Available online at www.sciencedirect.com

SCIENCE DIRECT®

Information and Software Technology 47 (2005) 129–140

INFORMATION AND SOFTWARE TECHNOLOGY

www.elsevier.com/locate/infsof

Fast mining of frequent tree structures by hashing and indexing

Dimitrios Katsaros, Alexandros Nanopoulos, Yannis Manolopoulos*

Department of Informatics, Aristotle University, Thessaloniki 54124, Greece

Received 13 January 2003; revised 26 May 2004; accepted 28 June 2004
Available online 3 August 2004

Abstract

Hierarchical semistructured data arise frequently in the Web, or in biological information processing applications. Semistructured objects describing the same type of information have similar but not identical structure. Usually they share some common 'schema'. Finding the common schema of a collection of semistructured objects is a very important task and due to the huge amount of such data encountered, data mining techniques have been employed.

In this paper, we study the problem of discovering frequently occurring structures in semistructured objects using the notion of association rules. We identify that discovering the frequent structures in the early phases of the mining procedure is the dominant cost and we provide a fast algorithm addressing this issue. We present experimental results, which demonstrate the superiority of the proposed algorithm and also its efficiency in reducing dramatically the processing cost.
© 2004 Elsevier B.V. All rights reserved.

Keywords: Tree mining; Hashing; Semistructured data; Association rules

1. Introduction

The recent advances in networking and storage technologies enabled a tremendous growth in the amount of data that need to be retrieved from the network and subsequently be processed. Characteristic examples include the XML and HTML documents and the biological data. An intrinsic feature possessed by all these data is that they do not have a rigid, well-defined structure. There are several reasons for that. For instance, the data source may not impose any structure upon them, e.g. the Web data, genome sequences. The data may be extracted from various heterogeneous information sources, e.g. business-to-business product catalogs, where data from multiple suppliers (each with their own schema) must be integrated so that buyers can query them. The term that prevailed in order to characterize such kind of information is *semi-structured data* [2].

Although, semistructured data do not obey a strict structure, it is very usual to share some common substructures. Consider, for instance, a university's portal, which organizes the Web sites of its faculty members. Usually, these sites are constructed in such a way that all individual sites have some Web pages in common (e.g. biographical information page, projects' page, publications page, etc.), but each member is allowed to add other pages as well, like family info, hobbies pages, etc. Another motivating example can be retrieved from the application of Web usage mining [21]. Given a database of Web accesses, we can form sequences of visited links, which have, in general, the form of a graph. Since most of the time the users' interests have some overlap, we deduce that these graphs share some common substructures. Common substructures, in particular tree patterns, arise also in bioinformatics. Researchers have collected vast amounts of RNA structures, which are essentially trees. Whenever they are interested in getting some information about a newly sequenced RNA, they investigate whether it shares some common topological patterns with other known RNA structures.

Discovering frequently occurring or common (sub)structures in collections of semistructured data is a very

* Corresponding author. Tel.: +30-231-099-6363; fax: +30-231-099-6360.

E-mail addresses: dimitris@skyblue.csd.auth.gr (D. Katsaros), alex@skyblue.csd.auth.gr (A. Nanopoulos), manolopo@skyblue.csd.auth.gr (Y. Manolopoulos).

0950-5849/$ - see front matter © 2004 Elsevier B.V. All rights reserved.
doi:10.1016/j.infsof.2004.06.006

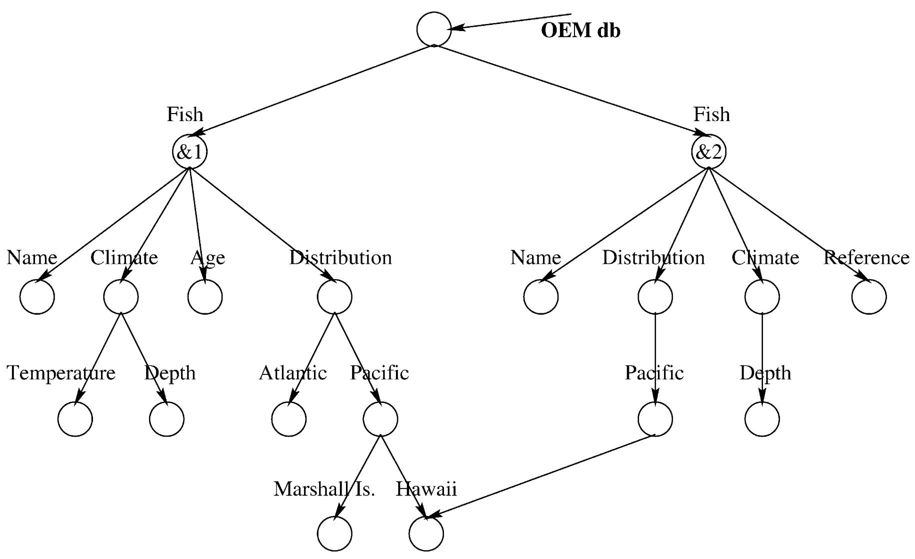

Fig. 1. A portion of 'fish' objects.

important task for numerous applications. It has applications on querying/browsing information sources [10], on building wrappers [29] and indexes [20], in storage for relational or object-oriented database systems [8], on query processing [19], on clustering documents based on their common structure, and on caching [33].

Semistructured data are usually modeled by a graph [24] or a tree [29], where a data object is represented by a node and a connection or relationship between objects is encoded by an edge between them. For the sake of convenience, we illustrate a small example of semistructured objects in Fig. 1, which is retrieved from the 'Catalogue of Life' site (located at http://www.sp2000.org). The example shows a portion of semistructured 'fish' objects.

In this work, we are interested in treating these data as *rooted*, *ordered* and *node-labeled* trees, and study the problem of discovering all frequent tree-like patterns that appear a minimum number of times in a given collection of semistructured data.

1.1. Schema discovery for semistructured data

Initial works on structural pattern discovery in collections of semistructured objects are described in Refs. [10,22,23,30]. The major drawback of these methods is that they are inherently unscalable, which is a serious problem due to the huge volume of semistructured data and due to their irregularity.

To deal with the huge volume of data, the exploitation of data mining methodologies offered a robust solution. Numerous algorithms were proposed to deal with the problem of discovering frequent tree or graph (sub)structures. The discovery of common substructures in graph-based data (*graph mining*) is investigated in the following works [13–16,18,28,32], whereas research work on *tree mining* is described in Refs. [1,5–7,29,31,34]. As pointed out in Ref. [34], graph mining algorithms are likely to be too general for tree mining. Considering the works, which address tree mining, some of them deal with unordered trees [6,31], and the rest with ordered trees [1,5,7,29,34], which is the focus of this paper.

We can classify the works that address the problem of ordered tree mining in two generic categories. The first includes the algorithm of Wang et al. [7,29]. This method is based on the original Apriori association rule mining paradigm and implements a generate-and-test strategy. The second category includes the works that devised an incremental algorithm that simultaneously constructs the set of frequent patterns and their occurrences level by level. Examples of this category are the TREEMINER [34] and the FREQT algorithm with its variations [1,5].

In this paper, we focus on the work of Wang et al. [7,29], which considers mining collections of paths in ordered trees with Apriori-style [3] techniques. We will refer to their method as the *WL* algorithm. Our interest stems from the special kind of patterns it discovers, that is, collections of paths.

1.2. Motivation

The *WL* algorithm is a multipass method, similar in nature to the Apriori [3]. *WL* first determines the frequent path expressions, which are sequences of node labels. The path expressions are called 1-*tree-expressions*, because they contain only one 'leaf', if we consider that every path expression is a tree, in which every node has only one child. Then, *WL* makes several iterations; at the kth ($k \geq 1$) iteration, it constructs a set of candidate $(k+1)$-tree-expressions using the frequent k-sequences and applying some pruning criteria. Each $(k+1)$-tree-expression is a tree, which contains exactly $k+1$ leaves. This set is a superset of the actual frequent $(k+1)$-tree-expressions. Then, it determines the support of the candidates by scanning over the database of transaction objects. We can easily see the similarities between *WL* and Apriori, if we consider that each 1-tree-expression (i.e. path expression) corresponds to a plain item of the Apriori, and each k-tree-expression to an Apriori's plain itemset of length k.

Though, the involvement of trees in *WL*—instead of Apriori's plain sets—introduces several challenges. Firstly, we need a novel criterion to test 'containment' of tree-expressions in the context of *WL*; Apriori dealt with it using the usual 'set containment' criterion. *WL* dealt with it providing the 'weaker than' relation (cf. Section 2), which is essentially a subtree testing algorithm. Secondly, we need a data structure to store the candidate tree-expressions, which will facilitate the 'containment' testing and the generation of new candidate tree-expressions; Apriori dealt with it using the hash-tree. *WL* dealt with it providing the Π_k tree, a condensed data structure (cf. Section 2). Thirdly, we need to address the problem related to the *combinatorial explosion* in the number of candidate itemset of length 2; Apriori dealt with it providing a solution based on hashing [25], which substantially reduces the number of itemsets of length 2, that must be tested for containment in the database. *WL* though, does not provide any mechanism for dealing with the *combinatorial explosion in the number of candidate 2-tree-expressions*. As we prove in Section 5, this issue is present in the context of *WL* and it is very important, because the 'weaker than' testing is much more time-consuming, since it involves subtree testing. In the next paragraphs, we describe in more detail the aspects of this defficiency of *WL*.

The execution cost of *WL* is determined by the number of database scans (I/O cost) and the number of candidate tree-expressions generated in each phase (CPU cost). Although *WL* follows a multiphase approach which requires several database scans (analogous to the Apriori paradigm), it has to be noticed that the I/O cost can be drastically reduced by a straightforward application of other multiphase approaches, like the ones described in Refs. [26,27], which will require up to two scans (or one at the best case for Ref. [27]).[1] However, regardless of the number of scans, the CPU cost can be significantly large, especially for low support values, as it is the case for mining association rules from itemsets [25]. Moreover, in the case of discovering frequent tree-expressions the CPU cost is increased due to two reasons:

(a) it involves the costly operation of tree matching required by the 'weaker than' criterion, and
(b) the number of candidates can increase very rapidly as a result of the large number of possible combinations due to the ordering and structuring of labels (differently from plain itemsets).

For the reduction of CPU cost at each phase k, the generated candidate k-sequences should have high probability of being frequent and also to avoid generating k-sequences that represent the same k-tree-expressions. The former requirement is addressed by combining only frequent $(k-1)$-sequences (*downward closure* or *anti-monotonicity property* [29]) and the latter by applying some pruning strategies introduced in Ref. [29, p. 361–362].

In the first iteration, we must construct the set of candidate 2-tree-expressions. In general, the cardinality of this set would be $L_1 \times (L_1 - 1)/2$, where L_1 is the cardinality of the set of frequent one-tree-expressions (path expressions). The *anti-monotonicity*, which states the only combinations of frequent k-tree-expressions can produce frequent $(k+1)$-tree-expressions, is not able to prune effectively the number of candidates in this early phase (see also Ref. [3]). Thus, the number of candidate 2-tree-expressions remains very large compared with that for later phases (for $k > 2$). Our experiments (see Fig. 3) attest that the cardinality of the set of candidate 2-tree-expressions is as much as two orders of magnitude larger than the respective cardinality for k-tree-expressions ($k > 2$). In *WL*, candidates are indexed with a trie structure (Π_k trie) which requires a large number of tree matching operations, especially during the second phase when the number of candidates is large (see also Section 2.2). The breakdown of the processing cost of each phase (see Section 5.1.2) shows that *the second phase corresponds to a performance bottleneck for the WL algorithm*.

The above observations come in accordance with analogous ones for the problem of discovering frequent (large) itemsets from basket data [3]. In that case, it was found that support counting for candidates in the early phases is the dominant processing cost, since the principle of anti-monotonicity could not help in pruning many of them. Algorithm *DHP* proposed in Ref. [25], during the kth pass (usually $k=1$ or 2), estimates the support of $(k+1)$-large itemsets so as to avoid generating too many candidates in the next phase. The estimation is derived by hashing during the kth phase, all the $(k+1)$-itemsets contained in each transaction so as to estimate their support during the $(k+1)$th phase by the size (number of entries) of the corresponding hash bucket. Another technique employed by *DHP* is the database 'trimming', i.e. the reduction in the number of transactions to be scanned and reduction in the number of items in each transactions, if it can be identified that some transactions or some items cannot contribute to the support of any candidate. Though, the issue of database trimming is orthogonal to our work and could be effectively combined with our technique.

However, the direct application of hashing pairs of 1-tree-expressions, contained in each transaction, into the same bucket of a hash table, in a way analogous to *DHP*, is not efficient in the case of 2-tree-expressions, because there are pairs of path-expressions that will never be combined to generate a candidate. Moreover, since *DHP* was proposed for itemsets, the hashing of 2-itemsets can be easily

[1] For reasons of fair comparison, we adopt the Apriori-like paradigm, similar to *WL*.

performed (e.g. by applying a function on the two items). In contrast, tree-expressions have structure and ordering, which have to be taken into account.

Finally, some recent works which proposed methods that avoid generating candidates, like the FP-tree [12], cannot be adapted in a straightforward manner to this problem, because of the structural properties of the tree-expressions.

Therefore, a new method is required that will consider the particular problems of the discovery of frequent tree-expressions and that will address the overhead presented during the early phase of the process, i.e. during the phase of the testing of 2-tree-expressions. Therefore, we seek for an indexing scheme for the space of candidate 2-tree-expressions, in order to discover very fast the set of candidates that satisfy the ‘weaker than’ relationship without the need to examine this costly relationship of tree matching between almost every pair of a candidate and a transaction (due to non-effective pruning).

1.3. Contributions

The focus of this paper is on the problem of how to efficiently index the search space of the candidate 2-tree-expressions. We are seeking for a hashing scheme for tree-expressions that takes into account the structural properties of each candidate, i.e. labels, ordering and nesting.

The present work makes the following contributions. Firstly, it identifies that support counting for 2-tree-expressions is the major performance bottleneck for the *WL* algorithm. Moreover, it identifies that the number of candidate 2-tree-expressions is the factor responsible for the running-time behavior of the algorithm.

Then, the paper describes an efficient hashing scheme for ordered labeled trees that can help to avoid repetitively performing a tree-matching algorithm. Finally, it presents experimental results for the proposed method in order to evaluate its efficiency.

The rest of this paper is organized as follows: Section 2 gives a comprehensive overview of the *WL* algorithm and a formal description of the problem we are addressing. Section 3 describes a hashing scheme for ordered labeled trees, which is an efficient solution for our problem and Section 4 presents the complete algorithm for the efficient discovery of structural associations between semistructured data. Finally, Section 5 presents the experimental results and Section 6 concludes with a summary of the paper and its contributions, and some issues that future work could address.

2. Preliminaries

2.1. Overview of WL algorithm

For our convenience we recall some definitions and give a comprehensive description of the *WL* algorithm [29].

The data model employed for the representation of semistructured data is that of labeled directed graphs. We adopt the *Object Exchange Model* [24] for the representation of semistructured objects. In this model, each object is identified by a unique identifier $\&a$ and its value val($\&a$). Its value may be either *atomic* (e.g. integer, string, float), a list $\langle l_1{:}\&a_1, l_2{:}\&a_2, \ldots, l_n{:}\&a_n\rangle$ or a bag $\langle l_1{:}\&a_1, l_2{:}\&a_2, \ldots, l_n{:}\&a_n\rangle$.[2] For the schema discovery problem (to be defined shortly after) the user must specify some objects, called *transaction objects* and denoted as $\top$, whose common structure we are interested in identifying (e.g. in Fig. 1 the target objects are the fish objects (&1) and (&2)).

Definition 1 (Tree expressions [29]). Consider an acyclic OEM graph. For any label l, let l^* denote either l or the wild card label ?, which matches any label.

1. The nil structure $\perp$ (that denotes containment of no label at all) is a *tree-expression*.
2. Suppose that te_i are tree expressions of objects a_i, $1 \leq i \leq p$. If val($\&a$)$=\langle l_1{:}\&a_1, l_2{:}\&a_2, \ldots, l_p{:}\&a_p\rangle$ and $\langle i_1, i_2, \ldots, i_q\rangle$ is a subsequence of $\langle 1, \ldots, p\rangle$ $q>0$, then $\langle l^*_{i_1} : te_{i_1}, \ldots, l^*_{i_q} : te_{i_q}\rangle$ is a *tree-expression* of object a.

Therefore, a tree expression represents a partial structure of the corresponding object, since it can ignore some references of the object and can stop at any level [29]. A *k-tree-expression* is a tree-expression containing exactly k leaf nodes. Each leaf node corresponds to a label path emanating from a transaction object. Hence, a 1-tree-expression is the familiar notion of a *path expression*, that is, a sequence of labels. Each k-tree-expression can be constructed by a sequence of k paths ($p_1, p_2, \ldots, p_k$), called *k-sequence*, where no p_i is a prefix of another. In order to account for the fact that some children have repeating outgoing labels, *WL* introduced superscripts for these labels. Hence, for each label l in val($\&a$), l^i represents the ith occurrence of label l in val($\&a$). Consequently, a k-tree-expression can be constructed by a k-sequence ($p_1, p_2, \ldots, p_k$), each p_i of the form $[\top, l^1_{j_1}, \ldots, l^n_{j_n}, \perp]$. Although Definition 1 holds for acyclic graphs, in Ref. [29] cyclic OEM graphs are mapped to acyclic ones by treating each reference to an ancestor (that creates the cycle) as a reference to a terminating leaf node. In this case, the leaf obtains a label, which corresponds to the distance of the leaf from its ancestor (i.e. the number of intermediate nodes). For this reason, henceforth, we consider only acyclic OEM graphs. Additionally, *WL* replicates each node that has more than one ancestor. The result of the above transformation is that each object is equivalently represented by a tree structure.

[2] Order does matter in a list but it does not in a bag. Label repetition is allowed both in a bag and a list node. In tile present paper we deal only with nodes of list type, since our target is ordered semistructured data (e.g. XML).

Definition 2 (Weaker than [29]). The nil structure $\perp$ is *weaker than* every tree-expression.

1. Tree-expression $\langle l_1{:}te_1, l_2{:}te_2, \ldots, l_n{:}te_n\rangle$ is weaker than tree-expression $\langle l'_1 : te'_1, l'_2 : te'_2, \cdots, l'_m : te'_m\rangle$ if for $1 \le i \le n$, te_i is weaker than some te'_{ji}, where either $l'_{ji} = l_i$ or $l_i = ?$ and $\langle j_1, j_2, \ldots, j_n\rangle$ is a subsequence of $\langle 1, 2, \ldots, m\rangle$.
2. Tree-expression *te* is weaker than identifier *&a* if *te* is weaker than val(*&a*).

This definition captures the fact that a tree-expression te_1 is weaker than another tree-expression te_2 if all information regarding labels, ordering and nesting present in te_1 is also present in te_2. Intuitively, by considering the paradigm of association rules [3], the notion of tree expression (Definition 1) is analogous of the *itemset* of a transaction and the notion of weaker than relationship (Definition 2) corresponds to the containment of an itemset by a transaction (or by another itemset). Wang and Liu in Ref. [29] introduced the use of wild card label ? in order to discover more general patterns. However, the label ? can produce non-meaningful patterns, because tree-expressions containing many ? tend to become frequent dominating the result. Moreover, as mentioned in Ref. [29], it increases the complexity of the problem. Thus, we focus on the paradigm of Ref. [3] where itemsets are drawn from a specified domain where no wild cards exist. The consideration of this topic is a subject of future work, in light of the results produced in Ref. [9].

Definition 3 (Discovery problem). Consider a collection of transaction objects in an OEM graph and a user specified minimum support MINSUP. The support of a tree-expression *te* is the percentage of transaction objects *t* such that *te* is weaker than *t*. We denote that *t* is *frequent* (also called large) if its support is above MINSUP. The discovery problem is to find all frequent tree-expressions.
WL first determines the frequent path expressions, that is, frequent 1-tree-expressions. Then it makes several iterations. At the *k*th ($k \ge 1$) iteration *WL* constructs a set of candidate $(k+1)$-tree-expressions using the frequent *k*-sequences and applying some pruning criteria. This set is a superset of the actual frequent $(k+1)$-tree-expressions. Then, it determines the support of the candidates by scanning over the database of transaction objects. The number of generated candidates is crucial to performance, since it involves the costly to compute *weaker than* relationship between tree-expressions, i.e. between pairs of candidate and transaction objects.

2.2. Problem description

The structure employed by *WL* for indexing the candidate tree-expressions of the *k*th phase is called $(k-1)$-*candidate-trie* (or Π_{k-1}). It facilitates efficient retrieval of pairs of $(k-1)$-sequences $(p_1, \ldots, p_{k-2}, p_{k-1})$ and $(p_1, \ldots, p_{k-2}, p_k)$, and the dynamic growth from F_{k-1} to F_k. Each sequence is mapped to an ID number, and each node of Π_{k-1} contains the ID of the corresponding sequence so as to minimize the storage requirements. During support counting, Π_{k-1} attains some pruning based on the *downward closure* property. This is because, while descending Π_{k-1} in a top down fashion, we are able to deduce that a candidate *k*-sequence is not weaker than a transaction object, if at least one of its constituent *j*-sequences, $1 \le j \le k$ is not weaker than the transaction object.[3] Nevertheless, as mentioned in Section 2.1, for the candidate 2-tree-expressions little pruning can be accomplished based on this property, because for every candidate 2-tree-expression, its first constituting 1-tree-expression is present in too many transaction objects. Whenever Π_{k-1} cannot prune the searching of a node, a tree matching operation is performed to test the weaker than relationship between the transaction and the corresponding candidate tree-expression that the examined sequences, up to this node, comprise. Thus, Π_{k-1} cannot reduce the processing cost for the support counting of candidate 2-tree-expressions, which is the most requiring step (with respect to the execution time) of the discovery of frequent tree-expressions.

Therefore, our objective is to have an indexing scheme for the space of candidate 2-tree-expressions, in order to discover very fast the set of candidates that satisfy the weaker than relationship without the need to examine this costly relationship of tree matching between almost every pair of a candidate and a transaction (due to non-effective pruning). Therefore, access to the leaves of the trie is required in such a selective manner, that for every transaction object we could locate those candidates contained in it. In other words, we would like to 'break-breakdown' the transaction into its component 2-tree-expressions and then increase their support, if they exist as candidates. Therefore, we can formally define the examined problem as follows:

Definition 4 (Containment searching problem). Let *C* be a set of trees with exactly two leaf nodes and with labels on edges drawn from a set of character strings. A member of this set will be called *pattern tree*. Let *T* be another set of trees with labels on edges drawn from the same set of character strings. Each member of this set may have from 1 to as many leaf nodes. A member of this set will be called *subject tree*. A pattern tree *c* is said to 'be contained' into a subject tree *t* if and only if it is 'weaker than' *t*, with list semantics. The problem is for each $t \in T$ to locate all $c \in C$ that are contained in it.

Our concern is to have a scheme, which incurs only a small computational overhead while dealing with the candidate discovery procedure. We would like our scheme

[3] It has to be noticed that Π_{k-1} is different from tile candidate-trie used for the case of plain itemsets, due to tile weaker than relationship used for the discovery of frequent tree expressions.

to avoid tree matchings and rely upon number comparisons. Additionally, to take into account the special characteristics of our problem, which are the nested structure of the 2-tree-expressions, the ordering between sibling nodes of a candidate, and the absence of regular expressions in our pattern trees. In other words, we seek for *a hashing scheme for ordered labeled trees*.

3. The hashing scheme for labeled trees

Katzenelson et al. [17] described an encoding scheme for solving the problem of type matching, i.e. the search of identical data types in programming languages. Data types are modeled with rooted labeled graphs, which are called *type-graphs*. The scheme initially maps type-graphs into multivariable polynomials, then each variable is assigned a number from the sequence of the logarithms of prime numbers. With this assignment, according to the Schanuel conjecture [17], the resulting value of the polynomial uniquely identifies the type-graph and is called *magic number*. The CMP algorithm is proposed in Ref. [17] for the calculation of magic numbers. CMP addresses the existence of back edges in the graph structure (i.e. edges of the type $v \rightarrow u$, where u is an ancestor of v) and the possibility of a node to have more than one ancestors. More details can be found in Ref. [17].

Ordered labeled trees examined in the present work are a special kind of type-graphs. An encoding scheme for ordered labeled trees can exploit the characteristics of magic numbers, which consider the ordering between sibling nodes and the nesting of edges. Nevertheless, the specialties of the labeled tree structure, compared to type-graph, and the requirement for a hashing scheme with low computational overhead (due to large database size in contrast to a programming language environment) call for a modified algorithm. The algorithm should produce identical results to those of CMP. However, it has to be more efficient than CMP for the case of ordered labeled trees. Thus, we develop a new algorithm based on the original CMP, which takes into account the special properties of our targeted structures, i.e. ordered, labeled trees. We first present the proposed algorithm for encoding labeled trees with magic numbers and then we describe the hashing scheme.

3.1. Encoding algorithm

Let T be a labeled tree. For each vertex v of T we denote the following:

- v.e: the label of the node v; recall that we deal with node-labeled trees. We assume that there exists a mapping of labels into the set of positive integers.
- v.S: the scan number assigned by an ordered depth first search tree traversal (i.e. children of a node are visited according to their order).
- v.l: the level of v (root is at level one and each child of a v is at level v.l + 1)
- v.O(u): (where u is a child of v) the index of u among the sequence of v's children.
- v.M: the magic number corresponding to the sub-tree rooted at v.

For the calculation of magic numbers we use the TLP variable, which is a table containing the sequence of logarithms of prime numbers (precomputed). Based on the above, the proposed algorithm, called Labeled Tree Encoding (*LTE*), is presented in Fig. 2.

LTE algorithm traverses the labeled tree in an depth-first order. The initial call of *LTE* (not shown in Fig. 2) commences from the root vertex. Compared to CMP, *LTE* follows a different approach. CMP separately produces the multivariate polynomial and then, in a second stage, it calculates the magic number. *LTE* calculates at each node the magic number and then propagates it to the parent node, up to the root. The magic number of the root corresponds to the magic number of the entire labeled tree. This approach is more efficient than maintaining a multivariate polynomial (using specialized data structures) during the tree traversal and calculating the magic number in a second stage. Also, differently from CMP, the depth-first traversal of *LTE* naturally corresponds to the structure of labeled trees.

Considering a vertex v during the traversal, step 1 of *LTE* initializes the v.M value to $v.e + v.S \times TLP$ [1]. Thus, both the fact that the tree is labeled and the nesting of vertices (through their scan number) are taken into account. The scan

```
Procedure LTE(TreeNode $v$, Array TLP)
begin
1. $v$.M = $v$.e + $v$.S × TLP[1];
2. if not ($v$ is leaf node)
3.     forall $u$ children of $v$
4.         $u$.M = LTE($u$, TLP);
5.         $u$.M = $u$.M × TLP[$(v.l + v.O(u))(v.l + v.O(u) - 1)/2 - v.O(u) + 3$];
6.         $v$.M = $v$.M + $u$.M;
7.     endfor
8. endif
9. return $v$.M;
end
```

Fig. 2. Labeled tree encoding algorithm.

number v.S is multiplied by TLP [1] based on the approach of CMP, because in the later the scan number is multiplied by the variable that is assigned to the logarithm of the first prime. If v is a leaf node (step 2), the traversal stops and the magic number v.M is equal to value calculated at step 1. Otherwise, the traversal continues to all children of v, at step 3 (children are visited according to their order in v, hence the ordered depth-first traversal). Having calculated the u.M of a child u, it is multiplied by the $(v.l+v.\mathrm{O}(u))(v.l+v.\mathrm{O}(u)-1)/2-v.\mathrm{O}(u)+3$ element of TLP (step 5). This is based on CMP algorithm, since this is the TLP value assigned to the corresponding variable in the polynomial. Finally, all u.M values are added to v.M (step 6). Therefore, the ordering of v's children is taken into account, because each u.M is assigned to a different TLP value.

3.2. Hashing scheme

The magic number of a multivariate polynomial that results after assigning to its variables values from the sequence of logarithms of prime numbers, can be considered as an irrational number. The Schanuel conjecture for the case of type-graphs, ensures no collisions, i.e. no two different type-graphs have the same magic number [17]. Therefore, the same can be stated for the case of *LTE* and ordered labeled trees. However, by using floating-point arithmetic (e.g. 64 bit double precision float numbers) instead of irrational ones, the resulting magic number is an approximation, which may produce collisions. Experimental results in Ref. [17], however, illustrate that for the case of type-graphs with one million instances (with heights in the range 5–20 and maximum number of a node's children equal to 10), no collisions were produced. Thus, the approximation of magic numbers with float numbers is a hashing scheme, mapping irrational magic numbers to floating-point ones that presents good quality in terms of collisions. Moreover, it has to be guaranteed that a magic number can fit within the limits of the specified floating-point arithmetic. For the latter issue more details can be found in Ref. [17], where normalization techniques are proposed.

For large databases of ordered labeled trees, the collection of magic numbers have to be maintained in a probe structure, so that their matching can be examined efficiently. We store the magic numbers in a hash table, the size of which is determined by the size of available main memory. However, due to the possibility of collisions resulted by the floating-point arithmetic, the matching of two magic numbers is verified by the actual matching of the corresponding labeled trees, which is examined by applying a tree-matching algorithm (based on the tree-matching algorithm used in Ref. [29]). The whole procedure is exemplified below:

Example 1. Let three labeled trees, t_1, t_2 and t_3, be depicted in Fig. 3a. Upon the incoming edge of each node, the corresponding label is drawn (assuming a mapping of labels

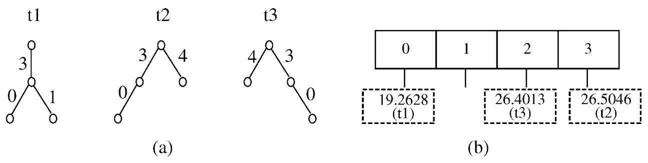

Fig. 3. (a) An example of three labeled trees. (b) The corresponding probe structure.

to integer numbers). By applying *LTE* on this collection of labeled trees, we get the resulting magic numbers $m_1=19.2628$, $m_2=26.5046$ and $m_3=26.4013$, respectively. It has to be noticed that although t_2 and t_3 consist of the same set of labels (i.e. {0, 3,4}), due to different structure and ordering, m_2 is different from m_3. The magic numbers are stored in hashing structure, depicted in Fig. 3b. Assuming a query tree q that is identical to t_2, we first apply *LTE* on q. Then, we probe the hash structure with the resulting magic number and we get the entries in position 3 of the structure as candidates. Since only m_2 matches the query, we perform a tree matching operation between q and t_2 in order to verify the actual matching of the labeled trees.

4. The complete mining algorithm

The complete mining algorithm for the discovery of structural associations consists of two different stages. The first one is the discovery of large path expressions (called 1-tree-expressions, which are trivial sequences), denoted as F_1. This stage does not involve nesting and ordering. Therefore, traditional techniques [4] can be applied. A straightforward method was adopted in Ref. [29], which selects all path-expressions and counts their support in a single pass. For purposes of fair comparison, we also follow this approach, although other optimizations can be applied (e.g. Web 1og mining algorithms).

The second stage, including phases F_k, $k\leq 2$, introduces the new *Containment searching problem* (Definition 4), involving tree-expressions. During the second-phase F_2 of the complete algorithm, 2-tree-expressions are examined. Since the number of candidates in this phase corresponds to the bottleneck of the overall procedure, we apply the labeled tree encoding algorithm and the representation with the hash structure, presented in Section 3. This representation facilitates the efficient support counting. Although this approach can be followed for the forthcoming phases as well, based on the observation in Ref. [25] that in phases larger than two the *DHP* algorithm does not produce significant improvements (and as verified by experiments we performed for the case of labeled trees), we follow the procedure of *WL* algorithm for the support counting of k-tree-expressions for $k>2$. The complete mining algorithm is depicted in Fig. 4.

As illustrated, steps 2–5 discover the set of frequent 1-tree-expressions (F_1). In steps 3 and 4 each transaction is

```
Procedure Mabers(float MINSUP)
begin
1.  /* phase 1 */
2.  compute $F_1$
3.  forall $t \in D$
4.      trim $t$ by removing each $p \in t, p \notin F_1$
5.  endfor

6.  /* phase 2 */
7.  $k = 2$
8.  $C_k$ = GenCandidates($F_{k-1}$)
9.  forall $c \in C_k$
10.     $m_c$ = LTE($c$)
11.     insert $m_c$ in $H$[hash($m_c$)]
12. endfor
13. forall $t \in D$
14.     forall $s \in t, s \in C_k$
15.         $m_s$ = LTE($s$)
16.         find all $c$ in $H$[hash($m_s$)] such that $m_c == m_s$
17.         if TreeMatching($s$, $c$)
18.             $c$.support++
19.         endif
20.     endfor
21. endfor
22. $F_k = \{c \in C_k | c.\text{support} \geq \text{MINSUP}\}$
23. $k$++
24. $C_k$ = GenCandidates($F_{k-1}$), construct $\Pi_k$

25. /* phase $k > 2$ */
26. while $C_k \neq \emptyset$
27.     forall $t \in D$
28.         forall $c \in \Pi_k$
29.             if WeakerThan($c$, $t$)
30.                 $c$.support++
31.             endif
32.         endfor
33.     endfor
34.     forall $c \in \Pi_k$
35.         if $c$.support < MINSUP
36.             delete $c$ from $\Pi_k$
37.         endif
38.     endfor
39.     $F_k = \{c \in C_k | c.\text{support} \geq \text{MINSUP}\}$
40.     $k$++
41.     $C_k$ = GenCandidates($F_{k-1}$), construct $\Pi_k$
42. endwhile
end
```

Fig. 4. Complete algorithm.

trimmed against F_1 by removing all path sequences that do not belong to F_1 This optimization helps the forthcoming phases, since the complexity of the tree matching procedure depends on the tree sizes (this optimization is also applicable to *WL* algorithm).

For all candidates generated from F_1, the magic number is found with the *LTE* algorithm, and it is stored in a hash structure together with a pointer to the corresponding labeled tree (steps 8–12). The size of the hash structure depends on the available main memory. To count the support of candidate 2-tree-expressions, the 2-tree-expressions components of each transactions are found and the corresponding magic numbers are calculated (step 14). For each such magic number the hash structure is probed and when a matching among magic numbers is found (steps 15 and 16), the corresponding trees are examined with the tree-matching algorithm (step 17). In case of a match, the support of the candidate is increased by one (step 18). The set of large candidates comprise the set of large 2-tree-expressions, denoted as F_2 (step 22). As described, for the next phases, the procedure of *WL* for the candidate support counting is followed (steps 25-42). Based on Ref. [29], support counting is facilitated by the Π_k structure.

Since the selectivity of the Π_k structure is not high (see Section 1.2), by considering the very large number of candidates in the second phase which is the bottleneck of the whole mining procedure, the *WL* algorithm presents a significantly larger cost compared to the procedure followed by the *Mabers* algorithm of Fig. 4. The objective of encoding labeled trees with magic numbers is to result in fast operations of probing the hash structure instead of performing tree matchings as in the case of *WL* and the Π_k structure. Experimental results in Section 5 verify this conjecture.

5. Experiments

This section presents the experimental results on the performance of the *Mabers* and the *WL* algorithm. Similar to Ref. [29], we used synthetic as well as real data in order to evaluate the performance of the algorithms over a range of data characteristics. For the synthetic data, it was pointed out in Ref. [29] that the performance of the algorithms is affected by the number of candidates of each phase and not from the number of transaction objects or the number of labels. Thus, we present experiments where the independent variable is the number of candidates. In this way, we are able to 'capture' a wide range of different database configurations (small/large databases, many/few labels, many/few nodes, etc.) by varying the number of candidates. As for the real datasets, similar to Ref. [29], we used a portion of the Image Database (found at us.imdb.com).

In the Section 5.1, we describe the experiments with synthetic data and in Section 5.2 we present the results for real datasets.

5.1. Experiments with synthetic data

5.1.1. Generation of synthetic workloads

We generated acyclic transaction objects whose nodes have list semantics only. Each workload is a set of transaction objects. The method used to generate synthetic transaction objects is based on Refs. [3,29] to the extent possible, since not all details are clearly documented. In the next paragraphs, we give a comprehensive overview of the workload generator.

Each transaction object is essentially a hierarchy of objects. *Atomic* objects are the objects having no descendants. The *height* of an object is the length of the longest path from that object to a descendant atomic object. Atomic objects are at level 1. The level of a non-atomic object equals the length of the longest path from that object to a descendant atomic object. All transaction objects are at the same level m, which is the *maximal nesting level*. Each object is recognized by an identifier. The number of identifiers for objects of level i is N_i. Each object is assigned one (*incoming*) label which represents a 'role' for that object. Any object i that has as subobject an object j, will be connected to j through an edge labeled with the label of object j. This means that once we select the subobjects for an object we immediately have decided about the label of their linking edge. All transaction objects have the same incoming label. Next we describe how to draw labels for object identifiers and how we select the subobjects of any non-atomic object.

Objects belonging to the same level are assigned labels drawn from a set which is different for each level and its cardinality is L_i for the ith level. We treat each object serially and draw a label using a self-similar (fractal) distribution [11]. This power law provides the means to select some labels more frequently than others and thus to simulate the fact that some 'roles' appear more frequently. A parameter of this distribution determines the skewness of the distribution ranging from uniform to highly skewed. In our experiments, we used a parameter equal to 0.36, so as to simulate a situation where labels are not equiprobably selected, but on the other hand almost all labels are used when the number of selections is large enough compared to the set of labels.

The next issue that must be addressed is how many and which lower level objects to select as subobjects for any given object at level i. First, the number of its subobject references is decided to be uniformly distributed with mean equal to T_i. The selection of subobjects must model the fact that some structures appear in common in many objects. In order to achieve this, we used the notion of *potentially large sets* [3]. Thus, subobject references for an object at level i are not completely random, but instead are drawn from a pool of *potentially large sets*. If the *maximum nesting level* equals m, then this pool is comprised by $m-1$ portions, namely $\Gamma_1,\Gamma_2,\ldots,\Gamma_{m-1}$. Each Γ_i is comprised by sets of level-i identifiers. The average size of such a set is I_i. For more details regarding the distribution of cardinality of these sets and the method for selecting object identifiers can be found in Ref. [3]. The construction of the objects is a bottom-up process. Starting from level-2, we must construct N_2 objects. For each object, we first choose the number of its subobject references (that is, its size) and then pick several potential large sets from Γ_1 until its size is reached. Recursively, we construct the level-3 objects and so on. For any object belonging to any level, say level $i>2$, we obligatorily choose one potentially large set from Γ_{i-1} and then we choose the rest potentially large sets equiprobably from all Γ_j, $1\leq j<i$.

Table 1
Notation used for the generation of synthetic data

Symbol	Explanation
L_i	Number of level-i labels
N_i	Number of level-I object identifiers
T_i	Average size of val(o) for level-i identifiers o
I_i	Average size of potentially large sets in Γ_i
P_i	Number of potentially large sets in Γ_i
m	Maximal nesting level

Table 2
Data sets used for the experimental evaluation

Symbol	Data set
D_1	⟨100,5K,1,3,100⟩, ⟨1K,5K,15,3,300⟩, ⟨1K,5K,15,3,700⟩, ⟨1K,5K,15,3,700⟩, ⟨1,30K,15,0,0⟩
$D2$	⟨100,5K,1,3,100⟩, ⟨500,5K,15,3,300⟩, ⟨500,5K,15,3,700⟩, ⟨500,5K,15,3,700⟩, ⟨1,30K,15,0,0⟩
$D3$	⟨100,5K,1,3,100⟩, ⟨750,5K,15,3,300⟩, ⟨750,5K,15,3,700⟩, ⟨750,5K,15,3,700⟩, ⟨1,30K,15,0,0⟩
$D4$	⟨100,5K,1,3,100⟩, ⟨500,5K,15,3,300⟩, ⟨500,5K,15,3,700⟩, ⟨500,5K,15,7,700⟩, ⟨1,30K,15,0,0⟩

Thus, a generated data set in which transaction objects are at level m will be represented as: $\langle L_1,N_1,I_1,P_1\rangle,\langle L_2,N_2,T_2,I_2,P_2\rangle,\ldots,\langle N_m,T_m\rangle$.[4] Table 1 summarizes the meaning of various parameters used by the generator, whereas Table 2 describes the datasets we used, according to the notation of Table 1. The examined values were selected along the lines of those used by Wang and Liu [29].

5.1.2. Cost breakdown

The first experiment aimed at identifying the computational requirements of each phase for *WL*. Due to reasons explained in Section 4, we focus on phases F_k, $k\geq 2$ (second stage). Table 3 depicts the number of candidates $|C_k|$ for phase k for the data sets D_1 and D_2. We can observe that $|C_2|$ is as much as two orders of magnitude larger than the number of candidates for the other phases. Notice also that the difference in $|C_2|$ between D_1 and D_2 is due to the smaller number of labels in D_2, which increases the search space, as indicated in Ref. [29]. The large size of $|C_2|$ comprises the bottleneck of the whole procedure, as illustrated in Fig. 5 which presents the execution times of each phase. This verifies the motivation stated in Section 1.2.

5.1.3. Comparison of Mabers and WL

Next, we conducted an experiment to compare the performance of *Mabers* and *WL* with respect to the number

[4] Remember that $T_1=0$, $L_m=1$ and that there is no Γ_m.

Table 3
Number of candidate tree-expressions for each phase

	D_1	D_2
C_2	33,016	117,870
L_2	146	161
C_3	364	358
L_3	20	28
C_4	15	29
L_4	0	0

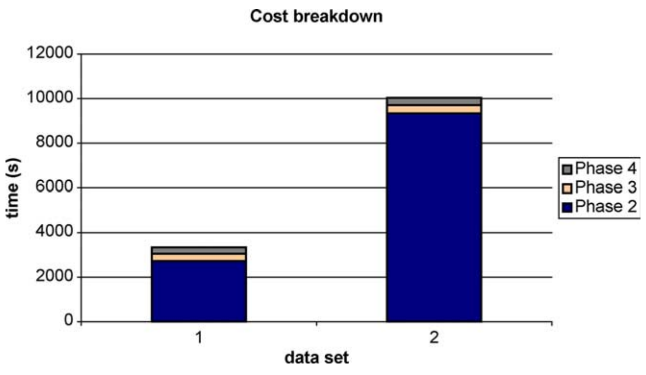

Fig. 5. Execution time for each phase.

of candidates. Both algorithms were implemented using the same components (wherever feasible). However, in order to decouple the comparison from specific software and hardware implementations, we used as performance measure the number of numerical operations executed by each algorithm.

Fig. 6 depicts the results of the comparison with respect to the number of candidates. We used the data sets D_1–D_4, which produce number of candidates in the range from a few tenths to a hundred of thousands. As depicted, the cost of both algorithms increases with increasing number of candidates. However, the increase for *WL* is much more noticeable indicating that it is affected by the bottleneck. On the other hand, *Mabers* is not affected as much as *WL* and it achieves a performance improvement up to two orders of magnitude.

5.2. Experiments with real data

In order to test our finding about the combinatorial explosion in the number of candidate 2-tree-expressions and the relative performance of the algorithms over real datasets, we extracted a small subset of the movie objects found at the Image Database (http://us.imdb.com). We extracted the relevant information about the movies produced in the United States during the last 20 years, i.e. (1983 ≤ *Released_-Year* ≤ 2003 AND *Country* = *USA*). In response we got around 11,000 movie titles. Along the lines of Ref. [29], we retrieved the most common fields of these movies for the OEM graph we built. We set the support threshold to 30% and run the two algorithms. Firstly, we examined the number of candidate tree-expressions of each phase, to test whether the combinatorial explosion problem still persists, because real data are 'denser' than synthetically generated. Although, we found a small alteration in the relative number of candidates in each phase and a larger number of phases, as can be seen in Table 4, the combinatorial explosion problem is still present.

As for the relative performance of the algorithms, the *WL* algorithm performs as much as 3×10^8 numerical operations (node comparisons), whereas the *Mabers* algorithm performs around 9×10^6 node comparisons, which is more than an order of magnitude faster than *WL*.

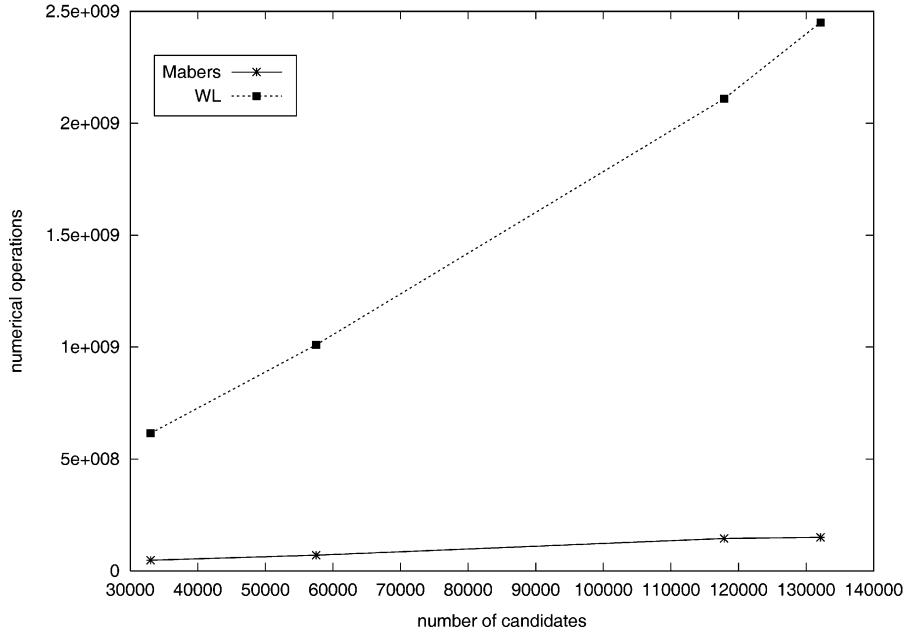

Fig. 6. Comparison between *Mabers* and *WL*.

Table 4
Number of candidate tree-expressions for each phase in real data

	$D1$
C_2	17,612
L_2	503
C_3	913
L_3	328
C_4	164
L_4	107
C_5	96
L_5	60
C_6	44
L_6	11
C_7	4
L_7	0

6. Conclusion

We considered the problem of the efficient discovery of structural associations for semistructured data. Motivated by the overhead incurred by the early phases of the discovery procedure, we examined how to efficiently index the search space of candidate tree-expressions.

We recognized that support counting for candidate 2-tree-expressions is the main performance bottleneck. We proposed a new algorithm, called *Mabers*, which uses a hashing scheme for ordered labeled trees in order to efficiently carry out the support counting procedure for the second phase. It compares favorably with a previously proposed algorithm, namely *WL* [29].

Using a synthetic data generator, the experimental results showed that *Mabers* clearly outperforms *WL*. For data sets with large number of candidates, *Mabers* is up to two orders of magnitude better than *WL*.

References

[1] K. Abe, S. Kawasoe, T. Asai, H. Arimura, S. Arikawa, Optimized substructure discovery for semi-structured data, in: Proceedings of the Sixth European Conference on Principles of Data Mining and Knowledge Discovery (PKDD), Lecture Notes in Artificial Intelligence, vol. 2431, 2002, pp. 1–14.

[2] S. Abiteboul, P. Buneman, D. Suciu, Data on the Web: From Relations to Semistructured Data and XML, Morgan Kaufmann, Los Altos, CA, 2000.

[3] R. Agrawal, R. Srikant, Fast algorithms for mining association rules in large databases, in: Proceedings of the 20th International Conference on Very Large Data Bases (VLDB), 1994, pp. 487–499.

[4] R. Agrawal, R. Srikant, Mining sequential patterns, in: Proceedings of the IEEE International Conference on Data Engineering (ICDE), 1995, pp. 3–14.

[5] T. Asai, K. Abe, S. Kawasoe, H. Arimura, H. Sakamoto, S. Arikawa, Efficient substructure discovery from large semi-structured data, in: Proceedings of the Second SIAM Conference on Data Mining (SDM), 2002, pp. 158–174.

[6] T. Asai, H. Arimura, T. Uno, S. Nakano, Discovering frequent substructures in large unordered trees, in: Proceedings of the Sixth International Conference on Discovery Science (DS), Lectures Notes in Artificial Intelligence, vol. 2843, 2003, pp. 47–61.

[7] G. Cong, L. Yi, B. Liu, K. Wang, Discovering frequent substructures from hierarchical semi-structured data, in: Proceedings of the Second SIAM Conference on Data Mining (SDM), 2002, pp. 175–192.

[8] A. Deutsch, M.F. Fernandez, D. Suciu, Storing semistructured data with STORED, in: Proceedings of the ACM International Conference on Management of Data (SIGMOD), 1999, pp. 431–442.

[9] M. Garofalakis, R. Rastogi, K. Shim, Mining sequential patterns with regular expression constraints, IEEE Transactions on Knowledge and Data Engineering 14 (3) (2002) 530–552.

[10] R. Goldman, J. Widom, Dataguides: enabling query formulation and optimization in semistructured databases, in: Proceedings of the 23rd International Conference on Very Large Data Bases (VLDB), 1997, pp. 436–445.

[11] J. Gray, P. Sundaresan, S. Englert, K. Baclawski, P.J. Weinberger, Quickly generating billion-record synthetic databases, in: Proceedings of the ACM International Conference on Management of Data (SIGMOD), 1994, pp. 243–252.

[12] J. Han, J. Pei, Y. Yin, Mining frequent patterns without candidate generation, in: Proceedings of the ACM International Conference on Management of Data (SIGMOD), 2000, pp. 1–12.

[13] J. Huan, W. Wang, J. Prins, Efficient mining of frequent subgraphs in the presence of isomorphism, in: Proceedings of the Third IEEE Conference on Data Mining (ICDM), 2003, pp. 549–552.

[14] A. Inokuchi, H. Kashima, Mining significant pairs of patterns from graph structures with class labels, in: Proceedings of the Third IEEE Conference on Data Mining (ICDM), 2003, pp. 83–90.

[15] A. Inokuchi, T. Washio, H. Motoda, An apriori-based algorithm for mining frequent substructures from graph data, in: Proceedings of the Fourth European Conference on Principles of Data Mining and Knowledge Discovery (PKDD), Lecture Notes in Artificial Intelligence, vol. 1910, 2000, pp. 13–23.

[16] A. Inokuchi, T. Washio, H. Motoda, Complete mining of frequent patterns from graphs: mining graph data, Machine Learning 50 (3) (2003) 321–354.

[17] J. Katzenelson, S.S. Pinter, E. Schenfeld, Type matching, type-graphs, and the Schanuel conjecture, ACM Transactions on Programming Languages and Systems 14 (2) (1992) 574–588.

[18] M. Kuramochi, G. Karypis, Frequent subgraph discovery, in: Proceedings of the First IEEE Conference on Data Mining (ICDM), 2001, pp. 313–320.

[19] J. McHugh, J. Widom, Query optimization for XML, in: Proceedings of the 25th International Conference on Very Large Data Bases (VLDB), 1999, pp. 315–326.

[20] T. Milo, D. Suciu, Index structures for path expressions, in: Proceedings of the Seventh International Conference on Database Theory (ICDT), 1999, pp. 277–295.

[21] A. Nanopoulos, D. Katsaros, Y. Manolopoulos, A data mining algorithm for generalized Wed prefetching, IEEE Transactions on Knowledge and Data Engineering 15 (5) (2003) 1155–1169.

[22] S. Nestorov, S. Abiteboul, R. Motwani, Extracting schema from semistructured data, in: Proceedings of the ACM International Conference on Management of Data (SIGMOD), 1998, pp. 295–306.

[23] S. Nestorov, J.D. Ullman, J.L. Wiener, S.S. Chawathe, Representative objects: concise representations of semistructured, hierarchical data, in: Proceedings of the 13th International Conference on Data Engineering (ICDE), 1997, pp. 79–90.

[24] Y. Papakonstantinou, H. Garcia-Molina, J. Widom, Object exchange across heterogeneous information sources, in: Proceedings of the IEEE International Conference on Data Engineering (ICDE), 1995, pp. 251–260.

[25] 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.

[26] A. Savasere, E. Omiecinski, S.B. Navathe, An efficient algorithm for mining association rules in large databases, in: Proceedings of the 21st International Conference on Very Large Data Bases (VLDB), 1995, pp. 432–444.

[27] H. Toivonen, Sampling large databases for association rules, in: Proceedings of the 22nd International Conference on Very Large Data Bases (VLDB), 1996, pp. 134–145.

[28] N. Vanetik, E. Gudes, S.E. Shimony, Computing frequent graph patterns from semistructured data, in: Proceedings of the Second IEEE Conference on Data Mining (ICDM), 2002, pp. 458–465.

[29] K. Wang, H. Liu, Discovering structural association of semistructured data, IEEE Transactions on Knowledge and Data Engineering 12 (3) (2000) 353–371.

[30] Q.Y. Wang, J.X. Yu, K.-F. Wong, Approximate graph schema extraction for semistructured data, in: Proceedings of the Seventh International Conference on Extending Data Base Technology (EDBT), Lecture Notes in Computer Science, vol. 1777, Springer, Berlin, 2000, pp. 302–316.

[31] Y. Xiao, J.-F. Yao, Z. Li, M.H. Dunham, Efficient data mining for maximal frequent subtrees, in: Proceedings of the Third IEEE Conference on Data Mining (ICDM), 2003, pp. 379–386.

[32] X. Yan, J. Han, gSpan: graph-based substructure pattern mining, in: Proceedings of the Second IEEE Conference on Data Mining (ICDM), 2002, pp. 721–724.

[33] L.H. Yang, M.L. Lee, W. Hsu, Efficient mining of XML query patterns for caching, in: Proceedings of the International Conference on Very Large Data Bases (VLDB), 2003, pp. 69–80.

[34] M.J. Zaki, Efficiently mining frequent trees in a forest, in: Proceedings of the Eighth ACM Conference on Knowledge Discovery and Data Mining (SIGKDD), 2002, pp. 71–80.