ELSEVIER

Data & Knowledge Engineering 37 (2001) 243–266

DATA & KNOWLEDGE ENGINEERING

www.elsevier.com/locate/datak

Mining patterns from graph traversals [☆]

Alexandros Nanopoulos [a], Yannis Manolopoulos [b,*,1]

[a] Data Engineering Lab, Department of Informatics, Aristotle University, Thessaloniki 54006, Greece

[b] Department of Computer Science, University of Cyprus, Nicosia 1678, Cyprus

Received 3 August 2000; received in revised form 22 November 2000; accepted 24 January 2001

Abstract

In data models that have graph representations, users navigate following the links of the graph structure. Conducting data mining on collected information about user accesses in such models, involves the determination of frequently occurring access sequences. In this paper, the problem of finding traversal patterns from such collections is examined. The determination of patterns is based on the graph structure of the model. For this purpose, three algorithms, one which is level-wise with respect to the lengths of the patterns and two which are not are presented. Additionally, we consider the fact that accesses within patterns may be interleaved with random accesses due to navigational purposes. The definition of the pattern type generalizes existing ones in order to take into account this fact. The performance of all algorithms and their sensitivity to several parameters is examined experimentally. © 2001 Elsevier Science B.V. All rights reserved.

Keywords: Web log mining; Path traversal; Graph model

1. Introduction

Several data models have a, direct or implicit, graph representation. Object-oriented models, like ODMG, can be represented with a graph structure which defines the relationships and the hierarchy of objects. Hypertext systems, as in case of Web, can also be modeled with graph structures containing the documents and the hypertext references between them. The web documents themselves, although they do not contain a rigid structure, are considered as semi-structured data. The proposed models for semi-structure data, like OEM [1], use a labeled graph representation of the documents.

For data with graph representation, several components of the graph, which are present in a significant number of data, can be consider as structural patterns. For example, from a set of semi-structured documents, parts of the documents, corresponding to the same components of the graph representation, which appear in a large number of documents can determine a synopsis of

[☆] A preliminary version of this work appeared in [14].

[*] Corresponding author. Tel.: +3031-996363; fax: +3031-996360.

E-mail addresses: alex@delab.csd.auth.gr (A. Nanopoulos), manolopo@ucy.ac.cy (Y. Manolopoulos).

[1] On sabbatical leave from Data Eng. Lab, Dept. of Informatics, Aristotle Univ.

PII: S0169-023X(01)00008-8

the complete document collection and are used as *data guides*. The problem of finding structural patterns in labeled graph representations (*OEM*) of semi-structured documents is examined in [21], where patterns have a tree-like structure. The efficiency of the method is based on constraining the search space by having the roots of the tree-like patterns belong to a user-defined set of documents.

Every data model provides means for user navigation, like query languages or browsers. Users may navigate through the contents of the model and this corresponds to a traversal through the graph structure which represents the model. Navigation can be done by using a browser and following links, as in the case of hypertext documents or by using path expressions from a query language, as in the case of object oriented models. For instance, let a query regarding objects of type *quide* and attributes *restaurant*, *address* and *zipcode*. The query contains a path expression 'guide.restaurant.address.zipcode', corresponds to a traversal through the graph representation of the model, since all query attributes correspond to vertices connected in the graph. Information regarding user navigation is usually collected and it contains the traversals of users through the graph. For example, in the case of Web, information about user access is collected in a log. From this collected information, patterns can be discovered, indicating the way user access is performed. Access patterns, different from structural patterns, do not have general tree structure but are sequences of accesses. An example of an access pattern is a sequence $\langle D_1, \ldots, D_n \rangle$ of visited hypertext documents, which has a large occurrence frequency. For the purpose of access pattern determination, each document D_i is considered as atomic, i.e., its structure is not taken into account.

The volume of collected information about user access and the size of the models, and consequently the size of the corresponding graph structures, tend to increase rapidly, especially in case of Web. For this reason, the extraction of access patterns requires the employment of data mining methods. Discovered access patterns may find several applications. For example, in the case of Web, patterns are useful for web site designing purposes (advertising), motivation of users with adaptive web sites and system performance analysis. Access patterns from queries with path expressions can be useful for selecting attributes for index construction or schema modifications.

The problem of mining access patterns from traversals through a data model with a graph structure representation, differs from those of mining sequential patterns or determining large itemsets for association rules generation. As with sequential patterns, the ordering of accesses inside the patterns is important whereas, for large itemset generation, ordering is not taken into account. However, sequential patterns do not consider the graph structure, through which navigation is done. Since access patterns are mined from graph traversals, the patterns themselves should also correspond to graph traversals. If accesses are considered just as events ordered in time, as in case of sequential patterns, the fact that they are done following the links of the graph structure is neglected.

Related work involves web log data mining methods [12,13,19,23,17] since their objective is to discover patterns from web page accesses. However, research has been focused on issues of representing the log entries in a format similar to basket data. Most of the proposed approaches use existing, or slightly modified, algorithms from the domains of mining association rules and sequential patterns from basket data. Despite of the graph structure of the Web, almost all methods do not consider this structure. Of course, in some applications the knowledge of the structure cannot be assumed. For example, it is the complete Web in the case of a proxy server log. But in most cases, as in the example of a web server log, it is the structure of the particular web site, which is available.

More particularly, research on web log data mining algorithms include the following. Several issues about the transformation of a web server log to a data format similar to basket data are discussed in [11] and patterns are mined with existing algorithms for mining association rules (Apriori). Two algorithms, *Full Scan* (*FS*) and *Selective Scan* (*SS*), are proposed in [9] for determining accesses sequences which have high occurrence frequency. These sequences consist of accesses which are consecutive in the log and their determination is performed with a modification of the *DHP* algorithm [16], which is used for mining association rules. A different approach is followed in [7], where the hypertext graph is considered. The notion of *composite association rule* is defined as a graph traversal with high occurrence frequency. Two algorithms, *Modified DFS* and *Incremental Step*, are proposed for finding composite association rules, which are based on a *depth first search* method of the hypertext graph structure. However, these algorithms scan the database of user traversals only once to count the occurrence frequency of pairs of accesses. The frequencies of traversals with larger lengths are estimated based on the frequencies of pairs of accesses, without verifying them in the sequel with the database contents. This approach is based on the assumption that user navigation is a *first-order markov procedure*.

In this paper, we focus on the problem of finding access patterns from a database of user traversals through a given graph structure. The main difference from existing approaches is that access patterns are also considered as graph traversals and not as arbitrary access sequences. These type of patterns indicate the exact way that navigation is done through a graph structure, i.e., by following the links of the structure. The proposed method extends existing algorithms for mining association rules in order to take the graph structure into account during support counting, candidate generation and pruning. In contrast to the approach followed in [7], which also considers the graph structure, the support (i.e., occurrence frequency) of the patterns are not estimated but are counted directly from the database contents. For this reason, Ref. [7] did not contain methods for support counting, candidate generation and pruning thus, the algorithms in [7] cannot be used for the proposed problem. Therefore, the proposed method, differently from existing ones, considers both the database contents and the graph structure for the determination of access patterns.

In the sequel, we assume that user traversals have been collected in a database and we do not examine any new methods of preprocessing user access information. Also, we do not consider any particular details of specific data models, besides the graph representation. First, we present a level-wise algorithm which is Apriori-like. Additionally, we present two non-level-wise algorithm which perform support counting of candidates with different lengths. The first algorithm is based on an existing one which merges several phases of the level-wise algorithm. The second one is based on a new method which estimates the support of candidates of different lengths and selects the ones that have large estimated support. The actual supports are determined in the database. The performance of all algorithms is examined experimentally using synthetic data.

The rest of this paper is organized as follows. Section 2 gives background information. Also, existing methods for web log data mining are surveyed briefly and the motivation is presented. The problem statement is given in Section 3 and the level-wise algorithm is presented in Section 4. Section 5 presents the selective algorithms, whereas the performance results of the algorithms are presented in Section 6. Finally, Section 7 contains the conclusions and directions of future work.

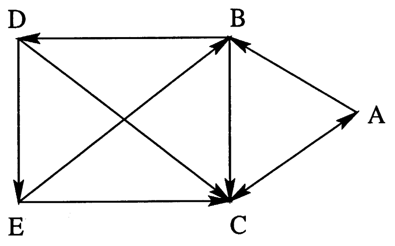


Fig. 1. A directed graph.

2. Background and motivation

2.1. Definitions

First, we give a general definition of a graph. An example is illustrated in Fig. 1.

Definition 1. A simple unweighted directed graph is a finite set of vertices and arcs. Each arc joins one ordered pair of vertices (directed). The graph contains no loops (simple), i.e., arcs joining a vertex with itself, no parallel arcs, i.e. arcs joining the same ordered pair of vertices. Additionally, no label is associated with any arc [2] (unweighted).

A traversal through the graph is a sequence of consecutive arcs which, for simple graphs, 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. A traversal which may contain both duplicate vertices and arcs is called a *walk*. In many cases, duplicate arcs or vertices inside the traversal do not contain useful information, like backward movements [9]. If arcs are distinct, the traversal is called *trail* whereas, if each arc and vertex is distinct, the traversal is called *path*. In the example of Fig. 1, $\langle A, B, C, A, B\rangle$ is a walk, $\langle A, B, D, E, B, C\rangle$ is a trail and $\langle A, B, D, C\rangle$ is a path.

Definition 2. 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 $P = \langle p_1, \ldots, p_n\rangle$ is a path, then $S = \langle s_1, \ldots, s_m\rangle$ is a section of P if there exist a $k \geqslant 0$ such that $p_{j+k} = s_j$ for all $1 \leqslant j \leqslant m$.

In the example of Fig. 1, $\langle A, B, D\rangle$ is a section of path $\langle A, B, D, C\rangle$. Next, we give a lemma on the number of the sections of a path, which is used in the sequel.

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

Proof. From P we can derive paths $S' = \langle p_1, \ldots, p_{n-1}\rangle$ (by omitting the last vertex) and $S'' = \langle p_2, \ldots, p_n\rangle$ (by omitting the first vertex) which are sections of P because their vertices are consecutive in P. Any other combination of $n-1$ vertices from P, even if corresponding to a path, is not a section of P because its vertices are not consecutive in P. $\square$

[2] In case the graph is not simple, the corresponding simple one can be derived by omitting loops and parallel arcs.

The specification of traversals from user navigational information depends on the specific application. For example, in the case of path expressions in user queries, each path expression is a traversal. On the other hand, for hypertext navigation, the user accesses are usually contained in a log. Each entry in the log corresponds to an access and is of the form (*userID*, *s*, *d*, *t*). The starting position is denoted by *s*, *d* denotes the destination position and *t* denotes time. All accesses that correspond to the same user identification number and are near by in time, are grouped together and they form a traversal. For several issues about the extraction of traversals from web server logs, see [11]. In the following, we assume that traversals have been extracted and are contained in a database.

In the following sections, we use terminology which is based on work on association rule mining [2]. Therefore, an *itemset* is a collection of items, the *support* of an itemset is the fraction of transaction itemsets that contains the itemset and *large* is an itemset that has support larger than a user-defined threshold. Moreover, for traversal patterns, a *transaction* is a user path traversal. The *database* is a collection of transactions and the *graph* structure G consists of the vertices and arcs of the given model (see Definition 1).

2.2. Overview of web log mining methods

In this section, we briefly describe existing methods for web log data mining. Since their objective is to discover patterns from user accesses contained in web logs, we examine whether each of them can be used for mining access patterns from traversals through a graph, having in mind that access patterns should correspond to graph traversals. More specifically, we consider standard association rules, sequential patterns, reference sequences and composite association rules.

2.2.1. Standard association rules and sequential patterns

Mining association rules requires the determination of *user transactions*, like in the case of basket data. Several issues of extracting transactions from a web server log are examined in [11]. If we consider the hypertext graph structure, each transaction corresponds to a graph traversal and it has the form $T = \langle v_1, \ldots, v_n \rangle$, where v_i are vertices of the graph (corresponding to web documents). These traversals are walks because it is not required that $v_i \neq v_j$ for $i \neq j$. After transaction determination, [11] uses standard algorithms for mining association rules (i.e. Apriori). Thus, the resulted patterns are association rules of the form $\{v_1, \ldots, v_m\} \Rightarrow \{w_1, \ldots, w_n\}$ (v_i, w_i are graph's vertices). The support of the rule is denoted as $fr(\{v_1, \ldots, v_m, w_1, \ldots, w_n\})$ and it is equal to the fraction of transactions that contain as *subset* the set of vertices $\{v_1, \ldots, v_m, w_1, \ldots, w_n\}$. The confidence of the rule is equal to

$$\frac{fr(\{v_1, \ldots, v_m, w_1, \ldots, w_n\})}{fr(\{v_1, \ldots, v_m\})}.$$

Sets $\{v_1, \ldots, v_m\}$ with support larger than a user defined value are called *large* and are arbitrary combination of vertices. Thus, these sets do not correspond in general to graph traversals because a pair (v_i, v_{i+1}) is not necessary an arc in the graph. Actually, in [11] a technique called *site filtering* is proposed for optional pruning of association rules which contain vertices connected with arcs in the graph. Even if site filtering is not used, the approach of finding large itemsets with standard association rules algorithms cannot determine effectively traversal patterns. Unlike a transaction of basket data, ordering of vertices has clearly a meaning in case of mining traversal patterns. By using standard algorithms for association rules, support counting is done with the *subset* criterion which does not take into account the ordering of vertices inside a transaction. For example, in the graph of Fig. 1, an access sequence $S = \langle C, D, B \rangle$ may be considered as large, although it does not correspond to a traversal. Since ordering is not considered, several transactions like, for example, $T = \langle A, B, C, D \rangle$, will contain S as a subset. This way, patterns are not necessarily traversals through the graph.

Sequential patterns [3] have been used for the purpose of web log mining [6]. They take into account only the ordering of accesses and not the graph structure. Therefore, patterns do not correspond to graph traversals. In the previous example, let a user sequence T, which contains two transactions. If $T = \langle (A, B, D), (B, D, E) \rangle$, then a possible pattern $P = \langle (B)(E) \rangle$ is supported by T, although a sequence of length two, starting from B and terminating to E is not a traversal in the graph of Fig. 1.

2.2.2. *Maximal reference sequences*

In algorithm *MF* of [9], user traversals are called *maximal forward references*. Each of them forms a transaction and it is a sequence of consecutive accesses by the same user, which contains no backward movements. Then, algorithms *FS* and *SS* are used to extract patterns. These algorithms resemble *DHP* [16], an Apriori-like algorithm which uses hash-pruning and transaction trimming. *SS* exploits available main memory and skips several database scans. The support counting for both algorithms is not done for arbitrary combinations of accesses, as in case of standard associations rule mining, but only for subsequences of accesses sequences, called *reference sequences*. A reference sequence contains accesses which are consecutive in a maximal forward reference (transaction). Discovered patterns are all *maximal reference sequences* which are reference sequences contained in a number of transactions which is larger than a user-defined parameter and they are not contained by any other reference sequence.

If the graph structure is taken into account, then, each maximal reference sequence corresponds to a path, since it consists of consecutive accesses to vertices that are connected with an arc and there are no duplicate vertices (backward movements are discarded). In the same manner, reference sequences are also paths and their support is counted using the *section containment* criterion. Recall from Definition 2 that a section S is contained by a path P if $P_{j+k} = S_j$, for some $k \geqslant 0$. Thus, the approach in [9] can be used for mining access patterns which are paths (i.e., reference sequences) contained as sections by a number of traversals over a graph (i.e., maximal forward references). However, the hypertext graph is not considered in [9].

Reference sequences are paths, but algorithms *FS* and *SS* examine arbitrary combinations of vertices during candidate generation, since they are similar to *DHP* algorithm for basket data mining and do not take into account the graph. For the determination of large reference sequences, the number of candidates for early phases (especially for the second) is much larger than when finding large itemsets because the ordering of vertices in the candidate is preserved. For example, if $\langle A \rangle$ and $\langle B \rangle$ are large singleton candidates, then they generate two candidates, $\langle A, B \rangle$ and $\langle B, A \rangle$, of length two. Of course, the hashing technique of *DHP* prunes a significant number of candidates, especially at the second phase. But still, as the graph structure is not considered, several candidates are generated that do not correspond to a path in the graph. In the previous example, if there is no arc between vertex A and vertex B, then candidate $\langle A, B \rangle$ can be pruned as it will be explained in the following sections.

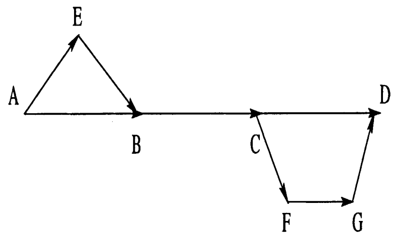


Fig. 2. A transaction corrupted by random accesses during navigation.

For a candidate of length k, apriori pruning and, consequently, the pruning of the hashing technique, cannot be based on all k subsets of length $k-1$. In [9], a candidate $P = \langle p_1, \ldots, p_{k+1}\rangle$ of length $k+1$, is generated by joining $S' = \langle p_1, \ldots, p_k\rangle$ and $S'' = \langle p_2, \ldots, p_{k+1}\rangle$, if both these sections of P, with length k, were found large. With respect to Lemma 1, P has only two sections of length k, i.e., S' and S''. In [9], it is not explained if any candidate pruning takes place, like in *DHP* where every subset is tested if it is large. Since S' and S'' are the only sections of P, any other combination of k vertices of P, even if it corresponds to a path in the graph, will not be supported by those transactions which support P. If a transaction T contained P and a combination of k vertices S''' from P, then S''' would be contained as a section in P, which contradicts with Lemma 1. Therefore, no further candidate pruning can take place.

During navigation, users usually follow one of the several patterns by accessing the vertices which belong to it. However, the accesses to the vertices of the pattern may not always be consecutive. Random accesses to vertices that are not part of the pattern may be done as well. Thus, inside a user traversal, patterns may be interleaved with random accesses. In Fig. 2, a transaction $T = \langle A, E, B, C, F, G, D\rangle$, is illustrated. If $P = \langle A, B, C, D\rangle$ is the pattern (maximal reference sequence), then, P is not a section of T. Thus, the support of P is not increased by transaction T. The random accesses of a traversal, i.e., those that are not part of a pattern, can be characterized as noise and the traversal which contains noise is called corrupted. If many transactions are corrupted, then, pattern P will not have adequate support and will be missed. Therefore, noise has an impact on the support of the patterns and their determination. In Section 3.1, we give a definition of *subpath containment* which, compared to section containment, can overcome the problem of identifying patterns from corrupted transactions.

2.2.3. Composite association rules

The approach in [7] differs from the previous ones because it considers the structure of the hypertext graph. User traversals are trails (traversals with distinct arcs) but patterns are not determined by counting frequencies of trails of arbitrary lengths. Only one database scan is performed and the number of occurrences of each arc is counted. By $fr(\langle A, B\rangle)$, we denote the normalized number of occurrences of arc $\langle A, B\rangle$, i.e., divided by the total number of arcs. In [7], the confidence of an arc $\langle A, B\rangle$ is defined as

$$conf(\langle A, B\rangle) = \frac{fr(\langle A, B\rangle)}{\sum_{\forall X \in G} fr(\langle A, X\rangle)}. \tag{1}$$

It is easy to notice that $conf(\langle A, B\rangle) = p(B\,|\,A)$. For a *trail* $T = \langle A_1, \ldots, A_n\rangle$, its probability of occurrence is

$$p(T) = p(A_1)p(A_2\,|\,A_1)p(A_3\,|\,A_2, A_1)\cdots p(A_n\,|\,A_{n-1}, \ldots, A_1). \tag{2}$$

By assuming the above procedure to be a *first-order markov chain*, we get

$$p(T) = p(A_1)p(A_2\,|\,A_1)p(A_3\,|\,A_2)\cdots p(A_n\,|\,A_{n-1}) \tag{3}$$

The confidence of a trail T is defined in [7] as

$$conf(T) = \prod_{i=1}^{n-1} conf(\langle A_i, A_{i+1}\rangle), \tag{4}$$

thus, $conf(T) = \prod_{i=1}^{n-1} p(A_{i+1}\,|\,A_i)$. Each trail T with $conf(T)$ larger than a user-defined parameter is called *composite association rule*. The confidence of a trail is not equal to its probability of occurrence because factor $p(A_1)$ of Eq. (3) is missing from the definition of confidence.

Two algorithms, *Modified DFS* and *Incremental Step*, are presented in [7], which find all composite association rules with a procedure based on a *Depth First Search* of the graph. Only one initial database scan is performed and the confidence of trails of length two is counted. These algorithms extent frequent trails with vertices that will result into trails with confidence larger than a user-defined parameter. Besides confidence, two metrics of support are defined. The first one is not monotonic and thus, it cannot be used for pruning. The second one defines the support of a trail $\langle A_1, \ldots, A_n\rangle$ as the $\min_{1 \leqslant i,j \leqslant n}\{fr(\langle A_i, A_j\rangle)\}$, which is monotonic. However, it is also not used for pruning because it tends to reduce the length of composite association rules due to over-pruning.

In the paradigm of mining association rules, as stated in [2] and adopted by the following work, patterns are identified by their probability of occurrence. These probabilities are the supports of the patterns and they are counted from the data and not estimated after the counting of frequent patterns of length two. Differently, in [7] the probability of each trail, called confidence (not support), is not counted in the database but is estimated from the counted frequencies of arcs, i.e., trails of length two. Since no further database scan is performed, this probability is an estimation based on the assumption of first-order markov property and is not verified in the database.

Moreover, the confidence of a rule $\{A_1, \ldots, A_n\} \Rightarrow \{B_1, \ldots, B_m\}$ (in the paradigm of association rules) is $p(B_1, \ldots, B_m\,|\,A_1, \ldots, A_n)$, which indicates correlation of order larger than one, as it is assumed by the first-order markov property. Actually, the objective of association rules is to identify correlation between items, which in general is of order larger than one.

2.3. Motivation

From the examination of the related work it becomes evident that for the problem of mining patterns from graph traversals it is required that:

- The graph structure has to be considered during candidate generation and pruning in order that generated candidates correspond to graph traversals. As described, this is in contrast to mining *sequential patterns* and mining association rules from *itemsets*, where patterns do not have such correspondence.
- The support of candidates should not be counted using the *section* containment criterion, as in the case of *maximal reference sequences*, since it is affected by noise. A different containment criterion is required, which will correctly count supports in the presence of noise.

- The support of patterns should not be just an estimate, as in the case of *composite association rules*, but it has to determine supports from the database contents.

In the following, we present the proposed method which tries to address the above requirements.

3. Mining access patterns from graph traversals

In this section, we give a formal definition of the problem of finding access patterns from graph traversals. As it is natural, and also assumed in other data mining problems, patterns are of the same type as the data. In case of basket data, which are sets of items, patterns are also sets of items (i.e., large itemsets). The same holds for sequential patterns. For this reason, we consider access patterns as traversals through the graph, since the data are also traversals. Access patterns which correspond to traversals indicate the actual way that navigation is performed in a graph structure by following links. For this type of patterns it is also required a definition of the *containment* criterion for the procedure of support counting, which is given in the following. Also, the *apriori-pruning* criterion is defined with respect to this type of patterns. In the sequel, we consider path traversals, although the proposed method can be generalized for trails or walks.

3.1. Subpath definition

The support counting requires the definition of a *containment* criterion of patterns in transactions. For the case of mining access patterns, we assume that both transactions and patterns are paths in a given graph. Thus, the criterion should preserve the ordering of accesses. Section containment, presented in Section 2.2.2, requires that the vertices (and the corresponding arcs) are consecutive in the path transaction. As described, this affects the determination of patterns in the presence of noise. Here, we give a generalization of the containment of a path inside another path.

Definition 3. We call subpath of a path $P = \langle p_1, \ldots, p_n \rangle$, the sequence $S = \langle s_1, \cdots, s_m \rangle$ for which

- S is also a path and
- there exist m integers $i_1 < \cdots < i_m$ such that $s_1 = p_{i_1}, \ldots, s_m = p_{i_m}$.

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 also preserved in the subpath. It is easy to show that the above definition determines a partial ordering for which the transitive property holds. Subpath criterion corresponds to that of subset. As in a subset, the vertices of a subpath are unique and contained in the path but, additionally, a subpath is a path, i.e., its vertices are connected with arcs and their order is preserved. Differently from the section criterion, vertices inside a subpath are not required to be consecutive in the path. In the example of Fig. 1, $\langle A,B,C \rangle$ is a subpath of $\langle A,B,D,C \rangle$. In Fig. 2, although pattern $P = \langle A,B,C,D \rangle$ is not a section of transaction $T = \langle A,E,B,C,F,G,D \rangle$, it is a subpath of T. Thus, noise does not affect the support of pattern P.

The following lemma shows that the notion of subpath is a generalization of that of section.

Lemma 2. *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 $(m \leqslant n)$, *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, there exists an integer $k \geqslant 0$ such that $p_{j+k} = s_j$ for all $1 \leqslant j \leqslant m$. Thus, regarding Definition 3, there exist m integers $k+1, k+2 \dots, k+m$ for which $s_j = p_{k+j}$ for $1 \leqslant j \leqslant m$. $\square$

The number of subpaths in a path is given by the following lemma.

Lemma 3. *A path $P = \langle p_1, \dots, p_n \rangle$ of length n has at least $n-k+1$ and at most $\binom{n}{k}$ subpaths of length k.*

Proof. If each pair of vertices p_i, p_j $(i < j)$ of P is connected with an arc, then, each combination S of k vertices from P is a path in the graph and the ordering of vertices in P is also preserved in S, therefore S is a subpath of P. Thus, there are at most $\binom{n}{k}$ subpaths. On the other hand, if no other pair p_i, p_j of vertices of P is connected with an arc, besides consecutive vertices p_i, p_{i+1}, then P contains at least $n-k+1$ sequences of consecutive vertices. Each of them is a section of P, and, thus, from Lemma 2, it is also a subpath of P. Thus, P contains at least $n-k+1$ subpaths. $\square$

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

Proof. It follows from Lemma 3, if $k = n-1$. The two subpaths that at least exist are $S' = \langle p_1, \dots, p_{n-1} \rangle$ (omitting the last vertex) and $S'' = \langle p_2, \dots, p_n \rangle$ (omitting the first vertex). These two subpaths are also the two sections of P (see Lemma 1). $\square$

3.2. Problem statement

Given a collection of transactions that are paths in a given graph, the problem of mining of access patterns involves finding all paths contained as subpaths in a fraction of transactions which is larger than *minSupport*. For a path P, this fraction is denoted as $fr(P)$, whereas following the notation of standard association rules, it is called *support* and all paths with support larger than *minSupport* are called *large*. 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. From Lemma 2 it follows that the set of all large paths with the subpath containment criterion is a superset of all large references sequences with the section containment criterion.

Fig. 3 illustrates an example with a database of five path transactions in the graph of Fig. 1, with *minSupport* equal to two. The generation of patterns, that are illustrated in the figure, will be explained in the following section.

3.3. Pruning with the support criterion

Apriori-pruning criterion [2] requires that a set of items $I = \{i_1, \dots, i_n\}$ is large only if every subset of I, of length $n-1$, is also large. In case of a path $P = \langle p_1, \dots, p_n \rangle$, pruning can be performed based on the following lemma.

Lemma 4. *A path $P = \langle p_1, \dots, p_n \rangle$ is large only if every subpath S of P, with length $n-1$ is also large.*

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	Supp
$\langle A\rangle$	5
$\langle B\rangle$	4
$\langle C\rangle$	5
$\langle D\rangle$	2
$\langle E\rangle$	1

L_1

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

C_2

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

C_3

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

L_3

Candidate	Supp
$\langle B,C,A\rangle$	2
$\langle D,C,A\rangle$	2

Fig. 3. Example of large path generation.

Proof. If path T is a transaction and P is a subpath of T, then S is also a subpath of T. Therefore, $fr(S) \geqslant fr(P)$ from which it follows that P is large only if every subpath S is large. Additionally, not every arbitrary combination S' of $n-1$ vertices, which is not a subpath of P, should be large. If S' is not a path in the graph, then $fr(S') = 0$, because it is not a subath of any transaction. If S' is a path in the graph but not a subpath of P, then it will not be supported by the same transactions. Thus, its support cannot determine if P is large. □

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 [9], only the two sections of the path are considered (see Lemma 1). For the determination of large paths, only the subpaths have to be tested, whose number, based on Corollary 1, may be less than $\binom{n}{n-1} = n$, as it is the case for itemsets. Therefore, no other combination of vertices except the subpaths should be examined, as described in the proof of the above lemma.

4. Level-wise determination of large paths

The determination of large paths can be performed in a level-wise manner, as in the Apriori algorithm [2]. In each pass of the algorithm, the database is scanned and the support of all candidates is counted. For each phase all candidates have the same length. At the end of each pass, all large candidates are determined and they are used for the computation of the candidates for the next pass. 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 in a graph G

```
 1. $C_1 \leftarrow$ the set of all paths of length 1, $\forall c \in C_1$ $c$.count $=$ 0
 2. $k = 1$
 3. while ($C_k \neq \emptyset$) {
 4.         for each path $p \in D$ {
 5.              $S = \{s \mid s \in C_k$, $s$ is subpath of $p\}$
 6.              for each $s \in S$ $s$.count++
 7.         }
 8.         $L_k = \{s \mid s \in C_k, s.\text{count} \geqslant minSupport\}$
 9.         $C_{k+1} \leftarrow$ genCandidates($L_k$, $G$), $\forall c \in C_{k+1}$ $c$.count $=$ 0
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) generating the candidates of the next phase (step 9)

differ significantly since the problem of determining large paths, as stated in Section 3.2, presents several differences compared to existing algorithms. Candidates have to be paths in the graph thus, the procedure of candidate generation (step 9) has to form only such candidates. It also has to perform apriori pruning based on Lemma 4. For support counting (steps 6 and 7), the subpath containment has to be performed based on Definition 2.

Additionally, although existing data structures like the *hash-tree* and *hash-tables* can be used for determining large paths [2,9], 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 recursive manner over the trie.

In the example of Fig. 3, notice that $\langle D, C, A\rangle$ is a large path and is a subpath of transactions with ID 2 and 4, although in transaction 4 its vertices are not consecutive. Also, notice that large path $\langle B, C, A\rangle$ cannot be further expanded because its terminating vertex A has only arcs to vertices B and C, which are already contained in the path. Large path $\langle D, C, A\rangle$ cannot be further expanded neither with vertex C, because it is already contained in the path, nor with vertex B, because path $\langle C, A, B\rangle$ was not found large.

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 [10,15], however, we assume that the graph size is such that the graph can fit in main memory. The reason is that for typical examples, like a graph representing the hypertext structure of a web site, the total number of 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 neighbourhood* of v. [3] This way, as it will be explained later, it is easy at any time to determine all extensions of a path from its terminating vertex. 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 [8,22] 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 is because the maximum fanout for a vertex v is equal to $|N^+(v)|$, which is the size of its positive neighborhood, and not to the total number of vertices. Thus, the trie occupies less space. After the step of counting the supports of candidates, the ones which were not found large are discarded from the trie but those found large remain in the trie to advocate the procedure of efficient generation of candidates for the next phase, as will be described in the following.

The trie has several advantages over the *hash-tree* data structure [2], which is used for fast counting of candidate itemsets supports. In the hash-tree, only leaves contain candidates, whereas in the trie every sequence of trie nodes, from the root to any node, represents a candidate path. This has an impact on the efficiency of support counting. Moreover, as it will be explained in the following, the trie grows dynamically as its leaves are extended and there is no need to build repeatedly a new hash-tree for every phase.

4.2. Support counting

The support counting for candidate paths (steps 5 and 6) is the most computationally intensive part of Algorithm 1. The support of candidates of length k is counted during the kth database scan. For each transaction T of length n, which is read from the database, all the possible subpaths of length k have to be determined (if $n < k$, then T is ignored). For this reason, T is decomposed to all its $\binom{n}{k}$ possible combinations of vertices (see Lemma 3) and each one is searched. For those which exist in the trie, i.e., they are subpaths of T, their support is increased by one. For example, if $T = \langle A, B, C, D \rangle$ 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 that 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), whereas 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 a graph can only determine if there exist a path connecting two vertices. Since the determination of the path existence would perform a search in the graph with cost $O(k)$ in the worst case, it follows that decomposing the

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

transaction to all possible combinations and searching the trie for all of them, does not require larger cost than $O(k \cdot [\binom{n}{k} - 2])$.

As mentioned earlier, the advantage of the trie over the hash-tree is that, since candidates in hash-tree are only at leaves, 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 [2]. On the other hand, in the case of a trie this operation requires k comparisons in 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

At each phase of the level-wise algorithm, 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 [2], these two steps are performed separately. First, at phase k, the set of all large candidates L_k is 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 (apriori pruning), 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, because 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. It is easy to show that it is equivalent to joining any fixed $k-1$ items, instead of the first $k-1$.

The generation of path candidates 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, unlike in Apriori, the first $k-1$ vertices may not be present in any other subpath besides the one of the two sections of C (see Corollary 1). Moreover, this joining in Apriori is performed to reduce the number of 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, only the extensions from the vertices in the positive neighborhood of the last vertex are considered for candidate paths. Therefore, their number is much less compared to the case of itemsets.

Conclusively, 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, if it does not belong to path L and subpath $L' = \langle \ell_2, \ldots, \ell_k, v \rangle$ is large, then 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. Next, 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 the trie {
    $N^+(\ell_k) = \{v \mid \text{there is an arc } \ell_k \to v \text{ in } G\}$
```

```
        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 $\ell_k$ with a branch to $v$
            }
        }
    }
```

Correctness. From Corollary 1, each $C = \langle \ell_1, \ldots, \ell_k, v \rangle$ has at least two subpaths, which are also sections of C. The first one is L itself and the other is $L' = \langle \ell_2, \ldots, \ell_k, v \rangle$ (by omitting the first vertex). If L' is not large, then neither C is large. Thus, L' has to be searched if it is large. L and L' are used to form candidate C. Also, with respect to Lemma 4, every other (if any) subpath of C, besides L', of length k has to be tested if it is large.

Additionally, candidate C has to be a path in graph G, otherwise it will have zero support, because no transaction will contain it as a subpath. Thus, it is sufficient to extent L with all vertices in the adjacency list of its last vertex ℓ_k, which are not already present in L (if a vertex is already present then C will not correspond to a path). Notice that the symmetrical case of extending L from the left with all vertices w, such that $C' = \langle w, \ell_1, \ldots, \ell_k \rangle$, is going to be handled when testing leaf $\langle w, \ell_1, \ldots, \ell_{k-1} \rangle$. If C' of length $k+1$ is large, then $\langle w, \ell_1, \ldots, \ell_{k-1} \rangle$, of length k, has to be large and thus, it will be encountered during the visit of large leaves at phase k.

All $k-2$ possible subpaths $S \neq L'$, of length k, of a candidate C have to be searched in the trie structure if they are large. Since every combination of vertices does not necessarily correspond to a subpath, it has to be determined only for actual subpaths if they are large. Therefore, 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 subpath. In case S is a subpath and it is not in the trie, then S is not large, thus, C is not large neither. Otherwise, if S is not a subpath, then it has to be ignored, since Lemma 4 holds only for subpaths. If $S = \langle s_1, \ldots, s_k \rangle$ and while descending the trie when searching for S, a 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 the arc does not exist, then S is not in the trie because it is not a subpath, since it contains a pair of vertices which are not connected, i.e., S does not correspond to a path. Otherwise, S is a subpath and is not present because vertex s_{i-1} was not expanded in a previous phase, as subpath $\langle s_1, \ldots, s_{i-1} \rangle$ was not large. In this case, 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 the 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. Non-level-wise algorithms for large path determination

The level-wise Algorithm 1 requires several database scans. In [2], an observation is made for the Apriori algorithm, which is also level-wise, that candidates of more phases can be generated and counted during the same scan. This approach reduces the number of database scans, but the total number of candidates increases, since apriori pruning is applied fewer times. Thus, this method pays off only during the last phases because the candidates are few and it does not worth to perform many database scans for them. Algorithm *Selective Scan*, proposed in [9], uses the same observation and merges phases of the level-wise algorithm to skip several database scans.

The reduction of multiple database scans has been addressed with partitioning [18] and sampling [20] methods. The former requires exactly two database scans. The latter requires one database scan in the best case but it does not have a guaranteed upper bound. However, experiments in [20] indicate that this method requires, at most, two and in many cases only one database scan. In the following, we focus on the partitioning algorithm because it presents a fix bound, although a similar approach could be followed for the sampling method. The partitioning algorithm has the following general structure. Initially, the database is divided into partitions which can be held in main memory. Then, all large patterns in each partition are found and are collected in *LC*. In a second step, all the members of *LC* are considered as candidates and with one additional database scan through *D*, the actual set *L* of large patterns is determined.

Partition Algorithm

```
1. Divide database $D$ to $p$ partitions $D_1, \dots, D_p$
2. for each partition $D_i$
3.         $LC$ += gen_large($D_i$)
4. $L$ = gen_large($D$) with $LC$ as the candidate set
```

Procedure *gen_large* finds all large patterns inside a partition by applying a level-wise algorithm. Of course, the level-wise algorithm operates on the partition which is contained in main memory, thus, it does not require any database scan. This approach is followed in [18], where Apriori algorithm is used for this purpose. [4] In case of mining access patterns, Algorithm 1 can be issued for the *gen_large* procedure. The partitioning algorithm does not use the method of phase merging during the application of *gen_large* procedure because each partition D_i resides in main memory and there is no need to reduce the number of scans.

Nevertheless, the phase merging approach does not only reduce the number of database scans but also reduces the computational cost of support counting. Since the supports of candidates with several lengths are counted together, candidates with the same prefixes are considered only once, in contrast to the level-wise algorithms, where each of them is counted separately in different phases. This method have been examined in [22], where candidates of several lengths, i.e., belonging to different phases of the level-wise algorithm, are generated. This method considers also main memory requirements and it uses the memory capacity as criterion to control the candidate generation. The experimental results indicate a significant reduction in computational cost for the support counting procedure.

[4] For the sampling algorithm [20], the level-wise Apriori algorithm is also used for the determination of large patterns.

In the case of mining access patterns, as explained, the generation of candidates is based on the graph structure and their number is constrained by this structure. For this reason, the generation of a larger number of candidates, required by the non-level-wise approach, can be used in the problem of mining access patterns. In the following, we present two methods for generating and counting candidates in a non-level-wise manner. We do not consider main memory issues, as in [22], because these methods do not present overwhelming space requirements, since the number of candidates is effectively controlled. The presented methods are used in the context of the *Partition* algorithm, as alternatives of using Algorithm 1 for the *gen_large* procedure. This way, the efficiency of the proposed methods is examined with respect to their computational cost for support counting, since they require the same number of database scans.

5.1. Phase-merging algorithm

Applying to Algorithm 1 the approach of phase-merging used by algorithm *Selective Scan* [9], we get Algorithm 2. Initially, all large paths of length one are found. Recall that these are the large paths for a database partition and that the actual large paths are found at the second (refinement) step of the *Partition* algorithm. At phase k, if the size of the candidate set C_k is small, candidates C_{k+1} of the next phase are also generated directly from C_k, by extending its members, without first finding large paths of length k. This procedure may be applied further more, for the remaining phases, until the number of candidates becomes too large.

The number of candidates is controlled by comparing the sizes of two successive candidate sets. If $|C_{k+1}| \leqslant |C_k|$, then support counting can be skipped. Otherwise, the support of all candidates is counted and the large ones are determined. This criterion is based on experimental results presented in [9] and is used as a trade-off between merging phases and generating too many candidates. In all cases, we assume that the generated candidates can be held in main memory. Algorithm 2 uses the same data structures as Algorithm 1 for candidate storing and support counting, thus, we omit any further details (which can be found in [9]).

5.2. Selective candidate extension

In Algorithm 2, either support counting for a phase is skipped or not, all the candidates of this phase are generated. This approach works in an all-or-none manner and it tends to produce a large number of candidates. A large number of candidates for the first step of the *Partition* algorithm may also lead to a large number of candidates at the second step. Instead of generating all candidates of a phase, if a selection could be made and only the ones which are expected to be large are generated, then the number of candidates and the computational cost for their generation and support counting could decrease.

The estimated support for a possible candidate path can be based on the support of its subpaths and more specifically, on the support of the subpaths of length two. Using Eq. (4), for a possible candidate $C = \langle v_1, \ldots, v_{k+1} \rangle$, its support can be estimated as $\widehat{fr}(C)$, where $\widehat{fr}(C) = \prod_{i=1}^{k} fr(\langle v_i, v_{i+1} \rangle)$. It is assumed that $k > 2$ and that the support of all paths of length two, i.e., arcs, has been counted. As an optimization step, all arcs which are not found large can be pruned from the graph because, following Lemma 4, no large path can contain an arc which is not large.

As in Algorithm 1, candidate generation is done with extension of large paths, i.e., by appending vertices at their end. Using the above estimation, instead of extending large paths with one vertice at a time in a level-wise manner, each path is extended to all its candidates, until a candidate which is not estimated to be large is encountered. After the extension of all large paths, we call *negative estimated border* Bd_{e}^{-} [20] the set of all candidates which have estimated support less than *minSupport*.

The algorithm for large path determination is called Algorithm 3 and is formed as following. Initially, all large paths of length one and two are found during the first two phases. In the sequel phases, each large path is extended until a candidate with estimated support less than *minSupport* is encountered. When no other large path can be extended any further, the support of candidates is counted. If a candidate is not found large, it is pruned. Candidates which are found large and do not belong to Bd_{e}^{-}, because they were estimated to be large, are inserted in the set of all large paths (this is the set of candidates for the second step of the *Partition* algorithm). But for candidates which were in Bd_{e}^{-} and were found large, we have to extend all this candidates again to find possibly large candidates which are their extensions, by applying the same procedure. The algorithm terminates when all candidates in Bd_{e}^{-} are found not large or when no more candidates can be generated. The algorithm is illustrated below. Procedure *genCandidates*3 generates candidates of Algorithm 3 as described above.

Algorithm 3. Non-level wise determination of large paths in a graph G

```
 1. $L \leftarrow$ the set of all large arcs
 2. **repeat**
 3.     $C \leftarrow$ genCandidates3($L$, $G$), $\forall c \in C$, $c$.count $= 0$
 4.     **for each** path $p \in D$ {
 5.         $S = \{s \mid s \in C$, $s$ is subpath of $p\}$
 6.         **for each** $s \in S$ $s$.count++
 7.     }
 8.     $L+ = \{s \mid s \in C, s.\mathrm{count} \geqslant minSupport\}$
 9.     make all large candidates in $Bd_{\mathrm{e}}^{-}$ extendible
10. **until** ($C == \emptyset$)
```

Algorithm 3, compared to Algorithm 1, generates and counts candidates of several lengths. Differently from Algorithm 2, it does not generate all candidates of a given phase but only those which are estimated to be large. This way, as it is verified by the experimental results, Algorithm 3, compared to Algorithm 2, generates less candidates. Of course, the efficiency of Algorithm 3 is based on the quality of support estimation. For those candidates which were estimated and were counted large, a separate support counting is skipped. Additionally, during the support counting, candidate with the same prefixes are examined once, instead of several times as in Algorithm 1. This reduces significantly the computational cost for support counting.

6. Performance results

This section contains the results of the experimental evaluation of all algorithms (implemented in C++) presented in the previous sections, using synthetic data. The experiments were run on a workstation with one Pentium II processor 450 MHz, 256 MB RAM, under Windows NT 4.0.

6.1. Synthetic data generator

First, the graph structure has to be generated. Two significant parameters are the total number N of vertices and the maximum number F of arcs leaving from of each vertex, called the *out-degree* or *fanout* of the vertex. We define two types of vertices: the ones with fanout larger than zero and the ones with fanout zero. The type of vertex is determined with a mixed distribution: Lognormal, for values less than a cut-off value and Pareto for larger ones. This way, as presented in [4,5], 93% of vertices have positive fanout. For each vertex of the first type, its fanout is determined following a uniform distribution between 1 and F.

Large paths are chosen from a collection of P paths representing the patterns. The length of each path pattern is determined with a Poisson distribution with average length L. A percentage of the vertices of the previous path is also kept in the current one. This percentage is called *correlation factor*. Besides the common vertices, the current path pattern is completed until it reaches the required length by choosing for the last vertex the link to follow. The next link is chosen randomly (i.e., with uniform distribution) from the set of all vertex links. Since this procedure may reach a vertex with no links or with links already contained in the path, a backtracking is performed until the specified length is achieved. If in some cases this cannot be achieved, then the path with the maximum length, from all those which were examined, is chosen.

Finally, transactions which represent the actual user traversals are formed. For each transaction, one of the P path patterns is chosen with a P-sided weighted coin. To represent the corruption of patterns with random accesses, we assume a corruption level C, which is the number of vertices from the path pattern that will be substituted with noise. This number follows Poisson distribution with average value C. Table 1 presents all the symbols used in the following.

6.2. Results

We examined the performance of Algorithm 1, Algorithm 2 and Algorithm 3. All algorithms, as explained, are used in the context of the *Partition* algorithm, for the implementation of the *gen_large* procedure. Therefore, all algorithms require the same number of database scans, i.e., two. For this reason, the comparison is done with respect to the computational cost for candidate generation and support counting. As a measure of the computational cost, we chose the number of integer operations required by this procedures. The computational cost is examined for both the two steps of *Partition* algorithm.

First, we tested the performance of all algorithms with respect to the *minSupport* value, given as percentage. The left part of Fig. 4 presents the results for $D = 100,000$ transactions, $P = 1000$ path patterns and $L = 7$ the length of a pattern. The graph has $N = 1000$ vertices and the

Table 1
Symbols representing the 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

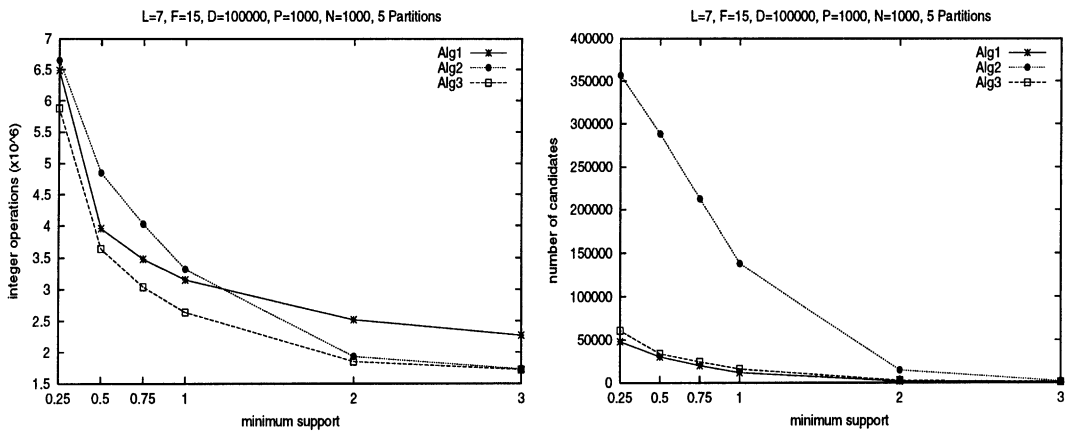


Fig. 4. (Left) Computational cost w.r.t. *minSupport*; (Right) number of candidates w.r.t. *minSupport*.

maximum fanout is $F = 15$. The number of partitions is 5. The corruption level is $C = 25\%$ and the correlation factor is 10%. We tested several values for the correlation factor, which did not result into significant differences in performance. For this reason, in the following, we maintain the correlation factor equal to 10%.

As it is illustrated, Algorithm 3 outperforms the other two algorithms. The reason is that it is not level-wise, as Algorithm 1. Thus, it counts in the same, and not in separate phases, candidates that have common prefixes. Algorithm 2 is also non-level wise, but it generates a large number of candidates. The right part of Fig. 4 presents the number of candidates for each algorithm with respect to the value of *minSupport* (as a percentage). As it is illustrated, Algorithm 3 generates marginally more candidates than Algorithm 1 but much less than Algorithm 2.

The number of partitions for the *Partition* algorithm is a result of the amount of available main memory, since each partition must be held in main memory. We assume that the available memory is at least 10% of the size of the database and, for this reason, we do not examine cases

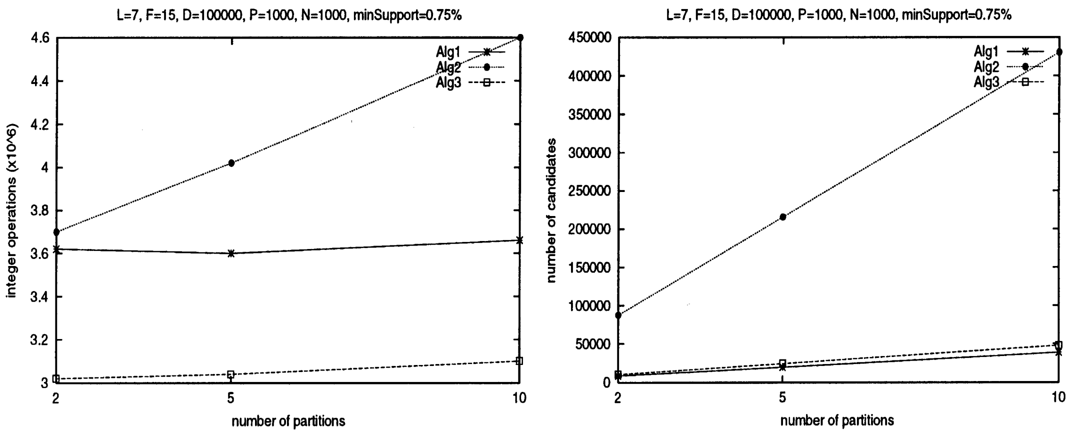


Fig. 5. (Left) Computational cost w.r.t. the number of partitions; (Right) number of candidates w.r.t. the number of partitions.

with more than 10 partitions. The left part of Fig. 5 illustrates the computational cost of all algorithms with respect to the number of partitions for *minSupport* equal to 0.75%. The other parameters are the same as in the previous experiment. As it is shown, only Algorithm 2 is affected significantly by the number of partitions and Algorithm 3 presents the best performance. The right part of Fig. 5 illustrates the number of candidates generated by each algorithm, with respect to the number of partitions, where it is shown that Algorithm 2 generates a large number of candidates. In the following experiments, we use five partitions, as this presents reasonable memory requirements.

Next, we examined the scale-up properties of the algorithms with respect to the number of transactions and the results are illustrated in Fig. 6, for *minSupport* equal to 0.75%. The other parameters are the same as in the previous experiments. It is shown that all algorithms present linear scale-up, but Algorithm 3 outperforms the other algorithms.

Then, we examined the performance of the algorithms with respect to the number of vertices *N*. The left part of Fig. 7 illustrates the results for *minSupport* equal to 0.75%, while the other parameters were the same as in the previous experiments. The computational cost reduces with respect to the number of vertices. The support of each vertex reduces when the total number of vertices increases and this results also into a reduction of the support of all paths and thus, to a reduction of the number of candidates. Again, Algorithm 3 outperforms the other ones.

The right part of Fig. 7 illustrates the results with respect to the pattern length *L*, for *minSupport* equal to 0.75%. The computational cost of all algorithms increases for patterns of larger lengths, since this result into an increase in the number of candidates that each transaction contains. As it is illustrated, Algorithm 2 is affected significantly by the increase in *L*.

Finally, we tested the impact of accesses which are not part of the patterns and are done for navigational purposes. These accesses are interleaved in transactions with accesses which belong to patterns. As explained, the *subpath* containment criterion can determine patterns in corrupted transactions. For this experiment, the corruption procedure interleaves between each two consecutive accesses of a pattern, a number of other accesses which follows a Lognormal distribution with mean value equal to two and standard deviation equal to one. The result forms transactions

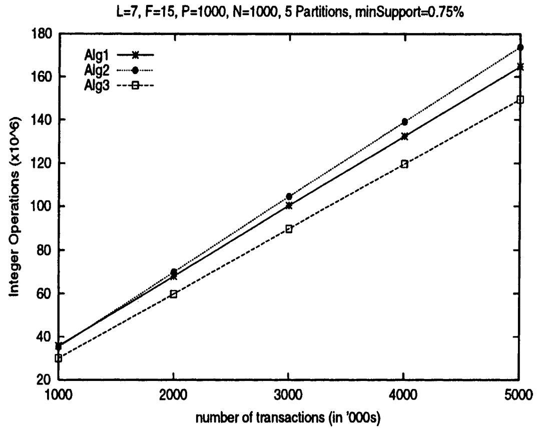
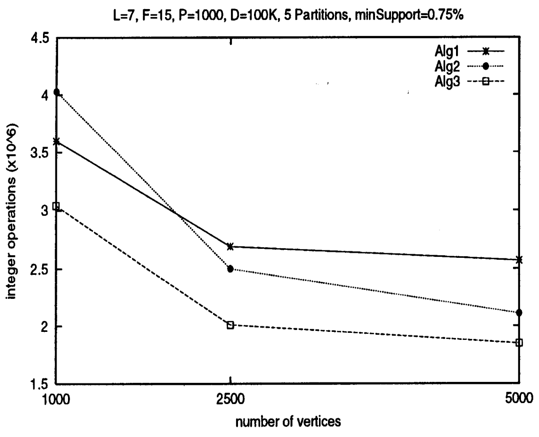


Fig. 6. Scale-up w.r.t. the number of transactions.



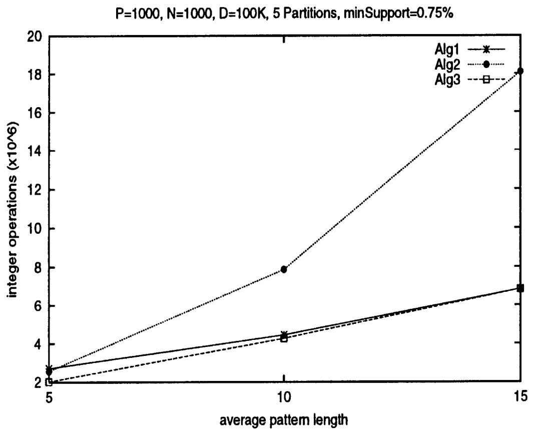


Fig. 7. (Left) Computational cost w.r.t. the number of vertices; (Right) computational cost w.r.t. the pattern length.

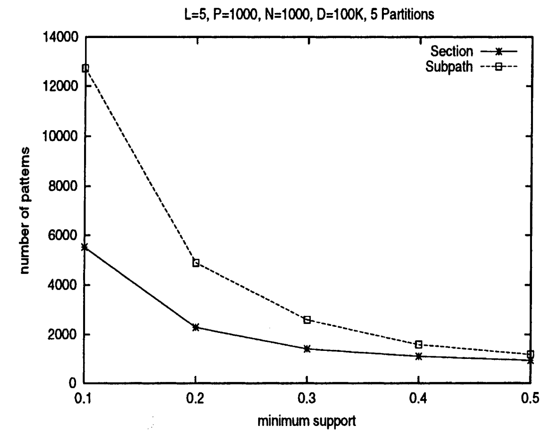


Fig. 8. Number of discovered patterns w.r.t containment criterion.

that are paths in the graph because interleaved accesses are selected in order that the result is a path. This type of corruption procedure is used because it clearly illustrates the impact of the interleaved accesses.

Fig. 8 presents the number of patterns discovered with the *subpath* criterion compared to the number of patterns discovered with the *section* criterion, with respect to the minimum support (as percentage). Recall that the *section* criterion requires that accesses inside patterns are consecutive. As it is illustrated, the number of patterns discovered with the *section* criterion is significantly less than the number of patterns discovered with the *subpath* criterion, especially for lower minimum support values. Thus, the corruption of patterns has larger impact when *section* criterion is used.

7. Conclusions

We examined the problem of determining patterns from graph traversals. We assumed that the traversals are produced from user access information in data models that have a graph representation. Differently from existing approaches, the proposed method considers both the database contents and the graph structure for the determination of patterns. Also, the patterns are defined in a way that takes into account the fact that irrelevant accesses, made for navigational purposes, may be interleaved with accesses which are part of the patterns. We presented a level-wise algorithm for pattern determination. This algorithm differs from the existing Apriori-like algorithms since it uses different data structures and methods for candidate generation and support counting. We also presented two non-level-wise methods. The first is based on an existing approach and the other one on a new approach of generating candidates with respect to their estimated support. The performance of all algorithms is tested experimentally with synthetic data.

The quality of the patterns can only be tested within a framework of a specific application. In the future, we plan to test the proposed patterns in the context of web data mining, for the purpose of web prefetching, i.e., the prediction of forthcoming web document requests of a user. This application requires the use of patterns for predictive purposes, thus, their quality will be tested with respect to their predictive accuracy. For such an application, the impact of noise will be tested more precisely, since it will impact the accuracy of prediction. In this context, the proposed approach in this paper, can be compared to other existing methods of web log mining.

Acknowledgements

Work supported by a national PABE project and an INTAS project (OPEN 97-11109).

References

[1] S. Abiteboul, Querying semi-structured data, in: Proceedings International Conference on Data Engineering (ICDE '97), 1997, pp. 1–18.
[2] R. Agrawal, R. Srikant, Fast Algorithms for mining association rules, in: Proceedings Very Large Data Bases Conference (VLDB '94), 1994, pp. 487–499.
[3] R. Agrawal, R. Srikant, Mining sequential patterns, in: Proceedings of the International Conference on Data Engineering (ICDE '95), 1995, pp. 3–14.
[4] M. Arlitt, C. Williamson, Internet web servers: workload characterization and performance, IEEE/ACM Transactions on Networking 5 (5) (1997).
[5] P. Barford, M. Crovell, Generating representative Web workloads for network and server performance evaluation, in: Proceedings of ACM Conference on Measurement and Modeling of Computer Systems (SIGMETRICS '98), 1998, pp. 151–160.
[6] B. Berendt, M. Spiliopoulou, Analysis of navigation behaviour in web sites integrating multiple information systems, VLDB Journal 9 (1) (2000) 56–75.
[7] J. Borges, M. Levene, Mining association rules in hypertext databases, in: Proceedings of the Conference on Knowledge Discovery and Data Mining (KDD '98), 1998, pp. 149–153.
[8] S. Brin, R. Motwani, J. Ullman, S. Tsur, Dynamic itemset counting and implication rules for market basket data, in: Proceedings of the Conference on ACM SIGMOD (SIGMOD '97), 1997, pp. 255–264.
[9] M.S. Chen, J.S. Park, P.S. Yu, Efficient data mining for path traversal patterns, IEEE Transactions on Knowledge and Data Engineering 10 (2) (1998) 209–221.
[10] Y. Chiang, M. Goodrich, E. Grove, R. Tamassia, D. Vengroff, J.S. Vitter, External-memory graph Algorithms, in: Proceedings of Symposium on Discrete Algorithms (SODA '95), 1995, pp. 139–149.
[11] R. Cooley, B. Mobasher, J. Srivastava, Data preparation for mining World Wide Web browsing patterns, Knowledge and Information Systems 1 (1) (1999) 5–32.
[12] K. Joshi, A. Joshi, Y. Yesha, R. Krishnapuram, Warehousing and mining Web logs, in: Proceedings of the Workshop on Web Information and Data Management, 1999, pp. 63–68.

[13] S. Madria, S. Bhowmick, W. Ng, P. Lim, Research issues in Web data mining, in: Proceedings of the Conference on Data Warehousing and Knowledge Discovery (DaWaK '99), 1999, pp. 303–312.
[14] A. Nanopoulos, Y. Manolopoulos, Finding generalized path patterns for Web log data mining, in: Proceedings of East-European Conference on Advanced Databases and Information Systems (ADBIS '00), 2000, pp. 215–228.
[15] M. Nodine, M. Goodrich, J.S. Vitter, Blocking for external graph searching, in: Proceedings of the Conference on ACM PODS (PODS '93), 1993, pp. 222–232.
[16] J.S. Park, M.S. Chen, P.S. Yu, Using a hash-based method with transaction trimming for mining association rules, IEEE Transactions on Knowledge and Data Engineering 9 (5) (1997) 813–825.
[17] J. Pei, J. Han, B. Mortazavi-Asl, H. Zhu, Mining access patterns efficiently from web logs, in: Proceedings of Pacific–Asia Conference on Knowledge Discovery and Data Mining (PAKDD '00), 2000.
[18] A. Savarese, E. Omiecinski, S. Navathe, An Efficient algorithm for mining association rules in large databases, in: Proceedings of the Conference on Very Large Data Bases (VLDB '95), 1995, pp. 432–444.
[19] M. Spiliopoulou, L. Faulstich, WUM: A tool for web utilization analysis, in: Extended version of Proceedings of EDBT Workshop (WebDB '98) (LNCS 1590), 1999.
[20] H. Toivonen, Sampling large databases for finding association rules, in: Proceedings of Very Large Data Bases Conference (VLDB '96), 1996, pp. 134–145.
[21] K. Wang, H. Liu, Discovering typical structures of documents: A road map approach, in: Proceedings of ACM SIGIR Conference (SIGIR '98), 1998, pp. 146–154.
[22] Y. Xiao, M. Dunham, Considering main memory in mining association rules, in: Proceedings of the Conference on Data Warehousing and Knowledge Discovery (DaWaK '99), 1999, pp. 209–218.
[23] O. Zaiane, M. Xin, J. Han, Discovering web access patterns and trends by applying OLAP and data mining technology on web logs, in: Proceedings on Advances in Digital Libraries (ADL '98), 1998, pp. 19–29.

Alexandros Nanopoulos received bachelor degree in Computer Science from the Aristotle Univ. of Thessaloniki (1996). He is a Ph.D. candidate at the Computer Science Department of Aristotle Univ. of Thessaloniki. His research interests include data mining, databases for web and spatial databases.

Yannis Manolopoulos received a B. Eng. (1981) in Electrical Eng. and a Ph.D. (1986) in Computer Eng., both from the Aristotle Univ. of Thessaloniki. He has been with the Department of Computer Science of the Univ. of Toronto, the Department of Computer Science of the Univ. of Maryland at College Park and the Department of Electrical and Computer Eng. of the Aristotle Univ. of Thessaloniki. Currently, he is Professor at the Department of Informatics of the latter university. He has published over 100 papers in refereed scientific journals and conference proceedings. He is co-author of a book on "Advanced Database Indexing" by Kluwer. He is also author of two textbooks on data/file structures, which are recommended in the vast majority of the computer science/engineering departments in Greece. His research interests include spatio-temporal databases, databases for web, data mining, data/file structures and algorithms, and performance evaluation of secondary and tertiary storage systems.