Finding Generalized Path Patterns for Web Log Data Mining *

Alex Nanopoulos and Yannis Manolopoulos

Data Engineering Lab,
Department of Informatics, Aristotle University
54006 Thessaloniki, Greece
{alex,manolopo}@delab.csd.auth.gr

Abstract. Conducting data mining on logs of web servers involves the determination of frequently occurring access sequences. We examine the problem of finding traversal patterns from web logs by considering the fact that irrelevant accesses to web documents may be interleaved within access patterns due to navigational purposes. We define a general type of pattern that takes into account this fact and also, we present a level-wise algorithm for the determination of these patterns, which is based on the underlying structure of the web site. The performance of the algorithm and its sensitivity to several parameters is examined experimentally with synthetic data.

1 Introduction

Log data which are collected by web servers contain information about user accesses to the web documents of the site. The size of logs increases rapidly due to two reasons: the rate that data are collected and the growth of the web sites themselves. The analysis of these large volumes of log data demands the employment of data mining methods.

Recently, several methods have been proposed for mining web log data [10, 14,16]. Following the paradigm of mining association rules [1], mined patterns are considered to be frequently occurring access sequences. An example of this kind of pattern is a sequence $\langle D_1, \ldots, D_n \rangle$ of visited documents in a web site. If such a sequence appears frequently enough, then it indicates a pattern which can be useful for designing purposes of the web site (advertising), users motivation with dynamic web documents, system performance analysis, etc.

The transformation of web log files to a data format similar to basket data is discussed in [9]. Each user session is split into several transactions, requiring that all accesses inside each transaction are close in time. Then, existing algorithms for mining association rules (e.g. Apriori) are used over the derived transactions. In [7], a more sophisticated transformation (algorithm *MF*) is proposed which identifies inside a transaction all *maximal forward sequences*, i.e. all sequences comprising accesses which have not been made by a backward movement. In

* Work supported by a national PABE project.

J. Štuller et al. (Eds.): ADBIS-DASFAA 2000, LNCS 1884, pp. 215–228, 2000.

the same paper, a variation of mining association rules with hash filtering and transaction trimming is proposed (algorithm *Full Scan–FS*). Also, by exploiting main memory, another algorithm is proposed (*Selective Scan – SS*), which is able to skip several database scans and count candidates belonging to many phases, within the same database scan. Finally, a different approach is followed in [5]. More specifically, one database scan is performed and the number of occurrences of each pair of accesses is determined, i.e. for each pair of web documents A and B the number of times pair AB occurred is counted. Then, the notion of *composite association rule* is defined which is based on the frequencies counted at the previous step. Two algorithms are proposed (*Modified DFS* and *Incremental Step*) for finding composite association rules.

Although all previous approaches mine patterns from access information contained in a log file, these patterns differ significantly. For the first approach, the patterns are standard association rules which are mined with existing algorithms. These rules do not take into account the web site structure. Thus, this approach may overevaluate associations, which do not correspond to the actual way of accessing the site. The method of maximal references finds frequently occurring sequences of consecutive accesses. As will be presented in more detail in Section 2.2, this approach is sensitive to noise, since patterns which are corrupted by noisy accesses, loose the property of consecutiveness. The composite-association rules method counts the frequencies of sequences of length two and, based on these frequencies, estimates the remaining ones of larger sequences, without verifying the estimates with the database contents. This approach is based on the assumption that the probability of an access depends only on the previous access (first-order markov property), which does not hold in all cases.

In this paper, we give a definition of traversal pattern, adopting the meaning given in [7], but we do not pose the constraint that accesses should be consecutive inside the patterns during the frequency counting procedure. These differences from the existing approach require the development of a new algorithm for finding such traversal patterns. We present an algorithm for generating and counting the support of candidate patterns that is level-wise, as the Apriori-like methods, which takes into account the underlying web site structure. Although the general structure of the proposed algorithm is Apriori-like, the data structures and the procedures for support counting and candidate generation differ significantly and these operations are performed efficiently by considering the site structure. The performance of the algorithm is examined experimentally, using synthetic data.

The rest of this paper is organized as follows. Section 2 gives background information and a brief overview of algorithms *Full Scan* and *Selective Scan* [7]. The problem statement is given in Section 3. Section 4 presents the level-wise algorithm, whereas Section 5 contains the performance results of the algorithm. Finally, Section 6 gives the conclusions and directions of future work.

2 Background

2.1 Definitions

A web site can be abstractly viewed as a set of web documents connected with hypertext links. The site can be represented by a simple unweighted directed graph, which is a finite set of vertices and arcs. A vertex corresponds to a document and an arc to a link. Each arc joins an ordered pair of vertices. The graph contains no loops (i.e. arcs joining a vertex with itself), no parallel arcs (i.e. arcs joining the same ordered pair of vertices), whereas no weight (e.g. distance, cost, etc.) is associated with any arc. Since traversal patterns contain information about user access, no quantitative (e.g. weights) or duplicate (e.g. loops, parallel arcs) information has to be considered [1]. An example of a simple, directed graph is illustrated in Figure 1.

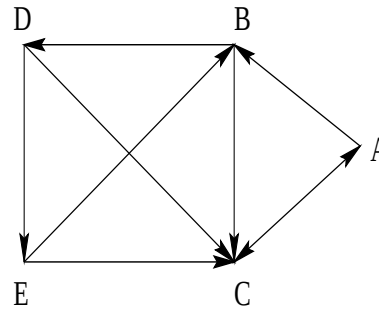

Fig. 1. A directed graph.

A traversal in the site is a sequence of consecutive arcs, i.e. links, that can be represented with the sequence of the terminating vertices of each arc. The length of the traversal is defined by the number of contained vertices. The traversals are contained in the log file. Each entry in the log is of the form (*userID*, *s*, *d*), which denotes the user identification number, the starting position *s* and the destination position *d*. Although the actual representation of a log may be different, since it contains additional information, for the purpose of mining traversal patterns, the log can be abstractly viewed as described previously. The beginning of a new traversal is marked with a triplet (*userID*, *null*, *d*). All pairs corresponding to the same user identification number are grouped together to form a traversal.

In many cases, duplicate arcs or vertices inside the traversal do not contain useful information, like backward movements [7]. This type of traversal where each arc and vertex is distinct, is called a *path*. In the example of Figure 1, $\langle A, B, D, C\rangle$ is a path, whereas $\langle B, D, E, B, C\rangle$ is not.

[1] In case the graph structure is not simple, a simple graph can be derived by omitting loops, multiple edges and weights

Definition 1 *Any subsequence of consecutive vertices in a path is also a path and is called a section. A section is contained in the corresponding path. If P represents a path $\langle p_1, \ldots, p_n \rangle$, then $S = \langle s_1, \ldots, s_m \rangle$ is a section of P, if there exist a $k \geq 0$ such that $p_{j+k} = s_j$ for all $1 \leq j \leq m$.* □

In the example of Figure 1, $\langle A, B, D \rangle$ is a section of path $\langle A, B, D, C \rangle$. The criterion of section containment requires that the vertices (and the corresponding arcs) are consecutive in the path. If this is not required then we define a *subpath*.

Definition 2 *Given a path $P = \langle p_1, \ldots, p_n \rangle$ in a graph G, we call subpath of P, the sequence $SP = \langle sp_1, \ldots, sp_m \rangle$ for which:*

- $\forall\ sp_i \in SP \Rightarrow sp_i \in P$
- *$sp_i = p_k$ and $sp_j = p_l$ and $i < j \Rightarrow k < l, \forall i < j$* □

In other words, the vertices (and arcs) of the subpath belong also to the corresponding path and the order of their appearance in the path is preserved in the subpath. Additionally, the subpath itself is a path in the corresponding graph. In the example of Figure 1, $\langle A, B, C \rangle$ is a subpath of $\langle A, B, D, C \rangle$. The following lemma shows that the notion of subpath is a generalization of that of section.

Lemma 1 *If $P = \langle p_1, \ldots, p_n \rangle$ is a path and $S = \langle s_1, \ldots, s_m \rangle$ a section of P, then S is also a subpath of P.*

Proof. Since P is a path, S is also a path. Also, since S is a section of P, then it holds that: $\forall\ s_i \in S \Rightarrow s_i \in P$. Finally, there exist a $k \geq 0$ such that: $p_{j+k} = s_j$ for all $1 \leq j \leq m$. If $s_j = p_{j+k}$ and $s_{j+1} = p_{j+k+1}$, then for $j < j+1 \Rightarrow j+k < j+k+1$, which is true for all $1 \leq j \leq m$. □

Next, we present some quantitative conclusions regarding the number of subpaths. The proofs can be found in [12].

Lemma 2 *A path $P = \langle p_1, \ldots, p_n \rangle$ of length n has at least $n - k + 1$ and at most $\binom{n}{k}$ subpaths of length k.* □

Corollary 1 *A path $P = \langle p_1, \ldots, p_n \rangle$ of length n has at least two and at most n subpaths of length $n - 1$.* □

Corollary 2 *A path $P = \langle p_1, \ldots, p_n \rangle$ of length n has exactly two sections of length $n - 1$.* □

2.2 Overview of Maximal Reference Sequences

In *algorithm MF* [7], a preprocessing of the log file is performed to extract *maximal forward references*. First, each user session is identified. A session is further decomposed into a number of transactions, [2] which are the paths resulting after

[2] In the sequel, terms transaction and path are used interchangeably.

discarding all backward movements. After this step, a modified algorithm for mining large itemsets is used. This algorithm is called *Full Scan* and resembles *DHP* [13], an Apriori-like algorithm which uses hash-pruning and transaction trimming. Also, algorithm *Selective Scan* is proposed, which exploits the available main memory and merges several phases of the Apriori-like algorithm.

Algorithms *Full Scan* and *Selective Scan* do not find the supports of arbitrary sets of vertices, as the standard algorithms for association rules do, but they take into account the ordering of vertices inside the transactions. These algorithms determine all large *reference sequences*, i.e. all *sections* (see Definition 2) of the graph *contained* in a sufficient number of transactions. Recall that a candidate S is contained as section in a path (transaction) P, if S is a section of P, i.e. $p_{j+k} = s_j$ for some $k \geq 0$. For the candidate generation, a candidate $S = \langle s_1, \ldots, s_{k+1} \rangle$ of length $k+1$ is generated by joining $S' = \langle s_1, \ldots, s_k \rangle$ and $S'' = \langle s_2, \ldots, s_{k+1} \rangle$, if both these sections of S with length k were found large. With respect to Corollary 2, S has only two sections of length k, i.e. S' and S''. In [7] it is not explained if any candidate pruning takes place, like in Apriori and DHP where every subset is tested if it is large. Since S' and S'' are the only sections of S, any other combination of k vertices of S, even if it is a path in the graph, will not be supported by the same transactions which support S. If a transaction P contained S and a combination of k vertices S''' from S, then S''' would be contained as a section by S, which contradicts with Corollary 2. Therefore, no further candidate pruning can take place.

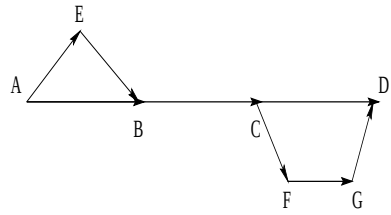

Fig. 2. A transaction corrupted by irrelevant accesses during navigation.

The resulting patterns are all large *maximal reference sequences*, i.e. all paths with the maximum possible length, contained as sections in a number of transactions which is larger than *minSupport*. Compared to standard association rules, maximal reference sequences are traversal patterns. However, patterns may be corrupted by noisy accesses which are random accesses, not parts of a pattern, which are done during user navigation. Thus, inside a user traversal, patterns are interleaved with noise. Figure 2, illustrates a transaction $T = \langle A, E, B, C, F, G, D \rangle$. If $P = \langle A, B, C, D \rangle$ is the pattern (maximal reference sequence), then P is not a section of T and thus, it is not supported by transaction T. If many transactions are corrupted, then pattern P will not have adequate support and will be missed. Therefore, irrelevant accesses for navigational purposes have an impact on the support of the patterns and their determination.

3 Mining Access Patterns Based on Subpath Definition

3.1 Problem Statement

Given a collection of transactions which are paths in a web site represented by a graph, all paths contained as subpaths by a fraction of transactions which is larger than *minSupport* are found. Following the notation of standard association rules, this fraction is called *support* and all such paths are called *large*. As described in Section 2.1, in the sequel we use the terms of vertex and arc instead of web document and hypertext link. Finally, as a post-processing step, the large paths of maximal length can be determined. Since this is a straightforward operation, it will not be considered any further.

The database consists of paths in the given graph which can be derived from the log file using algorithm *MF* [7]. A candidate path P is supported by a transaction T, if P is a subpath of T. Thus, vertices corresponding to noisy accesses in T do not affect the support of P. In the example of Figure 2, pattern $P = \langle A, B, C, D\rangle$ is a subpath of transaction $T = \langle A, E, B, C, F, G, D\rangle$ although it is not its section. Recall that P is a path in the graph and the ordering of vertices in T is preserved. Following Lemma 1, the set of all large paths with the subpath-containment criterion is a superset of all large references sequences with the section-containment criterion. Figure 3 illustrates an example with a database of five path transactions from the graph of Figure 1 with *minSupport* equal to two.

Database

ID	Path
1	$\langle A, B, C\rangle$
2	$\langle B, D, E, C, A\rangle$
3	$\langle C, A, B\rangle$
4	$\langle D, C, A\rangle$
5	$\langle B, C, A\rangle$

C_1

Candidate	Sup.
$\langle A\rangle$	5
$\langle B\rangle$	4
$\langle C\rangle$	5
$\langle D\rangle$	2
$\langle E\rangle$	1

L_1

Candidate	Sup.
$\langle A\rangle$	5
$\langle B\rangle$	4
$\langle C\rangle$	5
$\langle D\rangle$	2

C_2

Candidate	Sup.
$\langle A, B\rangle$	2
$\langle A, C\rangle$	1
$\langle B, C\rangle$	3
$\langle B, D\rangle$	1
$\langle C, A\rangle$	4
$\langle D, C\rangle$	2

L_2

Candidate	Sup.
$\langle A, B\rangle$	2
$\langle B, C\rangle$	3
$\langle C, A\rangle$	4
$\langle D, C\rangle$	2

C_3

Candidate	Sup.
$\langle A, B, C\rangle$	1
$\langle B, C, A\rangle$	2
$\langle C, A, B\rangle$	1
$\langle D, C, A\rangle$	2

L_3

Candidate	Sup.
$\langle B, C, A\rangle$	2
$\langle D, C, A\rangle$	2

Fig. 3. Example of large path generation.

The differences in mining large paths compared to mining standard association rules and reference sequences, require the development of a new algorithm which has to be based on the graph structure and take into account the notion of subpath containment.

3.2 Pruning with the Support Criterion

Apriori pruning criterion [1] requires that a set of items $I = \{i_1, \ldots, i_n\}$ is large only if every subset of I of length $n-1$ is also large. In case of a path $S = \langle s_1, \ldots, s_n \rangle$, pruning can be performed based on the following lemma:

Lemma 3 *A path $S = \langle s_1, \ldots, s_n \rangle$ is large only if every subpath S' of S, with length $n-1$ is also large.*

Proof. See [12]. □

With respect to Corollary 1, a path of length n has at least two and at most n subpaths of length $n-1$. Notice that in case of mining large reference sequences [7], only the two sections of the path are considered (see Corollary 2). For the determination of large paths, only the subpaths have to be tested, whose number may be less than $\binom{n}{n-1} = n$, as it is the case for itemsets. Therefore, not any other combination of vertices, besides the subpaths, should be tested.

4 Determination of Large Paths

The determination of large paths can be performed in a level-wise manner, as in the Apriori algorithm [1]. The general structure of the algorithm is given bellow. The minimum required support is denoted as *minSupport*. C_k denotes all candidates of length k, L_k the set of all large paths of length k, D the database and G the graph.

Algorithm 1: Level-wise determination of large paths over a graph G
1) $C_1 \leftarrow$ the set of all paths of length 1
2) $k = 1$
3) **while** $(C_k \neq \emptyset)$ {
4) **for each** path $p \in D$ {
5) $S = \{s | s \in C_k,\ s$ is subpath of $p\}$
6) **for each** $s \in S$, s.count++
7) }
8) $L_k = \{s | s \in C_k, s.\text{count} \geq \textit{minSupport}\ \}$
9) $C_{k+1} \leftarrow$ genCandidates(L_k, G)
10) k++
11) }

Although the general structure of the level-wise algorithm is similar to Apriori, its components for

a. candidate support counting (steps 5 and 6), and
b. the generation of the candidates of the next phase (step 9)

differ significantly since the problem of determining large paths, as stated in Section 3.1, presents several differences compared to the one of finding large

itemsets. First, candidates have to be paths in the graph and not arbitrary combination of vertices. Thus, the procedure of candidate generation (step 9) has to form only valid candidates. It also has to perform apriori pruning with respect to Lemma 3. For support counting (steps 6 and 7), the subpath containment has to be performed with respect to Definition 2. These differences require an algorithm which takes into account the graph structure. Additionally, although existing data structures like the *hash-tree* and *hash-table* can be used for determining large paths, as in [1,7], we use a trie data structure for counting the supports and for storing the large paths. The procedure of generating candidate paths for the next phase of the algorithm is performed with an efficient recursion manner over the trie. It is necessary to notice that all improvements to the Apriori algorithm, as in [13], are also applicable to *Algorithm 1.* Therefore, they are not used in order to make the differences more clear.

4.1 Data Structures

First, we need to store the graph in a main memory data structure. Although several approaches for the representation of graphs in secondary storage have been reported [8,11], however, we assume that the graph size is such that the graph can fit in main memory. The reason is that for the typical example of a web site, the total number of graph vertices is less than a few thousands. The graph is represented with its *adjacency lists*, which hold a list with all vertices connected with an arc starting from graph vertex v, for each v. This list is denoted as $N^+(v)$ and is called the *positive neighborhood* of v [3]. The adjacency list representation is more appropriate for less dense graphs, i.e. graphs with not many arcs.

The candidate paths are held in a trie, an approach which is also followed in [6,15] for itemsets. An important difference is that the fanout (i.e. the number of branches from a trie node) in the case of paths is much smaller, compared to the case of itemsets where any combination of items forms an itemset. This way, the trie occupies less space. Large paths remain in the trie to advocate the procedure of candidate generation.

4.2 Support Counting

The support counting for candidate paths (steps 5 and 6 in algorithm) is the most computationally intensive part of *Algorithm 1.* At the k-th database scan, the support of candidates of length k is counted. For each transaction P with length n that is read from the database, all possible subpaths of length k have to be determined (if P's length is less than k, it is ignored). P is decomposed into all its $\binom{n}{k}$ possible combinations of vertices (see Lemma 2) and each one is searched. For those which exist in the trie, i.e. they are subpaths of P, their

[3] The negative neighborhood $N^-(v)$ consist of all vertices w for which there is a vertex from w to v.

support is increased by one. For example, if path $P = \langle A, B, C, D\rangle$ and $k = 3$, then $\langle A, B, C\rangle$, $\langle B, C, D\rangle$, $\langle A, B, D\rangle$ and $\langle A, C, D\rangle$ are searched.

Since, in general, P has less than $\binom{n}{k}$ subpaths, an unsuccessful search is performed for each combination which is not a subpath. In the previous example, $\langle A, B, C\rangle$ and $\langle B, C, D\rangle$ are definitely subpaths (see Corollary 1), while the remaining two may not be subpaths. In the worst case, the complexity of each unsuccessful search is $O(k)$. Thus, the total cost in the worst case is $O(k \cdot [\binom{n}{k} - 2])$. The search for those combination of vertices, which are not subpaths, could be avoided only if for any combination of k vertices, it can be known that there exist a path in the graph connecting these vertices. Algorithms which find the closure of the graph can recognize if there exist a path connecting only two vertices. Since the determination of the path existence would perform a search in the graph with cost $O(k)$ at the worst case, the total cost would again been equal to $O(k \cdot [\binom{n}{k} - 2])$.

As mentioned earlier, the advantage of the trie over the hash-tree, is that, since candidates in hash-tree are only at leafs, at best k and at worst $k + k \cdot m$ comparisons are required for looking up a candidate, where m is the number of candidates stored in a leaf of the hash-tree [1]. On the other hand, in case of a trie, this operation requires k comparisons at the worst case. Apparently, hash-tree requires less memory but, as explained, by storing only paths instead of all arbitrary vertex combinations, memory requirements for the trie are reduced significantly.

4.3 Candidate Generation

After having scanned the database, all large candidates of this phase have to be determined (step 8) and all candidates for the next phase have to be generated (step 9). In Apriori [1], these two steps are performed separately. First, all large candidates L_k at phase k are determined and stored in a hash table. Then, a join $L_k \bowtie L_k$ is performed, using the hash table, for the generation of candidates of phase $k + 1$. For each candidate of length $k + 1$, which belongs to the result of $L_k \bowtie L_k$, all its k subsets of length k are searched whether they belong to L_k (using again the hash table).

Joining $L_k \bowtie L_k$ is done with respect to the first $k - 1$ items of the itemset. For example, if $k = 3$ and two itemsets $I_1 = \{1, 2, 3\}$ and $I_2 = \{1, 2, 4\}$ are large, then they are joined to form a possible candidate $C = \{1, 2, 3, 4\}$. If I_1 or I_2 does not exist, then C will not be a candidate since not all its subsets are large (I_1 or I_2 are not large). Therefore, joining is performed for avoiding generating candidates which will be pruned by the apriori-pruning criterion. As it is easy to show, it is equivalent as if joining on any fixed $k - 1$ items was done, instead of the first $k - 1$ ones.

In case of path candidates, their generation cannot be based on joining $L_k \bowtie L_k$, on a fixed combination of $k - 1$ vertices, because each candidate path C has a different number of large subpaths. For example, the first $k - 1$ vertices (as in Apriori) may not be present in any other subpath besides the

one of the two sections of C (see Corollaries 1 and 2). Moreover, this joining in Apriori is performed to reduce the number of examined possible candidates, since any item can be appended to a candidate of length k to produce a possible candidate of length $k+1$. On the other hand, for candidate paths, only the extensions from the vertices in the positive neighborhood of the last vertex are considered, whose number is much less compared to the case of itemsets.

Therefore, for the generation of candidates of the next phase for *Algorithm 1*, a different approach is followed and steps 8 and 9 are performed together. By visiting all trie leaves, if a candidate $L = \langle \ell_1, \ldots \ell_k \rangle$ is large, then the adjacency list of $N^+(\ell_k)$ (last vertex) is retrieved. For each vertex v in the adjacency list, which does not belong to path L and subpath $L' = \langle \ell_2, \ldots, \ell_k, v \rangle$ is large, a possible candidate $C = \langle \ell_1, \ldots, \ell_k, v \rangle$ of length $k+1$ is formed by appending v at the end of L. Then, all subpaths of C of length k, besides L', are searched in the trie and if all are large, then C is considered to be a candidate of length $k+1$ by adding a branch in the trie from vertex ℓ_k to vertex v. The following algorithm describes the candidate generation procedure.

Procedure: genCandidates(L_k, G)
//L_k is the set of large paths of length k and G is the graph
for each large leaf $L = \langle \ell_1, \ldots, \ell_k \rangle$ of trie {
    $N^+(\ell_k) = \{v \mid$ there is an arc $\ell_k \rightarrow v$ in $G\}$
    for each $v \in N^+(\ell_k)$ {
        if (v not already in L) **and** $L' = \langle \ell_2, \ldots, \ell_k, v \rangle$ is large {
            $C = \langle \ell_1, \ldots, \ell_k, v \rangle$
            if ($\forall$ subpath $S \neq L'$ of $C \Rightarrow S \in L_k$)
                insert C in the trie by extending ℓ_k with a branch to v
        }
    }
}

Correctness. See [12]. □

All $k-2$ possible subpaths $S \neq L'$ of length k of a candidate C have to be searched in the trie structure to verify if they are large. During an unsuccessful search of a possible subpath S, it has to be determined if it is not in the trie because it is not large or because it is not a valid subpath. Thus, if $S = \langle s_1, \ldots, s_k \rangle$ and vertex s_i is not present, then it has to be tested if there is an arc $s_{i-1} \rightarrow s_i$ in the graph. If not, then S is not a valid subpath and is ignored. Otherwise, S does not exist, because vertex s_{i-1} was not expanded in a previous phase since subpath $\langle s_1, \ldots, s_{i-1} \rangle$ was not large therefore, S is also not large and thus, C is pruned.

With regards to the efficiency of the procedure *genCandidates* we notice the following. For testing if the subpaths of a candidate are large, there is no need to create a separate hash table (as in case of Apriori) because large candidates are already present in the trie. Testing if a vertex from the adjacency list is already present in the candidate path is done by using a temporary bitmap, which is maintained during the visit of the trie leaves. This way, testing containment is

done in O(1). The way trie leaves are expanded by using the adjacency list of their terminating vertex justifies the selection of the graph representation with its adjacency lists. Finally, the trie grows dynamically with simple leaf extensions and there is no need to create a hash-tree from the beginning, as in Apriori.

5 Performance Results

This section contains the results of the experimental evaluation of *Algorithm 1* using synthetic data. The experiments were run in a workstation with one Pentium III processor 450 MHz, 256 MB RAM, under Windows NT 4.0.

5.1 Synthetic Data Generator

First, the site structure has to be generated. Two significant parameters are the total number N of web documents and the maximum number F of hypertext links inside a document. Following the results of [3,4], each web document has a size (in KB) which follows a mixed distribution: Lognormal, for sizes less than 133 KB and Pareto for larger ones. This way, as presented in [4], 93% of the documents are html documents and the remaining ones are large binary objects (images, sounds, etc). For each document, its fanout, i.e. the number of its hypertext links, is determined following a uniform distribution between 1 and F. Documents with sizes larger than 133 KB do not have links because we assume they correspond to large binary objects.

Table 1. Symbols representing parameters of synthetic data.

Symbol	Definition
N	number of vertices
F	maximum fanout
L	average pattern length
P	total number of patterns
C	corruption level

Large paths are chosen from a collection of P paths which represent the patterns. The length of each path pattern is determined by a Poisson distribution with average length L. For the formation of each transaction, representing a user traversal, one of the P path patterns is chosen with uniform distribution. The corruption of patterns is represented with the corruption level C, which is the number of vertices from the path pattern which will be substituted. This number follows a Poisson distribution with average value C. Table 1 presents all the symbols used in the following. Datasets are characterized by the values for parameters L, F and D. For example, $L5F3D100K$ denotes a dataset with average path pattern length 5, maximum fanout 3 and 100,000 transactions. More details for the synthetic data generator can be found in [12].

5.2 Results

First, we examined the scale-up properties of *Algorithm 1* with respect to the number of database path transactions. Figure 4a illustrates the execution times for databases with 1×10^6, 2.5×10^6 and 5×10^6 transactions. For this experiment, the average pattern length $L = 7$, the number of patterns $P = 1,000$, the maximum fanout $F = 15$, and the corruption level $C = 25\%$. Thus, these sets can be denoted as $L7F15D1M$, $L7F15D2.5M$ and $L7F15D5M$. Results are represented for three characteristic values of *minSupport*. As it can be noticed from Figure 4a, *Algorithm 1* presents a linear scale-up with respect to the database size.

Also, we examined the scale-up properties of *Algorithm 1* with respect to the site size, i.e. the number of the graph vertices. We used $P = 1,000$, $L = 7$ and $D = 100,000$ transactions. The maximum fanout $F = 15$, since for large graphs we would like that they are not very sparse, whereas $C = 25\%$. The dataset is denoted as $L7F15D100K$ and Figure 4b illustrates the results for three number of vertices: 1000, 2500 and 5000, again for the same three *minSupport* values. As it can be seen, the execution times increases with respect to the number of vertices since there are more candidates to be examined.

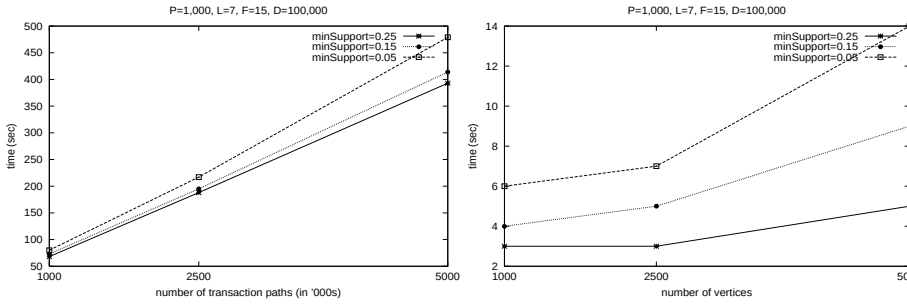

Fig. 4. Scale-up results w.r.t.: **a.** number of transactions, **b.** number of web documents (graph vertices).

Then, we examined the properties of *Algorithm 1* with respect to the pattern length. For this experiment, $P = 1000$ patterns, $N = 1000$ vertices, $D = 100,000$ transactions, the maximum fanout $F = 15$ and the corruption level $C = 25\%$. Figure 5a illustrates the results for three average pattern lengths: 5, 10 and 15. As shown, the execution time increases for larger lengths, especially for lower *minSupport* values, since more phases and thus, database scans are required.

Finally, we tested the impact of the corruption level. Recall that this parameter represents the expected number of vertices within a pattern which are corrupted with other vertices (i.e. not part of the pattern). Figure 5b illustrates the number of large paths founded in the result with respect to three values of corruption level: 0.1%, 0.25% and 0.4%. The dataset was $L7F15D100K$ with

$P = 1000$ patterns. As it is expected, the number of large paths reduces with respect to the corruption level because patterns are changed. However, the reduction is not remarkable since pattern identification is not affected significantly by the corruption.

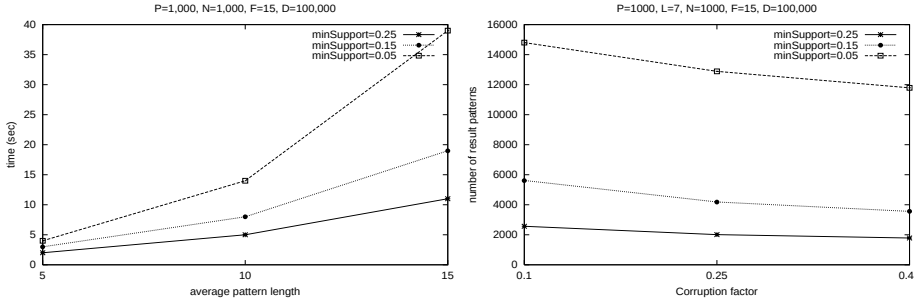

Fig. 5. a. Execution time w.r.t. average pattern length. **b.** Number of large paths w.r.t. the corruption level.

6 Conclusions

We examined the problem of determining traversal patterns from web logs. These patterns take into account the fact that irrelevant accesses, made for navigational purposes, may be interleaved with accesses which are not part of the pattern. We presented a level-wise algorithm for counting the support of these patterns. This algorithm differs from the existing Apriori-like algorithms because it has to take into account the graph structure which represents the web site. The performance of the algorithms is tested experimentally with synthetic data which are produced following several results on recent web statistics.

The quality of the patterns can only be tested within the framework of a specific application. In the future, we plan to test the proposed patterns for the purpose of web prefetching, i.e. the prediction of forthcoming web document requests of a user. For such an application, the impact of noise, as defined in this paper, will be tested more precisely.

Acknowledgments. We would like to acknowledge the effort of Mr. Dimitrios Katsaros on the development of the synthetic data generator.

References

1. R. Agrawal and R. Srikant: "Fast Algorithms for Mining Association Rules", *Proceedings Very Large Data Bases Conference (VLDB'94)*, pp.487-499, 1994.
2. R. Agrawal and R. Srikant: "Mining Sequential Patterns", *Proceedings International Conference on Data Engineering (ICDE'95)*, pp.3-14, 1995.

3. M. Arlitt and C. Williamson. "Internet Web Servers: Workload Characterization and Performance", *IEEE/ACM Transactions on Networking*, Vol.5, No.5, 1997.
4. P. Barford and M. Crovell: "Generating Representative Web Workloads for Network and Server Performance Evaluation", *Proceedings ACM Conference on Measurement and Modeling of Computer Systems (SIGMETRICS'98)*, pp.151-160, 1998.
5. J. Borges and M. Levene: "Mining Association Rules in Hypertext Databases", *Proceedings Conference on Knowledge Discovery and Data Mining (KDD'98)*, pp.149-153, 1998.
6. S. Brin, R. Motwani, J. Ullman and S. Tsur: "Dynamic Itemset Counting and Implication Rules for Market Basket Data", *Proceedings ACM SIGMOD Conference (SIGMOD'97)*, pp.255-264, 1997.
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*, Vol.10, No.2, pp.209-221, 1998.
8. Y. Chiang, M. Goodrich, E. Grove, R. Tamassia, D. Vengroff and J.S. Vitter: "External-Memory Graph Algorithms", *Proceedings Symposium on Discrete Algorithms (SODA'95)*, pp.139-149, 1995.
9. R. Cooley, B. Mobasher and J. Srivastava: "Data Preparation for Mining World Wide Web Browsing Patterns", *Knowledge and Information Systems*, Vol.1, No.1, pp.5-32, 1999.
10. K. Joshi, A. Joshi, Y. Yesha and R. Krishnapuram: "Warehousing and Mining Web Logs", *Proceedings Workshop on Web Information and Data Management*, pp.63-68, 1999.
11. M. Nodine, M. Goodrich and J.S. Vitter: "Blocking for External Graph Searching", *Proceedings ACM PODS Conference (PODS'93)*, pp.222-232, 1993.
12. A. Nanopoulos and Y. Manolopoulos: "Finding Generalized Path Patterns for Web Log Data Mining", Technical report, Aristotle University, http://delab.csd.auth.gr/publications.html, 2000.
13. J.S. Park, M.S. Chen and P.S. Yu: "Using a Hash-based Method with Transaction Trimming for Mining Association Rules", *IEEE Transactions on Knowledge and Data Engineering*, Vol.9, No.5, pp.813-825, 1997.
14. J. Pei, J. Han, B. Mortazavi-Asl and H. Zhu: "Mining Access Patterns Efficiently from Web Logs", *Proceedings Pacific-Asia Conference on Knowledge Discovery and Data Mining (PAKDD'00)*, 2000.
15. Y. Xiao and M. Dunham: "Considering Main Memory in Mining Association Rules", *Proceedings Conference on Data Warehousing and Knowledge Discovery (DaWaK'99)*, pp.209-218, 1999.
16. O. Zaiane, M. Xin and J. Han: "Discovering Web Access Patterns and Trends by Applying OLAP and Data Mining Technology on Web Logs", *Proceedings on Advances in Digital Libraries (ADL'98)*, pp.19-29, 1998.