 3$.

To count the occurrences of each $k$-reference in $C_k$ to determine $L_k$, we need to scan through a trimmed version of database $D_F$. From the set of maximal forward references, we determine, among $k$-references in $C_k$, large $k$-references. After the scan of the entire database, those $k$-references in $C_k$ with count exceeding the threshold become $L_k$. If $L_k$ is non-empty, the iteration continues for the next pass, i.e., pass $k+1$. Same as in DHP, every time when the database is scanned, the database is trimmed by FS to improve the efficiency of future scans.

### 3.2.2 Algorithm on Selective Scan (SS)

Algorithm SS is similar to algorithm FS in that it also employs hashing and pruning techniques to reduce both CPU and I/O costs, but is different from the latter in that algorithm SS, by properly utilizing the information in candidate references in prior passes, is able to avoid database scans in some passes, thus further reducing the disk I/O cost. The method for SS to avoid some database scans and reduce disk I/O cost is described below. Recall that algorithm FS generates a small number of candidate 2-references by using a hashing technique. In fact, this small $C_2$ can be used to generate the candidate 3-references. Clearly, a $C_3'$ generated from $C_2 * C_2$, instead of from $L_2 * L_2$, will have a size greater than $|C_3|$ where $C_3$ is generated from $L_2 * L_2$. However, if $|C_3'|$ is not much larger than $|C_3|$, and both $C_2$ and $C_3'$ can be stored in the main memory, we can find $L_2$ and $L_3$ together when the next scan of the database is performed, thereby saving one round of database

[2]This approach of generating $C_k$ directly from $L_{k-1}$ is proposed by algorithm Apriori in [4] in generating candidate itemsets for association rules.

scan. It can be seen that using this concept, one can determine all $L_k$'s by as few as two scans of the database (i.e., one initial scan to determine $L_1$ and a final scan to determine all other large reference sequences), assuming that $C'_k$ for $k \geq 3$ is generated from $C'_{k-1}$ and all $C'_k$s for $k > 2$ can be kept in the memory.

Note that when the minimum support is relatively small or potentially large references are long, $C_k$ and $L_k$ could become large. With $C'_{k+1}$ being generated from $C'_k * C'_k$, if $|C'_{k+1}| > |C'_k|$ for $k \geq 2$, then it may cost too much CPU time to generate all subsequent $C'_j$, $j > k+1$, from candidate sets of large references since the size of $C_j$ may become huge quickly, thus compromising all the benefit from saving disk I/O cost. For the illustrative example in the Appendix, if $C_3$ was determined from $C_2 * C_2$, instead of from $L_2 * L_2$, then $C_3$ would be {{ABC}, {ABE}, {ACE}, {BCE}}. This fact suggests that a timely database scan to determine large reference sequences will in fact pay off. After a database scan, one can obtain the large reference sequences which are not determined thus far (say, up to $L_m$) and then construct the set of candidate $(m+1)$-references, $C_{m+1}$, based on $L_m$ from that point. According to our experiments, we found that if $|C'_{k+1}| > |C'_k|$ for some $k \geq 2$, it is usually beneficial to have a database scan to obtain $L_{k+1}$ before the set of candidate references becomes too big. (Same as in FS, each time the database is scanned, the database is trimmed by SS to improve the efficiency of future scans.) We then derive $C'_{k+2}$ from $L_{k+1}$. (We note that $C'_{k+2}$ is in fact equal to $C_{k+2}$ here.) After that, we again use $C'_j$ to derive $C'_{j+1}$ for $j \geq k+2$. The process continues until the set of candidate $(j+1)$-references becomes empty.

Illustrative examples for FS and SS are given in Table 2 where the number of reference paths $|D| = 200,000$ and the minimum support $s = 0.75\%$. Extensive experiments are conducted in Section 4. In this example run, FS performs a database scan in each pass to determine the corresponding large reference sequences, resulting in six database scans. On the other hand, SS scans the database only three times (skipping database scans in passes 2, 4 and 5), and is able to obtain the same result. The CPU and disk I/O times for FS are 19.48 seconds and 30.8 seconds, respectively, whereas those for SS are 18.75 seconds and 17.8 seconds, respectively. Considering both CPU and I/O times, the execution time ratio for SS to FS is 0.73, showing a prominent advantage of SS.

# 4 Performance Results

To assess the performance of FS and SS, we conducted several experiments to determine large reference sequences by using an RS/6000 workstation with model 560. The methods used to

Table 2: Results from an example run by FS and SS.

| $k$ | 1 | 2 | 3 | 4 | 5 | 6 | time (sec) |
|---|---|---|---|---|---|---|---|
| Algorithm FS | | | | | | | |
| $C_k$ | | 121 | 84 | 58 | 22 | 3 | |
| $L_k$ | 94 | 91 | 84 | 58 | 21 | 3 | 19.48 |
| $D_k$ | 12.8MB | 12.8MB | 12.2MB | 5.3MB | 1.9MB | 0.26MB | 30.80 |
| Algorithm SS | | | | | | | |
| $C_k$ | | 121 | 144 | 58 | 22 | 3 | |
| $L_k$ | 94 | 91 | 84 | 58 | 21 | 3 | 18.75 |
| $D_k$ | 12.8MB | – | 12.8MB | – | – | 5.3MB | 17.80 |

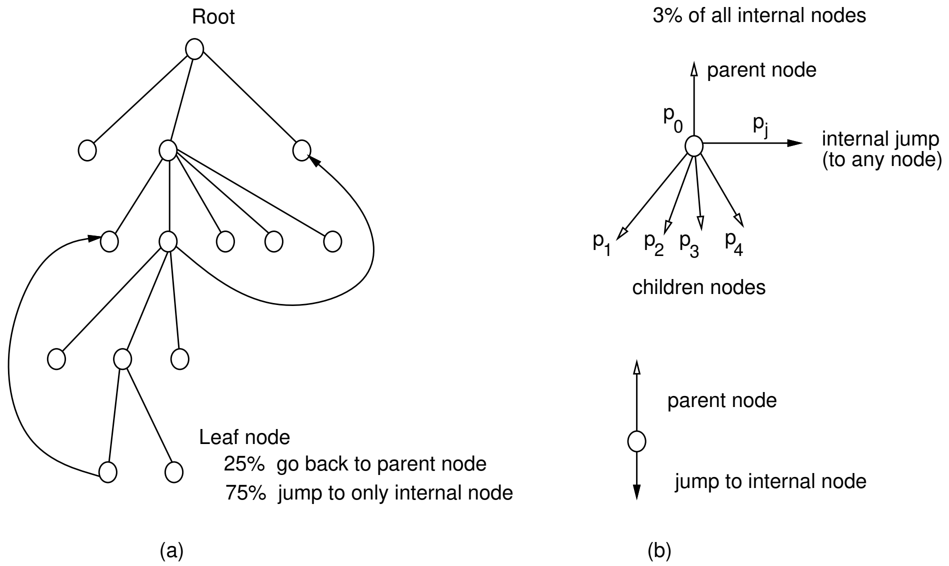


Figure 2: A traversal tree to simulate WWW.

generate synthetic data are described in Section 4.1. Performance comparison of these two methods is given in Section 4.2. Sensitivity analysis is conducted in Section 4.3.

### 4.1 Generation of Synthetic Traversal Paths

In our experiment, the browsing scenario in a World Wide Web (WWW) environment is simulated. To generate a synthetic workload and determine the values of parameters, we referenced some logged traces which were collected from a gateway in our working location [18]. First, a traversal tree is constructed to mimic WWW structure whose starting position is a root node of the tree. The traversal tree consists of internal nodes and leaf nodes. Figure 2a shows an example of the traversal tree. The number of child nodes at each internal node, referred to as *fanout*, is determined from a uniform distribution within a given range. The height of a subtree whose subroot is a child node of the root node is determined from a Poisson distribution with mean $\mu_h$. Then, the height of a subtree whose subroot is a child of an internal node $N_i$ is determined from a Poisson distribution with mean equal to a fraction of the maximum height of the internal node $N_i$. As such, the height of a tree is controlled by the value of $\mu_h$.

A traversal path consists of nodes accessed by a user. The size of each traversal path is picked from a Poisson distribution with mean equal to $|P|$. With the first node being the root node, a traversal path is generated probabilistically within the traversal tree as follows. For each internal node, we determine which is the next hop according to some predetermined probabilities. Essentially, each edge connecting to an internal node is assigned with a weight. This weight corresponds to the probability that each edge will be next accessed by the user. As shown in Figure 2b, the weight to its parent node is assigned with $p_0$, which is generally $\frac{1}{n+1}$ where $n$ is the number of child nodes. This probability of traveling to each child node, $p_i$, is determined from an exponential distribution with unit mean, and is so normalized that the sum of the weights for all child nodes is equal to $1 - p_0$. If this internal node has an internal jump and the weight for this jump is $p_j$, then $p_0$ is changed to $p_0(1 - p_j)$ and the corresponding probability for each child node is changed to $p_i(1-p_j)$ such that the sum of all the probabilities associated with this node remains one. When the path arrives at a leaf node, the next move would be either to its parent node in backward (with a probability 0.25) or to any internal node (with an aggregate probability 0.75). Some internal nodes in the tree have internal jumps which can go to any other nodes. The number of internal nodes with internal jumps is denoted by $N_J$, which is set to 3% of all the internal nodes in general cases. The sensitivity of varying $N_J$ will also be analyzed. Those nodes with internal jumps are decided randomly among all the internal nodes. Table 3 summarizes the meaning of various parameters used in our simulations.

### 4.2 Performance Comparison between FS and SS

Figure 3 represents execution times of two methods, FS and SS, when $|D|$ = 200,000, $N_J$ = 3%, and $p_j$ = 0.1. H$x$P$y$ means that $x$ is the height of a tree and $y$ is the average size of the reference paths. D200K means that the number of reference paths is 200,000. A tree for H10 was obtained when the height of a tree is 10 and the fanout at each internal node is between 4 and 7. The root node consists of 7 child nodes. The number of internal nodes is 16,200 and the number of leaf nodes is 73,006. The number of internal nodes with internal jumps is thus 16200 $\times N_J$ =486. Note that the

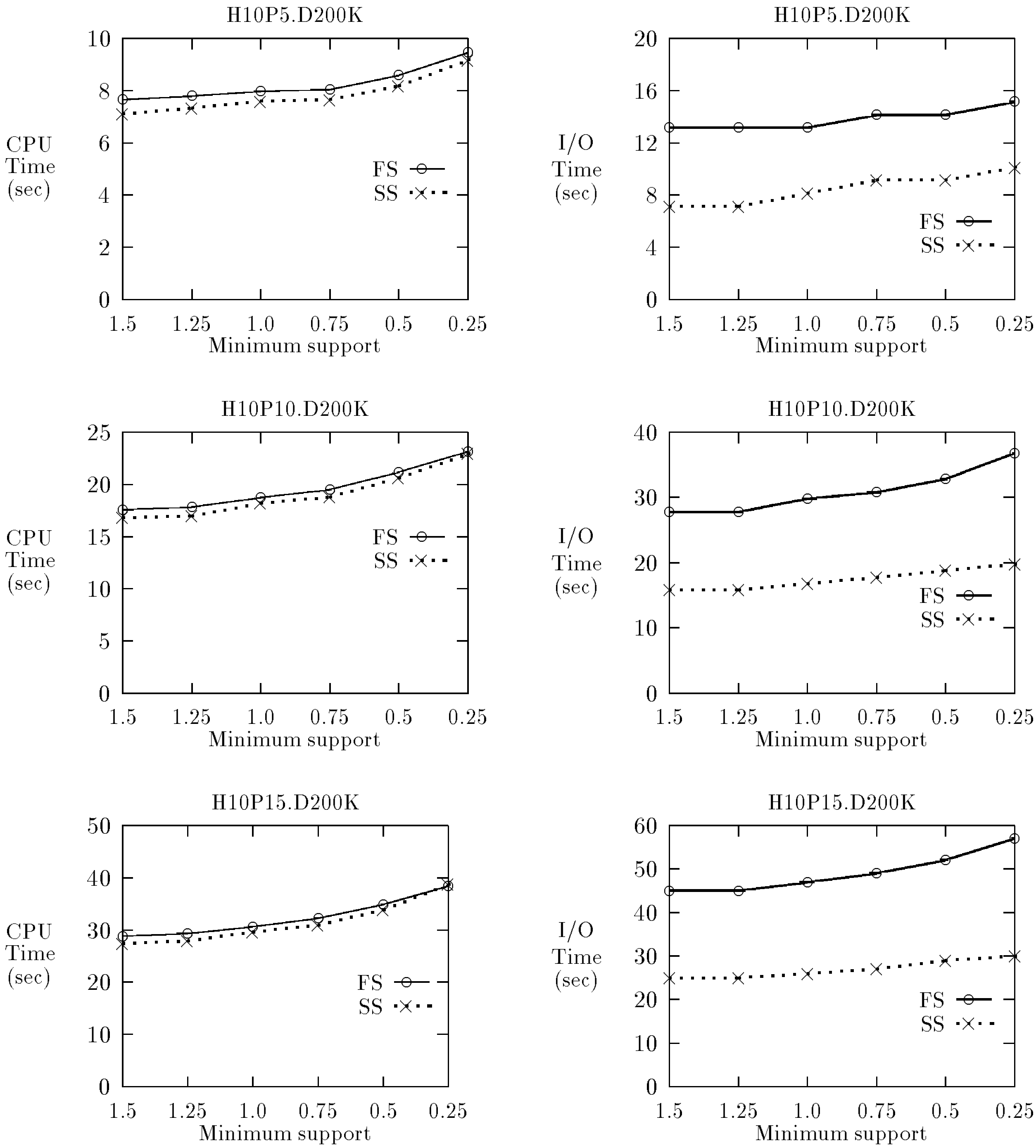


Figure 3: Execution Times for FS and SS.

Table 3: Meaning of various parameters.

| | |
|---|---|
| $H$ | The height of a traversal tree. |
| $F$ | The number of child nodes, fanout. |
| $N_J$ | The number of internal nodes with an internal jump . |
| $p_0$ | Backward weight in probability to its parent node. |
| $p_j$ | Jump weight in probability to its internal jump. |
| $\theta$ | A parameter of a Zipf-like distribution. |
| H$x$P$y$ | $x$ is the height of a tree and $y = \|P\|$. |
| $\|D\|$ | The number of reference paths (size of database). |
| $D_k$ | Set of forward references for $L_k$. |
| $C_k$ | Set of candidate $k$-reference sequences. |
| $L_k$ | Set of large $k$-reference sequences. |
| $\|P\|$ | Average size of the reference paths. |

total number of nodes increases as the height of a tree increases. To make the experiment tractable, we reduced the fanout to $2-5$ for the tree of H20 with the height of 20. This tree contained 616,595 internal nodes and 1,541,693 leaves. In Figure 3, the left graph of each H$x$P$y$.D200K represents the CPU time to find all the large reference sequences, and the right graph shows the I/O time to find them where the disk I/O time is set to 2 MB/sec and 1 MB buffer is used in main memory. It can be seen from Figure 3 that algorithm SS in general outperforms FS, and their performance difference becomes prominent when the I/O cost is taken into account.

To provide more insights into their performance, in addition to Table 2 in Section 3, we have Table 4 which shows the results by these two methods when $|D| = 200,000$ and $s = 0.75\%$. In Table 4, FS scans the database eight times to find all the large reference sequences, whereas SS only involves three database scans. Note that after initial scans, disk I/O involved by FS and SS will include both disk read and disk write (i.e., writing the trimmed version of the database back to the disk). The I/O time for these two methods is shown in Figure 4. Considering both CPU and I/O times, the total execution time of FS is 143.94 seconds, and that of SS is 100.89 seconds. Note that the execution time ratio for FS to SS is 0.70 in this case, which is slightly better than the one associated with Table 2.

Figure 5 shows scale-up experiments, where both the CPU and I/O times of each method increase linearly as the database size increases. For this experiment, the traversal tree has 10 levels, the fanout of internal nodes is between 4 and 7, and the minimum support is set to 0.75%. It can be seen that SS consistently outperforms FS as the database size increases.

Table 4: Number of large reference sequences and execution times for H20P20.

| $k$ | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | time (sec) |
|---|---|---|---|---|---|---|---|---|---|
| Algorithm FS | | | | | | | | | |
| $C_k$ | | 206 | 146 | 106 | 75 | 37 | 15 | 4 | |
| $L_k$ | 141 | 139 | 124 | 103 | 70 | 36 | 15 | 4 | 58.94 |
| $D_k$ | 29MB | 29MB | 27.3MB | 13.8MB | 10.1MB | 5.3MB | 1.9MB | 0.6MB | 85.00 |
| Algorithm SS | | | | | | | | | |
| $C_k$ | | 206 | 370 | 106 | 75 | 37 | 15 | 4 | |
| $L_k$ | 141 | 139 | 124 | 106 | 70 | 36 | 15 | 4 | 57.89 |
| $D_k$ | 29MB | – | 29MB | – | – | – | – | 14.1MB | 43.00 |

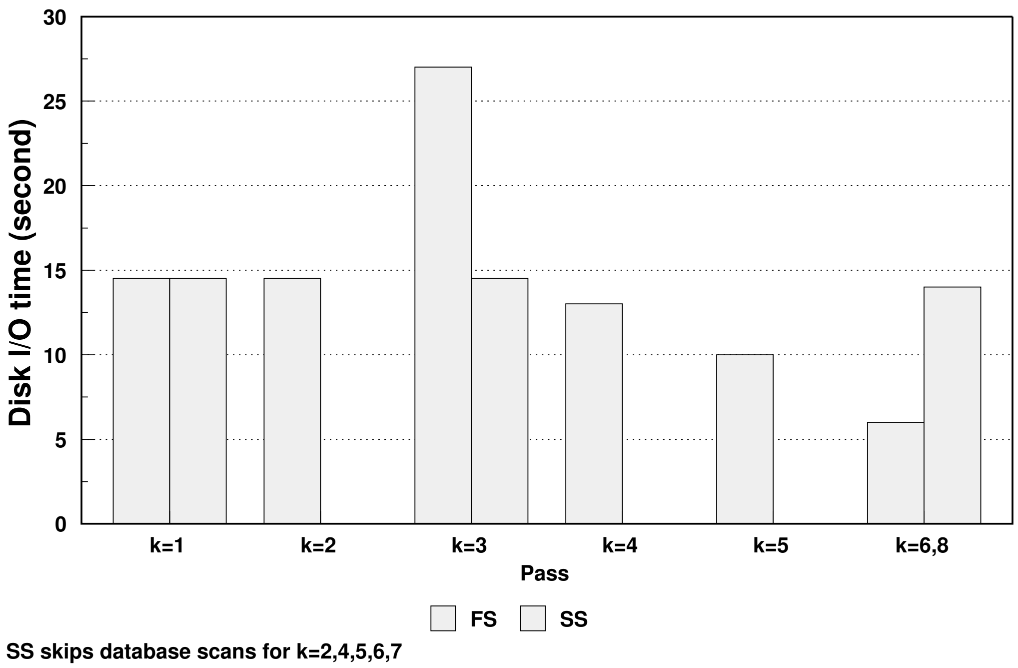


Figure 4: I/O cost for FS and SS in each pass.

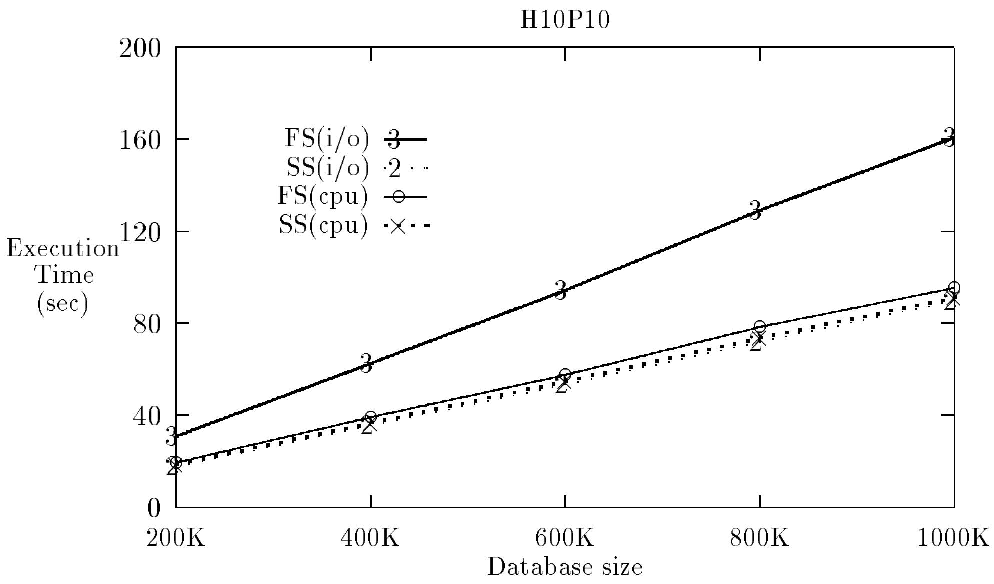


Figure 5: Execution time of FS and SS when database size increases.

### 4.3 Sensitivity Analysis

Since algorithm SS in general outperforms FS, without loss of generality, we shall conduct in this section the sensitivity analysis on various parameters for algorithm SS. Performance evaluation was carried out under the condition that the database size is 200,000, the average size of traversal paths is 10, i.e., $|P| = 10$, and the minimum support is 0.75%.

Figure 6 shows the number of large reference sequences when the probability to backward at an internal node, $p_0$, varies from 0.1 to 0.5. As the probability increases, the number of large reference sequences decreases because the possibility of having forward traveling becomes smaller. Figure 7 shows the number of large reference sequences when the number of child nodes of internal nodes, i.e., fanout $F$, varies. The three corresponding traversal trees all have the same height 8. The tree with $2-4$ fanout consists of 483 internal nodes and 1,267 leaf nodes. The tree for the second bar consists of 11,377 internal nodes and 62,674 leaf nodes, and the one for the third bar consists of 74,632 internal nodes and 634,538 leaf nodes. The results show that the number of large reference sequences decreases as the degree of fanout increases, because with a larger fanout the traversal paths are more likely to be dispersed to several branches, thus resulting in fewer large reference sequences. Clearly, when the large reference sequences decreases, the execution time to find them

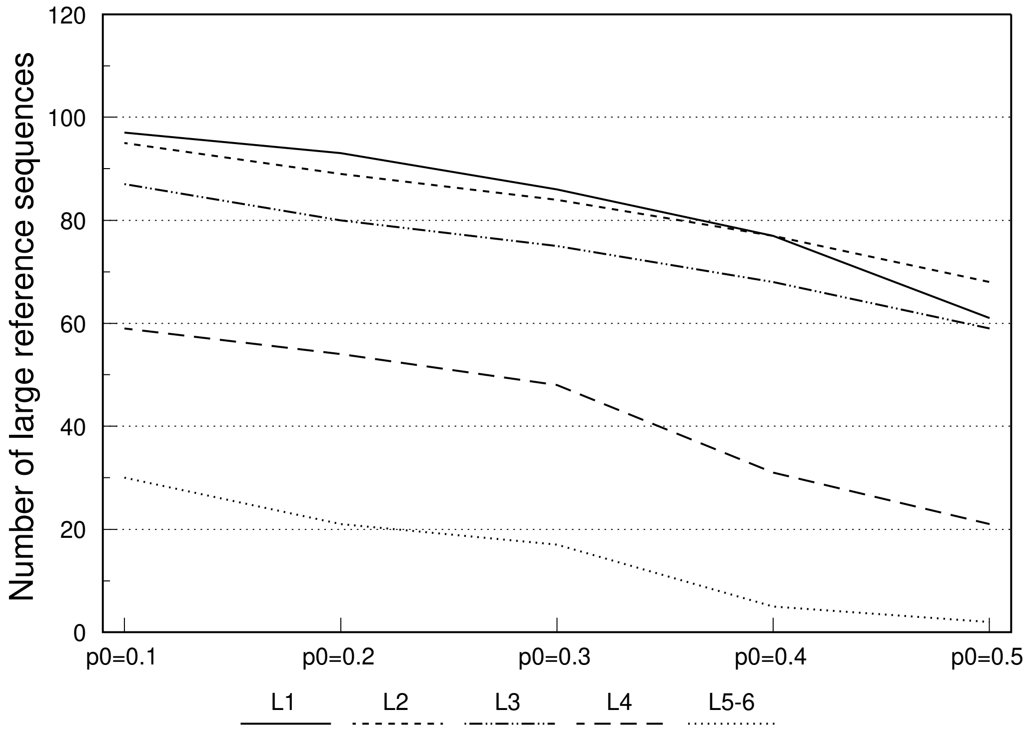


Figure 6: Number of large reference sequences when backward weight $p_0$ is varied.

also decreases.

Figure 8 gives the number of large reference sequences when the probability of traveling to each child node from an internal node is determined from a Zipf-like distribution. Different values of parameter $\theta$ for the Zipf-like distribution are considered. The Zipf-like distribution of branching probabilities to child nodes is generated as follows. The probability $p_i$ that the $i$-th child node is accessed by a traversal path is $p_i = c/i^{(1-\theta)}$, where $c = 1/\sum_{i=1}^{n}(1/i^{(1-\theta)})$ is a normalization constant and $n$ is the number of child nodes at an internal node. After we get each $p_i$, it is then normalized so that $p_0 + \sum_{i=1}^{n} p_i + p_j = 1$ as in Section 4.1. Setting the parameter $\theta = 0$ corresponds to the pure Zipf distribution, which is highly skewed, whereas $\theta = 1$ corresponds to the uniform distribution. The results show that the number of large reference sequences increases when the corresponding probabilities are more skewed.

Table 5 shows the performance results of SS when the number of internal nodes with internal jumps, $N_J$, varies from 3% to 27 % of the total internal nodes. The number of large reference sequences decreases slightly as $N_J$ increases, meaning that it is less likely to have large reference

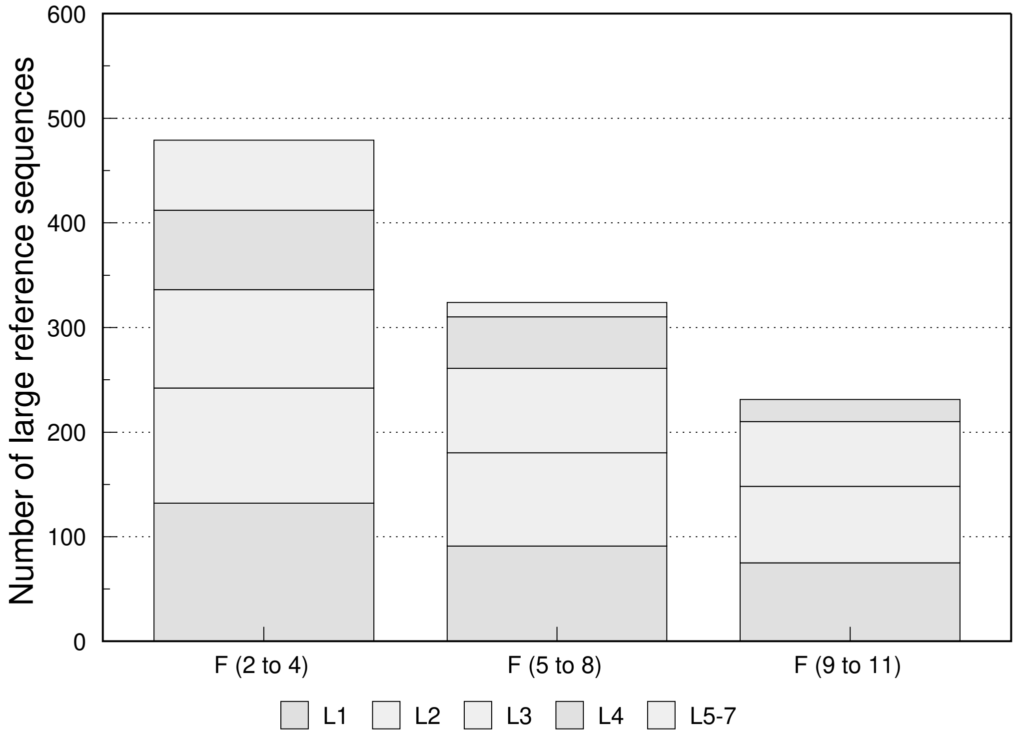


Figure 7: Number of large reference sequences when the fanout $F$ is varied.

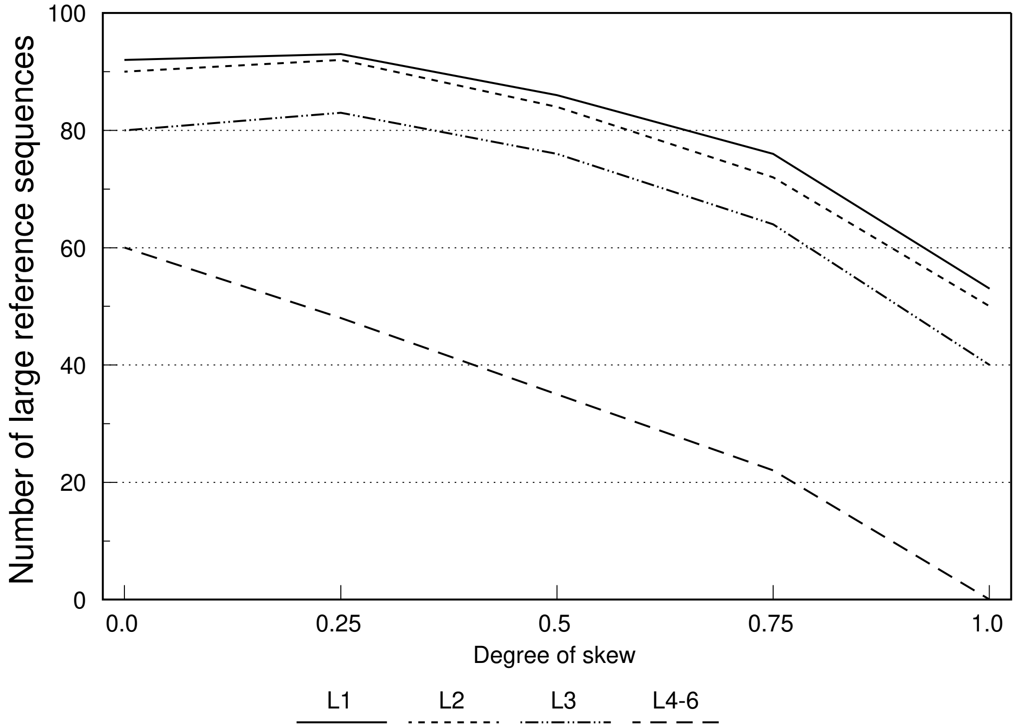


Figure 8: Number of large reference sequences when parameter $\theta$ of a Zipf-like distribution is varied.

Table 5: Number of large reference sequences when the percentage of internal jumps $N_J$ is varied.

| $N_J$[%] | $k$ | 1 | 2 | 3 | 4 | 5 | 6 | Time(sec) |
|---|---|---|---|---|---|---|---|---|
| 3 | $L_k$ | 94 | 91 | 84 | 58 | 21 | 3 | 18.76 |
| 9 | $L_k$ | 94 | 92 | 83 | 56 | 22 | 2 | 18.70 |
| 15 | $L_k$ | 93 | 90 | 83 | 55 | 22 | 3 | 18.88 |
| 21 | $L_k$ | 93 | 90 | 82 | 55 | 22 | 3 | 18.95 |
| 27 | $L_k$ | 90 | 87 | 80 | 53 | 20 | 2 | 18.69 |

Table 6: Number of large reference sequences when the height of a traversal tree $H$ is varied.

| $H$ | $k$ | 1 | 2 | 3 | 4 | 5 | 6 | 7 | Time(sec) |
|---|---|---|---|---|---|---|---|---|---|
| 3 | $L_k$ | 64 | 93 | 60 | 42 | 9 | | | 15.52 |
| 5 | $L_k$ | 157 | 136 | 103 | 76 | 41 | 11 | | 17.90 |
| 10 | $L_k$ | 116 | 111 | 100 | 80 | 48 | 20 | 4 | 19.68 |
| 15 | $L_k$ | 111 | 110 | 100 | 81 | 43 | 14 | 1 | 20.39 |
| 20 | $L_k$ | 98 | 97 | 92 | 73 | 46 | 19 | 4 | 21.01 |

sequences when we have more jumps in traversal paths. It is noted that performance of SS is less sensitive to this parameter than to others.

Table 6 shows results of SS when the height of a traversal tree varies. The fanout is between 2 and 5. As the height increases, the numbers of internal nodes and leaf nodes increase exponentially. The height of a traversal tree is increased from 3 to 20, As the height of a traversal tree increases, the number of candidate nodes for $L_1$ increases and the execution time to find $L_1$ thus increases. On the other hand, $|L_1|$ decreases as the height of the tree increases since the average visit to each node decreases. The number of large reference sequences slightly decreases, for $1 \leq k \leq 3$, when the height of the tree increases from 5 to 20.

## 5 Conclusion

In this paper, we have explored a new data mining capability which involves mining traversal patterns in an information providing environment where documents or objects are linked together to facilitate interactive access. Our solution procedure consisted of two steps. First, we derived algorithm MF to convert the original sequence of log data into a set of maximal forward references. By doing so, we filtered out the effect of some backward references and concentrated on mining meaningful user access sequences. Second, we developed algorithms to determine large reference sequences from the maximal forward references obtained. Two algorithms were devised for determining large reference sequences: one was based on some hashing and pruning techniques, and the other was further improved with the option of determining large reference sequences in batch so as to reduce the number of database scans required. Performance of these two methods has been comparatively analyzed. It is shown that the option of selective scan is very advantageous and algorithm SS thus in general outperformed algorithm FS. Sensitivity analysis on various parameters was conducted.

## Acknowledgements

M.-S. Chen is in part supported by National Science Council, Project No. NSC 86-2621-E-002-023-T, Taiwan, ROC. J. S. Park is supported by the Grants for Professors of Sungshin Women's University in 1997, Korea.

## References

[1] R. Agrawal, C. Faloutsos, and A. Swami. Efficient Similarity Search in Sequence Databases. *Proceedings of the 4th Intl. conf. on Foundations of Data Organization and Algorithms*, October, 1993.

[2] R. Agrawal, S. Ghosh, T. Imielinski, B. Iyer, and A. Swami. An Interval Classifier for Database Mining Applications. *Proceedings of the 18th International Conference on Very Large Data Bases*, pages 560–573, August 1992.

[3] R. Agrawal, T. Imielinski, and A. Swami. Mining Association Rules between Sets of Items in Large Databases. *Proceedings of ACM SIGMOD*, pages 207–216, May 1993.

[4] R. Agrawal and R. Srikant. Fast Algorithms for Mining Association Rules in Large Databases. *Proceedings of the 20th International Conference on Very Large Data Bases*, pages 478–499, September 1994.

[5] R. Agrawal and R. Srikant. Mining Sequential Patterns. *Proceedings of the 11th International Conference on Data Engineering*, pages 3–14, March 1995.

[6] T.M. Anwar, H.W. Beck, and S.B. Navathe. Knowledge Mining by Imprecise Querying: A Classification-Based Approach. *Proceedings of the 8th International Conference on Data Engineering*, pages 622–630, February 1992.

[7] T. Berners-Lee, R. Fiekding, and H. Frystyk. Hypertext Transfer Protocol-HTTP/1.0. *Internet Draft*, February 1996.

[8] M. Bieber and J. Wan. Backtracking in a Multiple-Window Hypertext Environment. *ACM European Conf. on Hypermedia Technology*, pages 158–166, 1994.

[9] E. Caramel, S. Crawford, and H. Chen. Browsing in Hypertext: A Cognitive Study. *IEEE Transactions on Systems, Man and Cybernetics*, 22(5):865–883, September 1992.

[10] L. D. Catledge and J. E. Pitkow. Characterizing browsing strategies in the world-wide web. *Proceedings of the 3rd WWW Conference*, April 1995.

[11] J. December and N. Randall. *The World Wide Web Unleashed.* SAMS Publishing, 1994.

[12] J. Han, Y. Cai, , and N. Cercone. Knowledge Discovery in Databases: An Attribute-Oriented Approach. *Proceedings of the 18th International Conference on Very Large Data Bases*, pages 547–559, August 1992.

[13] J. Han and Y. Fu. Discovery of Multiple-Level Association Rules from Large Databases. *Proceedings of the 21th International Conference on Very Large Data Bases*, pages 420–431, September 1995.

[14] R.T. Ng and J. Han. Efficient and Effective Clustering Methods for Spatial Data Mining. *Proceedings of the 18th International Conference on Very Large Data Bases*, pages 144–155, September 1994.

[15] J.-S. Park, M.-S. Chen, and P. S. Yu. An Effective Hash Based Algorithm for Mining Association Rules. *Proceedings of ACM SIGMOD*, pages 175–186, May, 1995.

[16] G. Piatetsky-Shapiro. Discovery, Analysis and Presentation of Strong Rules. *Knowledge Discovery in Databases*, pages 229–248, 1991.

[17] J.R. Quinlan. Induction of Decision Trees. *Machine Learning*, 1:81–106, 1986.

[18] N. R. Trio. Personal communication, May 1995.

[19] J. T.-L. Wang, G.-W. Chirn, T.G. Marr, B. Shapiro, D. Shasha, and K. Zhang. Combinatorial Pattern Discovery for Scientific Data: Some Preliminary Results. *Proceedings of ACM SIGMOD, Minneapolis, MN*, pages 115–125, May, 1994.

Database $D$

| TID | Items |
|---|---|
| 100 | A C D |
| 200 | B C E |
| 300 | A B C E |
| 400 | B E |

Scan $D$ $\longrightarrow$

$C_1$

| Itemset | Sup. |
|---|---|
| {A} | 2 |
| {B} | 3 |
| {C} | 3 |
| {D} | 1 |
| {E} | 3 |

$L_1$

| Itemset | Sup. |
|---|---|
| {A} | 2 |
| {B} | 3 |
| {C} | 3 |
| {E} | 3 |

$C_2$

| Itemset |
|---|
| {A B} |
| {A C} |
| {A E} |
| {B C} |
| {B E} |
| {C E} |

Scan $D$ $\longrightarrow$

$C_2$

| Itemset | Sup. |
|---|---|
| {A B} | 1 |
| {A C} | 2 |
| {A E} | 1 |
| {B C} | 2 |
| {B E} | 3 |
| {C E} | 2 |

$L_2$

| Itemset | Sup. |
|---|---|
| {A C} | 2 |
| {B C} | 2 |
| {B E} | 3 |
| {C E} | 2 |

$C_3$

| Itemset |
|---|
| {B C E} |

Scan $D$ $\longrightarrow$

$C_3$

| Itemset | Sup. |
|---|---|
| {B C E} | 2 |

$L_3$

| Itemset | Sup. |
|---|---|
| {B C E} | 2 |

Figure 9: An example for candidate set generation.

## **Appendix:** Generation of large itemsets and candidate itemsets

Given an example transaction database $D$, the large itemsets and candidate itemsets can be determined as follows. In essence, generated first are large 1-itemsets, which are then used to construct candidate itemsets in the next pass. With th minimal support equal to two, after each database scan, large itemsets are determined from candidate itemsets with the number of occurrences greater than or equal to two. A detailed algorithm can be found in [5].