

# Discovering Structural Association of Semistructured Data

Ke Wang and Huiqing Liu

**Abstract**—Many semistructured objects are similarly, though not identically, structured. We study the problem of discovering “typical” substructures of a collection of semistructured objects. The discovered structures can serve the following purposes: 1) the “table-of-contents” for gaining general information of a source, 2) a road map for browsing and querying information sources, 3) a basis for clustering documents, 4) partial schemas for providing standard database access methods, and 5) user/customer’s interests and browsing patterns. The discovery task is impacted by structural features of semistructured data in a nontrivial way and traditional data mining frameworks are inapplicable. We define this discovery problem and propose a solution.

**Index Terms**—Association rule, database, data mining, knowledge discovery, semistructured data, web mining.

## 1 INTRODUCTION

### 1.1 Motivation

MANY on-line documents, such as HTML, Latex, BibTex, SGML files, and those found in digital libraries, are *semistructured*. Semistructured data arises when the source does not impose a rigid structure (such as the Web) and when data is combined from several heterogeneous sources (such as data warehousing). Unlike unstructured raw data (such as image and sound), semistructured data does have some structure. Fig. 1 shows a segment of semistructured movie objects maintained by IMDb (<http://us.imdb.com>). Each circle plus the text inside represents a subobject (e.g., a HTML file) and its identifier (e.g., URL). The links and their labels, identifiable by special tags or a grammar, represent subobject references and their roles. In this paper, the term *structure* refers to the hierarchy of such references and roles. The structure of an object gives a sense of what sort of questions might be answered by a more intensive examination of the object and how the information is represented. A recent review has revealed that, nearly always, references to important objects are labeled rather than in the form of free-running text [6].

Unlike structured data (such as relational or object-oriented databases), semistructured data has no absolute schema or class fixed in advance and each object contains its own “schema.” For example, some movies have more actors than others; some fields (e.g., *Award*) are missing for some movies; some actors have their birthdays recorded and some do not; some have spouses and some do not, etc. As a result, the structure of objects is irregular and a query over the structure is as important as query over the data. This structural irregularity, however, does not imply that there is

no structural similarity among semistructured objects. On the contrary, it is common for semistructured objects describing the same type of information to have similar structures. For example, every movie object has *Title* and *Director* labels; every *Actor* object has a *Name* label; 50 percent of *Actor* objects have a *Nationality* label, etc. Some examples of semistructured objects having similar structures are universities, countries, census data, branch information within an organization, etc. The topic of this paper is discovering the structural similarity of a collection of semistructured objects. We first define the problem and then discuss its applications.

### 1.2 Main Results

We consider the following discovery problem: Given a collection of semistructured objects, find all “typical” (sub)structures that occur in a minimum number of objects specified by the user. We formally define this problem in Section 2. It is worth mentioning that though we refer to the “structure” of an object, it is up to the user to specify what the structure is. For example, if the user wants to find frequent cooccurrences of keywords in several text documents (thus, no structure in the usual sense), he/she can specify keywords as labels, in which case a typical structure is a set of keywords that cooccur in some minimum number of text documents. In this view, our framework generalizes the classical association rule problem motivated in the supermarket environment [2] where the core problem is finding typical subsets of (supermarket) items that are contained in some minimum number of (supermarket) transactions. The generalization lies in that we consider general structures, instead of flat sets, that have interesting features such as hierarchy, labeling, ordering, and cyclicity.

It should be pointed out that our work differs from those on extracting the structure of a single individual object [15]. We consider a collection of graph structures, each representing a semistructured object, and discover substructures that appear in some minimum number of graph structures. In particular, we have to deal with the requirement on the minimum number of occurrences of substructures. Prior to

- K. Wang is with the School of Computing Science, National University of Singapore, 119260. E-mail: wangk@comp.nus.edu.sg
- H. Liu is with the BioInformatics Centre, National University of Singapore, 119260. E-mail: huiqing@bic.nus.edu.sg.

Manuscript received 18 Aug. 1997; accepted 21 July 1999.

For information on obtaining reprints of this article, please send e-mail to: tkde@computer.org, and reference IEEECS Log Number 105515.

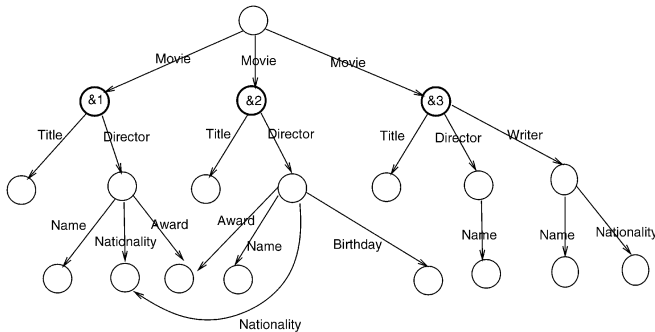


Fig. 1. A segment of movie objects.

the discovery task, the structure of each object should be extracted by removing unstructured data, such as image and video, that do not contribute to the structure of the source. Often, a low-level representation (such as HTML) should be transformed to a conceptual model at a higher level of abstraction to hide away details not interesting to the user. These could include links and layers that are not interesting to the user. Some sources provide “wrappers” or one can write a parser to do this [15]. We assume that such extraction has been done. Another concern is when the discovery is performed. Depending on applications, the discovery can be performed either off-line, where discovered structures are saved for future retrievals, or on-line where the discovery is done for a specific request. Each discovered structure can be associated with identifiers (e.g., URL) of the objects that contain the structure. This will allow relevant objects to be retrieved and examined for further analysis.

### 1.3 Application

The following list gives examples of applications used for discovering typical structures of semistructured objects.

- **Road maps for querying/browsing information sources.** One limitation of querying and browsing semistructured data is the disorientation resulting in the infamous “lost-in-hyperspace” syndrome, due to the lack of external schema. To formulate any meaningful query, say in WebSQL [7] or W3QS [5] for Web documents that matches some of the source’s structure, we first need to discover something about how the information is represented in the source. This subtask can be formulated as discovering typical structures of objects. Some Web query languages allow specification of a wild card label in a query that matches any label. Discovering typical structures that may contain wild cards is helpful for formulating such queries.
- **General information content.** Very often, a user may not be looking for anything specific at all but rather may wish to discover the general information content of a source. For such users, it is hard to formulate a query to browse all documents. A more appropriate search mode would be examining the structure of the source, just like examining the table-of-contents if a reader would like to gain the gist of a book. This can be done by requesting the display of

the structure of each document if there are only a few documents, or the display of some typical structures if there are many documents. Since such requests are likely to be frequent, typical structures should be discovered off-line and stored in a database that is queried or browsed on demand. Based on the structures examined, the user may at any time switch to a more focused search method, such as formulating a query or browsing some documents.

- **A guideline for building indexes and views.** To speed up information retrieval, it is desirable to construct indexes and views on frequently retrieved, typically occurring structures. Discovering typical structures can help this task. We quote [1] for the motivation in this context: “one could envision the use of general purpose data mining tools to extract structuring information. One can then use the information extracted from the files to build a structured layer above the layer of more unformed data. This structured layer references the lower data layer and yields a flexible and efficient access to the information in the lower layer to provide the benefits of standard database access methods.” For example, if *Phone* label is typical of person objects and are often used to retrieve personal information, building an index on *Phone* (e.g., by a B-tree, hash table, or inverted list) can speed up the retrieval.
- **Structure-based document clustering.** The tree-like structure of subdocument references within a document is usually ignored by traditional clustering methods. In a semistructured document, each subdocument reference is labeled by its role and the “topic” of a document is represented by the tree-like structure of such roles rooted at the document. Consequently, the topic of a subdocument is relative to that of its superdocument. For example, nation’s birthday and person’s birthday are considered as different topics. If documents are clustered based on such topical structures, the search for nation’s birthday information will not return person’s birthday information.
- **Discovering interests/access patterns.** Detecting user’s interests and browsing patterns on the Web can help organize Web pages and attract more businesses. This can be modeled as discovering typical structures of a collection of semistructured objects. Each semistructured object consists of hyperlinked Web pages accessed in a single session. By labeling each page with either topic or site information, a typical structure captures user’s interests or access patterns.

This paper is organized as follows: Section 2 defines the problem of discovering typical structures. Section 3 presents an algorithm. Section 4 evaluates the efficiency of the algorithm. Section 5 presents a case study using a real dataset. Section 6 reviews related work. Section 7 concludes the paper.

## 2 THE PROBLEM

We first define a representation of semistructured data. Then, we define the discovery problem.

## 2.1 The Object Exchange Model

We adopt the *Object Exchange Model (OEM)* for representing semistructured data. For a detailed account of the OEM, the interested reader may refer to [1], [4], [10]. In OEM, every object  $o$  consists of an *identifier*, denoted  $\&o$ , and a *value*, denoted  $val(\&o)$ . The identifier  $\&o$  uniquely identifies object  $o$ . The value  $val(\&o)$  is either an *atomic*, such as an integer or a string; or a *list*  $\langle l_1 : \&o_1, \dots, l_p : \&o_p \rangle$ ,  $p > 0$ ; or a *bag*  $\{l_1 : \&o_1, \dots, l_p : \&o_p\}$ ,  $p > 0$ .  $\&o_i$  are identifiers of subobjects  $o_i$ .  $l_i$  are *labels* that describe the role of subobjects  $o_i$ . There is no requirement that subobjects  $o_i$  are uniformly lists or bags. As usual, the order in a bag does not matter, but it does in a list. Repeating of subobjects  $\&o_i$  or labels  $l_i$  is allowed in a bag and a list. The original OEM considers only the bag semantics. We extend it to the list semantics to deal with ordered subobject references. For example, actor subobjects of a movie object are usually listed in the order of actor's credits; subroutine calls in a procedure are listed in the order of calls.

OEM is conveniently represented by a labeled multigraph. In the graph, each node represents an object identifier  $\&o$  and each edge  $(\&o, \&o_i)$  labeled  $l_i$  represents a reference  $l_i : \&o_i$  in  $val(\&o)$ . The outgoing edges at node  $\&o$  may or may not be ordered, depending on whether  $val(\&o)$  is a list or a bag. We use a circled node to represent an identifier  $\&o$  of a bag value  $val(\&o)$ , and use a squared node to represent an identifier  $\&o$  of a list value  $val(\&o)$ . An OEM database is *cyclic* if its graph is cyclic. Indeed, OEM graphs of many Web documents are cyclic. For example, *Spouse* links are cyclic.

For the discovery task (defined shortly), the user needs to specify a collection of objects in the OEM graph for which typical structures are discovered. These objects are called *transaction objects*. For example, if the user is interested in typical structures of a collection of movie objects, the nodes representing movie objects should be specified as transaction objects; however, if the user is interested in typical structures of actor objects, the nodes representing actor objects should be specified as transaction objects. (Note that transaction objects are not necessarily the root nodes in the whole OEM graph.) The purpose of specifying transaction objects is analogous to that of specifying transactions in the context of mining association rules [2] where the user has to decide, for example, whether to include data from the shoe department, toy department, and food department for a particular discovery task. Typically, transaction objects should contain similar types of information—it does not make sense to discover common structures of actor objects and country objects. To automate the specification of transaction objects, one can quantify the sequence of leading labels (thus, the role) of transaction objects in the OEM graph. For example, the sequence of labels *Movie : Director : Award* specifies all award objects of directors as transaction objects. More generally, the collection of transaction objects could be returned by a query for semistructured data [7], [5]. Thus, in one case, we could find common structures for movies in English and in another case, we could find common structures for movies in foreign languages.

## 2.2 Generalizing Several Objects

A key concept in our discovery problem is that of generalizing the structure of objects. This is done by partially expanding subobject references: If object  $\&o$  contains subobject references  $l_1 : \&o_1, \dots, l_p : \&o_p$ , a partial structure of  $\&o$  consists of some of these references and, optionally, their partial structures. The expansion is *partial* because it can ignore some references and can stop at any level. The significance of partial structures lies in that several objects may share partial structures even though they do not share the full structure. For the rest of the paper, symbol  $?$  denotes the *wild card* label that matches any label and symbol  $\perp$  denotes the *nil structure* that contains no label. A partial structure of  $\&o$  is represented by a tree of labels, called *tree-expressions* below.

**Tree-expressions.** First, we consider an acyclic OEM graph. For any label  $l$ , let  $l^*$  denote either  $l$  or the wild card label  $?$ .

1. The nil structure  $\perp$  is a *tree-expression* of any object;
2. Suppose that  $te_i$  are *tree-expressions* of objects  $o_i$ ,  $1 \leq i \leq p$ . If  $val(\&o) = \{l_1 : \&o_1, \dots, l_p : \&o_p\}$  and  $\{i_1, \dots, i_k\}$  is a subset of  $\{1, \dots, p\}$ ,  $k > 0$ , then  $\{l_{i_1}^* : te_{i_1}, \dots, l_{i_k}^* : te_{i_k}\}$  is a *tree-expression* of object  $o$ ;
3. Suppose that  $te_i$  are *tree-expressions* of objects  $o_i$ ,  $1 \leq i \leq p$ . If  $val(\&o) = \langle l_1 : \&o_1, \dots, l_p : \&o_p \rangle$  and  $\langle i_1, \dots, i_k \rangle$  is a subsequence of  $\langle 1, \dots, p \rangle$ ,  $k > 0$ , then  $\langle l_{i_1}^* : te_{i_1}, \dots, l_{i_k}^* : te_{i_k} \rangle$  is a *tree-expression* of object  $o$ .

One additional requirement is that  $?$  should not appear as the “terminal” label on a label path in a tree-expression. This follows from the intended use of wild card label  $?$ , i.e., to ignore an upper part of an object's structure in order to discover something common at a lower part. This requirement can be phrased as: if  $te_{i_j}$  is  $\perp$ ,  $l_{i_j}^*$  must be  $l_{i_j}$ . A tree-expression  $\{l_{i_1} : te_{i_1}, \dots, l_{i_k} : te_{i_k}\}$  or  $\langle l_{i_1} : te_{i_1}, \dots, l_{i_k} : te_{i_k} \rangle$  has a natural tree representation: it consists of  $k$  subtrees  $te_{i_j}$ , each being labeled  $l_{i_j}$ .

**Example 2.1.** Consider Fig. 1. By recursively applying construction 2 of tree-expressions,

$$te_1 = \{Director : \{Name : \perp\}, Title : \perp\}$$

is a tree-expression of  $\&1$ . Similarly,  $te_1$  is a tree-expression of  $\&2$  and  $\&3$ . If we replace *Director* with  $?$  in  $te_1$ , the result is still a tree-expression of  $\&1, \&2, \&3$ . However, if we replace *Name* or *Title* with  $?$  in  $te_1$ , the result is not a tree-expression because a “terminal” label cannot be the wild card.

$$te_2 = \{Director : \{Name : \perp, Nationality : \perp\}, Title : \perp\}$$

and

$$te_3 = \{Director : \{Name : \perp, Nationality : \perp, Award : \perp\}, Title : \perp\}$$

are tree-expressions of  $\&1$  and  $\&2$ , but not of  $\&3$ .  $te_4 = \{? : \{Name : \perp, Nationality : \perp\}\}$  is a tree-expression of  $\&1, \&2, \&3$ . Fig. 2 shows the tree representation for  $te_1, te_2, te_3, te_4$ .

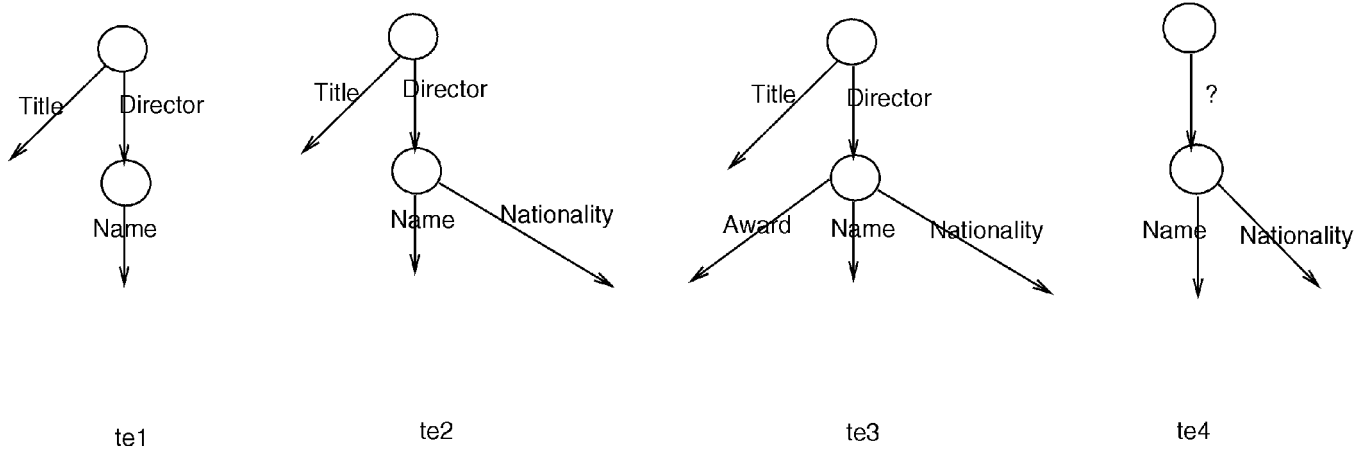


Fig. 2. Some tree-expressions of movie objects.

We would like to mention that other choices of wild card labels are possible. For example, a wild card label could match any label in a given set, but not any label outside it. If such wild card labels are fixed, our framework can be easily modified to discover tree-expressions that may contain such wild cards. However, if there is no fixed set of such wild-card labels, the complexity of the discovery problem will be drastically increased because every wild card defined by a superset containing label  $l$  is a generalization of  $l$ . To keep the problem manageable, we do not consider such wild cards.

For a cyclic OEM graph, tree-expressions defined above may be infinitely large. To address this problem, we allow a leaf node in a tree-expression to be named by a special symbol  $\perp_i$ ,  $i > 0$ . Essentially, a leaf node named  $\perp_i$  is the alias of the ancestor that is  $i$  nodes above the leaf node. This ancestor is called the  $i$ th ancestor. Fig. 3 shows how a cycle (on the left) is represented in a tree-expression (on the right). The “leaf” named  $\perp_3$  is the alias of its third ancestor  $A$ . By treating each  $\perp_i$  node as a leaf node, we are able to deal with a tree-expression containing cyclic references (like the one on the left in Fig. 3) as a tree (like the one on the right in Fig. 3) without losing information. Therefore, all tree-expressions, cyclic or acyclic, are treated as trees.

Sometimes, we are interested in the most “informative” partial structures. For example, in Fig. 2,  $te_3$  is more informative than  $te_2$  which is more informative than  $te_1$ . The “weaker than” relationship below compares the informativeness of tree-expressions.

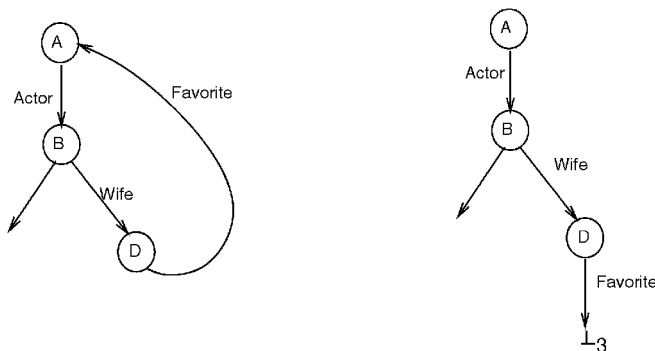


Fig. 3. Tree-expressions extended to represent cycles.

**Weaker than.** The nil structure  $\perp$  is weaker than every tree-expression.  $\perp_i$  is weaker than itself.

- Tree-expression  $\{l_1 : te_1, \dots, l_p : te_p\}$  is weaker than tree-expression  $\{l'_1 : te'_1, \dots, l'_q : te'_q\}$  if for  $1 \leq i \leq p$ ,  $te_i$  is weaker than some  $te'_{j_i}$ , where either  $l'_{j_i} = l_i$  or  $l_i = ?$ , and  $\{j_1, \dots, j_p\}$  is a subset of  $\{1, \dots, q\}$ ;
- tree-expression  $\langle l_1 : te_1, \dots, l_p : te_p \rangle$  is weaker than tree-expression  $\langle l'_1 : te'_1, \dots, l'_q : te'_q \rangle$  if for  $1 \leq i \leq p$ ,  $te_i$  is weaker than some  $te'_{j_i}$ , where either  $l'_{j_i} = l_i$  or  $l_i = ?$ , and  $\langle j_1, \dots, j_p \rangle$  is a subsequence of  $\langle 1, \dots, q \rangle$ ;
- tree-expression  $te$  is weaker than identifier  $\&o$  if  $te$  is weaker than  $val(\&o)$ .

Intuitively, if tree-expression  $te$  is weaker than tree-expression  $te'$ , all structural information of  $te$  (about labeling, nesting, and ordering) are found in  $te'$ , starting at the root of  $te'$ .

### 2.3 The Discovery Problem

**Definition 2.1.** Consider a collection of transaction objects in an OEM graph and a minimum support  $MINISUP$  (in percentage). The support of a tree-expression  $te$  is the percentage of transaction objects  $t$  such that  $te$  is weaker than  $\&t$ .  $te$  is frequent if the support of  $te$  is not less than  $MINISUP$ .  $te$  is maximally frequent if  $te$  is frequent and is not weaker than other frequent tree-expressions. The discovery problem is to find all frequent tree expressions. The maximal discovery problem is to find all maximally frequent tree-expressions.

**Example 2.2.** In Fig. 1, suppose that  $\&1, \&2, \&3$  are the user-specified transaction objects, written in boldface. Refer to Fig. 2 for tree-expressions  $te_1, te_2, te_3, te_4$ . The support of  $te_1$  and  $te_4$  is  $3/3$ , and the support of  $te_2$  and  $te_3$  is  $2/3$ .  $te_1, te_2, te_4$  are weaker than  $te_3$ . Therefore, if  $MINISUP = 2/3$ ,  $te_1, te_2, te_3, te_4$  are frequent, but only  $te_3$  is maximally frequent. If  $MINISUP = 3/3$ , both  $te_1$  and  $te_4$  are maximally frequent.

Using the discovered frequent tree-expressions, one can derive association rules about substructures of objects. An association rule has the form  $\alpha \rightarrow \beta$ , where  $\alpha$  and  $\beta$  are frequent tree-expressions such that  $\alpha$  is weaker than  $\beta$ .

Assume that  $a$  and  $b$  are supports of  $\alpha$  and  $\beta$ .  $\alpha \rightarrow \beta$  says that a transaction object containing  $\alpha$  will contain  $\beta$  at confidence of  $b/a$  and support of  $a$ . Interesting association rules  $\alpha \rightarrow \beta$  must satisfy a minimum confidence and minimum support specified by the user. Since constructing association rules from frequent tree-expressions is straightforward, for the rest of the paper, we focus on the discovery problem and maximal discovery problem.

Before ending this section, let us explain our choice of trees as substructures versus graphs. First of all, without changing the role of a subobject, an OEM graph can be equally represented by a tree through replicating shared subobjects. As such, our goal of discovering roles of subobjects is not affected by using trees as substructures. There is indeed some information loss on sharing of subobjects by going from graphs to trees; it is no longer possible to tell if several references in a tree-expression are referring to a shared or different subobjects. To obtain such information, the identity of nodes involved (in addition to labels) needs to be kept in a tree-expression. This will drastically increase the number of tree-expressions and blow up the search space. Our choice of trees as substructures is a compromise between the completeness of information and the efficiency of implementation.

### 3 THE ALGORITHM

In this section, we present an algorithm for the discovery problems in Definition 2.1. The problem of finding frequent subsets from a collection of supermarket baskets [2] is related to our problems here. However, [2] is not directly applicable to objects having structures in the form of labeled hierarchical subobject references. Also, the flat representation in [2] is not able to represent partially ordered references. In addition, our search space includes substructures containing the wild card label that match any label. These new requirements justify the need to present a new mining algorithm.

We do not assume that the OEM graph  $G$  fits in the memory. Each node in the graph is accessed by its address, either on disk or in memory. To avoid repeatedly traversing subgraphs, due to multiple edges between two nodes in a multigraph, we assume that there is at most one “physical” edge from one node to another and that a set of labels is associated with each edge.  $L(\&w, \&z)$  denotes the set of labels associated with edge  $(\&w, \&z)$ , defined as the set of labels for  $\&z$  in  $val(\&w)$ . The intended use of  $L(\&w, \&z)$  is as follows: Each time a path  $\&w_1, \dots, \&w_k$  is traversed, where  $\&w_i$ s are nodes, all paths  $\&w_1, l_2, \&w_2, \dots, l_k, \&w_k$  are considered traversed, where  $(l_2, \dots, l_k)$  is in the cross product  $L(\&w_1, \&w_2) \times \dots \times L(\&w_{k-1}, \&w_k)$ . The information stored at each node  $\&w$  in  $G$  includes 1) the address and  $L(\&w, \&z)$  for every subnode  $\&z$  and 2) the positions in  $val(\&w)$  for each label in  $L(\&w, \&z)$ . For example, suppose that  $\&o = \{l_1 : \&o_1, l_2 : \&o_1, l_1 : \&o_2\}$ . Then  $L(\&o, \&o_1) = \{l_1, l_2\}$  and  $L(\&o, \&o_2) = \{l_1\}$ . At node  $\&o$ , the following information is stored: 1) the addresses of  $\&o_1$  and  $\&o_2$ ,  $L(\&o, \&o_1)$  and  $L(\&o, \&o_2)$  and 2)  $\&o_1$  is labeled  $l_1$  and  $l_2$  at Positions 1 and 2, and  $\&o_2$  is labeled  $l_1$  at Position 3.

An important property of our algorithm is traversing only *simple* paths of  $G$  in the depth-first order (a path is

simple if only the last node on it can repeat). Ideally, nodes of  $G$  should be stored in this depth-first order. However, since several supernodes may reference the same subnode, nodes adjacent in the depth-first order may not be necessarily on the same disk page. To reduce the disk access, frequently referenced nodes, i.e., those with a large in-degree and at lower levels, can be stored in memory and infrequently referenced nodes stored on the disk. This can be implemented by *pinning* the pages containing frequently referenced nodes in memory until they are not needed. However, the exact implementation on disk is transparent to the presentation of our algorithm.

#### 3.1 Representing Tree-Expressions

The set of tree-expressions defines the search space of the discovery problem. Before presenting a search algorithm, we need a convenient representation of tree-expressions.

A  $k$ -tree-expression is a tree-expression containing exactly  $k$  leaf nodes (i.e., nodes for  $\perp$  or  $\perp_i$ ). Each leaf node corresponds to a *label path* (path for short) of the form  $[\top, l_1, \dots, l_n, \perp]$ , where symbol  $\top$  represents a generic transaction object and  $l_i$  are labels on a simple path in  $G$  starting from a transaction object. As discussed in Section 2,  $\perp$  is replaced with  $\perp_i$  if the last node on the path repeats its  $i$ th ancestor. Each  $k$ -tree-expression can be constructed by a sequence of  $k$  paths  $(p_1, \dots, p_k)$  of the above form, where no  $p_i$  is a prefix of another.  $(p_1, \dots, p_k)$  is called a  $k$ -sequence. Intuitively, the tree-expression is the “prefix tree” of  $k$  “strings” given by  $p_1, \dots, p_k$  such that the left-to-right order of these strings is preserved. To construct the “prefix tree,” initially, the  $\top$  node of all paths  $p_i$  form the root of the tree-expression. Recursively, under each node all paths sharing the same next label  $l_i$  will go to a branch labeled  $l_i$ , provided that  $p_i$  is the  $i$ th root-to-leaf path from left to right in the final tree. The next example illustrates this construction.

**Example 3.1.** Consider the transaction object  $t$  defined as

$$\begin{aligned} val(\&t) &= \{Director : \&d, Cast : \&c\} \\ val(\&c) &= \{Actor : \&a_1, Invited\_Actor : \&a_2, Actor : \&a_3\} \\ val(\&a_2) &= \{Org : \&o_1, Nationality : \&o_2\} \end{aligned}$$

and consider two tree-expressions of  $t$ :

$$\begin{aligned} te_1 &= \{Cast : \{Invited\_Actor : \{Org : \perp, Nationality : \perp\}, \\ &\quad Actor : \perp\}\}, \\ te_2 &= \{Cast : \{? : \{Org : \perp, Nationality : \perp\}, Actor : \perp\}\}, \\ te_3 &= \{Cast : \{Invited\_Actor : \{Org : \perp\}, Actor : \perp, \\ &\quad Invited\_Actor : \{Nationality : \perp\}\}\}. \end{aligned}$$

As shown in Fig. 4,  $te_1$  is constructed by the three-sequence  $(p_1, p_2, p_3)$  (the first tree), and  $te_2$  by the three-sequence  $(p_4, p_5, p_3)$  (the second tree), and  $te_3$  by the three-sequence  $(p_1, p_3, p_2)$  (the third tree), where  $p_i$ s are path-expressions:

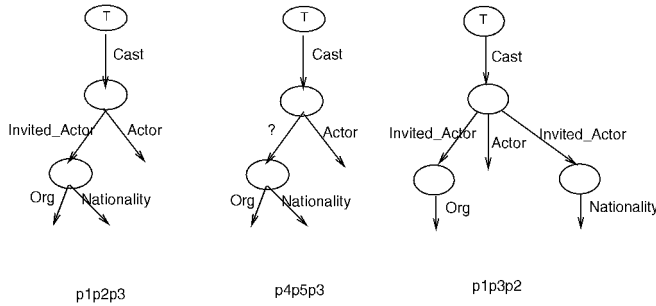


Fig. 4. Constructing tree-expressions.

$$\begin{aligned}
 p_1 &= [\top, \text{Cast}, \text{Invited\_Actor}, \text{Org}, \perp], \\
 p_2 &= [\top, \text{Cast}, \text{Invited\_Actor}, \text{Nationality}, \perp], \\
 p_3 &= [\top, \text{Cast}, \text{Actor}, \perp], \\
 p_4 &= [\top, \text{Cast}, ?, \text{Org}, \perp], \\
 p_5 &= [\top, \text{Cast}, ?, \text{Nationality}, \perp].
 \end{aligned}$$

Note that different orders  $(p_1, p_2, p_3)$  and  $(p_1, p_3, p_2)$  represent different tree-expressions, despite the fact that all values  $val(\&o)$  are bags. On the other hand,  $(p_1, p_2, p_3)$  and  $(p_2, p_1, p_3)$  represent the same tree-expression because the children of a bag node are not ordered.

However, the above representation suffers from two problems. The first problem is that some children with repeating labels cannot be constructed. For example, two-tree-expression  $\{\text{Cast} : \{\text{Actor} : \perp, \text{Actor} : \perp\}\}$ , which says that the movie has two actors, cannot be constructed by using path  $[\top, \text{Cast}, \text{Actor}, \perp]$  twice. This is because the construction does not know whether *Actor* labels in the two paths are for the same or different actors. We can solve this problem by superscripting repeating *Actor* label in  $val(\&c)$ ; instead of generating only one path  $[\top, \text{Cast}, \text{Actor}, \perp]$ , we generate two paths

$$[\top, \text{Cast}, \text{Actor}^1, \perp]$$

and

$$[\top, \text{Cast}, \text{Actor}^2, \perp],$$

to represent the first and second actors in  $val(\&c)$ , respectively. In general, for each label  $l$  in  $val(\&o)$ ,  $l^i$  represents the  $i$ th occurrence of  $l$  in  $val(\&o)$ . The maximal superscript  $i$  of  $l$  with respect to  $\&o$ , denoted  $Occur(\&o, l)$ , is the number of occurrences of  $l$  in  $val(\&o)$ . The second problem is that the wild card label  $?$  is not considered. To solve this problem, we add  $?$  to  $L(\&w, \&z)$  for each edge  $(\&w, \&z)$ .  $Occur(\&o, ?)$  is defined as the number of references to nonatomic objects in  $val(\&o)$ .

With these modifications, a  $k$ -tree-expression can now be constructed by a  $k$ -sequence  $(p_1, \dots, p_k)$ , each  $p_i$  of the form  $[\top, l_1^{j_1}, \dots, l_n^{j_n}, \perp]$  or  $[\top, l_1^{j_1}, \dots, l_n^{j_n}, \perp_i]$ , satisfying the following conditions:

1.  $(l_1, \dots, l_n)$  is in the cross product

$$L(\&t, \&w_1) \times \dots \times (L(\&w_{n-1}, \&w_n) - \{?\})$$

for some simple path  $\&t, \&w_1, \dots, \&w_n$  in  $G$  starting at some transaction object  $\&t$ ;

2. for  $1 \leq i \leq n$ , superscript  $j_i$  ranges from 1 to  $UP_i$ , where  $UP_i$  is the largest  $Occur(\&w_{i-1}, l_i)$  for all nodes  $\&w_{i-1}$  in condition 1;
3. no  $p_i$  is a prefix of another.

After the superscripting, we consider only  $k$ -tree-expressions in which superscripted labels  $l_i^{j_i}$  branching out of a node are distinct. Paths  $p_i$ s of the above form are called *path-expressions*. For the rest of the paper, the concatenation  $p_1 \dots p_k$  denotes the  $k$ -tree-expression constructed by the  $k$ -sequence  $(p_1, \dots, p_k)$ .

### 3.2 The Overview

The core of the algorithm is computing all  $k$ -sequences  $(p_1, \dots, p_k)$  such that  $p_1 \dots p_k$  are frequent tree-expressions. This set of  $k$ -sequences is denoted by  $F_k$ . Note that several  $k$ -sequences may construct the same tree-expression because the latter does not depend on superscripts of labels (as shown by  $(p_1, p_2)$  and  $(p_1, p_3)$  in Fig. 9), and thus, that  $F_k$  may contain redundant  $k$ -sequences as far as tree-expressions are concerned. We will deal with this problem in Section 3.5 by pruning the search space so that at most one  $k$ -sequence is generated for each frequent tree-expression. Until Section 3.5, we focus on finding all  $k$ -sequences  $(p_1, \dots, p_k)$  such that  $p_1 \dots p_k$  are frequent,  $k \geq 1$ . Obviously, searching the entire space of  $k$ -sequences is prohibitive. Fortunately, we do not need to examine a  $k$ -sequence if some “substructure” of it is known to be infrequent. This observation forms the foundation of our algorithm, which is stated as follows:

**Theorem 3.1 (The downward closure property).** *Let  $p_i$  denote a path-expression. If  $k$ -tree-expression  $p_1 \dots p_k$  is frequent, then any  $(k-1)$ -tree-expression*

$$p_1 \dots p_{i-1} p_{i+1} \dots p_k$$

*is frequent, where  $1 \leq i \leq k$ ; in particular,  $p_1 \dots p_{k-2} p_{k-1}$  and  $p_1 \dots p_{k-2} p_k$  are frequent.*

**Proof.** This follows because the  $(k-1)$ -tree-expressions are weaker than the  $k$ -tree-expression and because the weaker than relationship is transitive.  $\square$

Following Theorem 3.1, we compute  $F_k$  in the order of  $k$  in two phases. In Phase I, we make one pass over transaction objects to find all path-expressions  $p_i$  representing frequent one-tree-expressions, i.e.,  $F_1$ . In Phase II, in the  $k$ th ( $k > 1$ ) pass over transaction objects we generate a  $k$ -sequence  $(p_1, \dots, p_k)$  *only if*  $(k-1)$ -sequences  $(p_1, \dots, p_{k-2}, p_{k-1})$  and  $(p_1, \dots, p_{k-2}, p_k)$  are in  $F_{k-1}$ .  $(p_1, \dots, p_k)$  is only a *candidate*  $k$ -sequence because  $p_1 \dots p_k$  may not be frequent. We find  $F_k$  by computing the support of candidates in one scan of transaction objects. Phase II terminates when  $F_k$  is empty for some  $k$ . The search space can be pruned by ignoring the order of the children of a bag node. We will discuss this pruning in Section 3.5. For the maximal discovery problem, we need one additional phase, Phase III, to remove all nonmaximally frequent tree-expressions. In general, nonmaximally frequent tree-expressions, such as  $p_1 \dots p_{i-1} p_{i+1} \dots p_k$  if  $p_1 \dots p_k$  is frequent, cannot be removed

```

compute the support:
foreach transaction object  $\&t$  do
  foreach simple path  $\&t, \&w_1, \dots, \&w_n$  do
    foreach label sequence  $(l_1, \dots, l_n)$  in
       $L(\&t, \&w_1) \times L(\&w_1, \&w_2) \times \dots \times (L(\&w_{n-1}, \&w_n) - \{?\})$  do
        Case 1:  $\&w_n \neq \&w_i$  for all  $i$ 
          if  $sup(l_1, \dots, l_n, \perp)$  was not increased for  $\&t$  then  $sup(l_1, \dots, l_n, \perp) ++$ 
        Case 2:  $\&w_n = \&w_i$  for some  $i < n$ 
          if  $sup(l_1, \dots, l_n, \perp_i)$  was not increased for  $\&t$  then  $sup(l_1, \dots, l_n, \perp_i) ++$ 
    return frequent path-expressions:
    foreach  $sup(l_1, \dots, l_n, \perp)$  or  $sup(l_1, \dots, l_n, \perp_i)$  not less than  $MINISUP$  do
      output path-expressions  $[\top, l_1^{j_1}, \dots, l_n^{j_n}, \perp]$  or  $[\top, l_1^{j_1}, \dots, l_n^{j_n}, \perp_i]$ ,  $1 \leq j_i \leq UP_i$ 

```

Fig. 5. Computing  $F_i$ 

immediately because they are needed to generate maximally frequent tree-expressions, such as  $p_1 \dots p_k$ . However, we will identify one special case where some nonmaximally frequent tree-expressions can be removed before the end of Phase II.

At this point, the above computation seems similar to a priori in [3] that is based on the subset property in [3]: An itemset  $\{i_1, \dots, i_k\}$  is frequent only if both  $\{i_1, \dots, i_{k-2}, i_{k-1}\}$  and  $\{i_1, \dots, i_{k-2}, i_k\}$  are frequent. The reader may wonder why not simply map each tree-expression  $p_1 \dots p_k$  to itemset  $\{p_1, \dots, p_k\}$ , by considering each  $p_i$  as an item, and apply a priori to solve the problem at hand. Unfortunately, this “reduction” does not work for the following reasons. First,  $p_1 \dots p_k$  is weaker than  $p'_1 \dots p'_m$  does not imply  $\{p_1, \dots, p_k\}$  is a subset of  $\{p'_1, \dots, p'_m\}$ ; consequently,  $p_1 \dots p_k$  may be frequent, but itemset  $\{p_1, \dots, p_k\}$  is not. For example, in Fig. 4,  $p_4 p_5 p_3$  is weaker than  $p_1 p_2 p_3$ , but  $\{p_4, p_5, p_3\}$  is not contained in  $\{p_1, p_2, p_3\}$ . This example also shows that it does not work either to map tree-expression  $p_1 \dots p_k$  to sequence  $(p_1, \dots, p_k)$  of items  $p_i$  and replace the weaker than relationship with the subsequence containment. We use  $k$ -sequences  $(p_1, \dots, p_k)$  only as a representation of tree-expressions; to decide if it generalizes an object, the

represented tree-expression and the weaker than relationship must be used.

### 3.3 Phase I: Computing $F_1$

This phase finds all one-sequences  $p_i$  representing frequent one-tree-expressions in the form of path-expressions  $[\top, l_1^{j_1}, \dots, l_n^{j_n}, \perp]$  or  $[\top, l_1^{j_1}, \dots, l_n^{j_n}, \perp_i]$ . These one-sequences are later used to construct  $k$ -tree-expressions  $p_1 \dots p_k$ , as discussed in Section 3.1. The first question is how to compute the support of a path-expression. It is important to note that all path-expressions that differ only in superscripts of labels represent the same one-tree-expression. Therefore, the support of path-expression  $[\top, l_1^{j_1}, \dots, l_n^{j_n}, \perp]$  or  $[\top, l_1^{j_1}, \dots, l_n^{j_n}, \perp_i]$  should be associated with the sequence  $l_1, \dots, l_n, \perp$  or  $l_1, \dots, l_n, \perp_i$ . We denote this support by  $sup(l_1, \dots, l_n, \perp)$  or  $sup(l_1, \dots, l_n, \perp_i)$ , defined as the number of transaction objects from which there is a simple path labeled  $l_1, \dots, l_n$ . Fig. 5 gives the computation of  $F_1$ .  $UP_i$  is the largest  $Occur(\&w_{i-1}, l_i)$  for all simple paths  $\&t, \&w_1, \dots, \&w_n$  that are labeled  $l_1, \dots, l_n$ , where  $\&t$  is a transaction object. We have omitted the computation of  $UP_i$  for clarity.

**Example 3.2.** For the rest of this section, we use the OEM graph in Fig. 6 to illustrate the discovery algorithm. Recall that a circled node denotes a bag and a squared node denotes a list. Suppose that  $\&t_1$  and  $\&t_2$  are transaction objects, containing information about two electronic shopping transactions. For example,  $\&t_1$  consists of subtransaction  $\&a$  followed by a purchase of item  $\&o_1$  in cash.  $\&a$  consists of two purchases of  $\&o_1$  in any order, one by credit card and the other in cash. The definitions of  $\&t_1$  and  $\&t_2$  are given by:

$$\begin{aligned}
val(\&t_1) &= \langle Unused : \&a, Cash : \&o_1 \rangle \\
val(\&t_2) &= \langle Unused : \&b, Cash : \&o_1 \rangle \\
val(\&a) &= \{Credit : \&o_1, Cash : \&o_1\} \\
val(\&b) &= \{Credit : \&o_1, Cash : \&o_1, Cash : \&o_2\}.
\end{aligned}$$

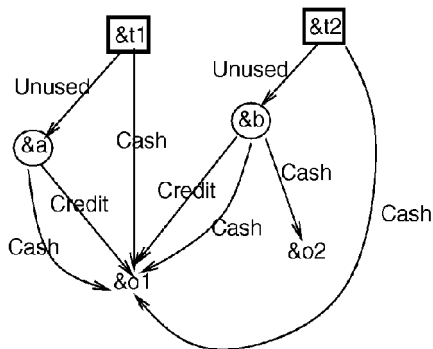


Fig. 6. Example 3.2.

TABLE 1  
 $F_1$  in Example 3.2

path-expressions	tree-expressions represented
$p_1 : [\top, Cash^1, \perp]$	$\langle Cash : \perp \rangle$
$p_2 : [\top, ?^1, Credit^1, \perp]$	$\langle ? : \{Credit : \perp\} \rangle$
$p_3 : [\top, ?^1, Cash^1, \perp]$	$\langle ? : \{Cash : \perp\} \rangle$
$p_4 : [\top, ?^1, Cash^2, \perp]$ (pruned)	$\langle ? : \{Cash : \perp\} \rangle$
$p_5 : [\top, Unused^1, Credit^1, \perp]$	$\langle Unused : \{Credit : \perp\} \rangle$
$p_6 : [\top, Unused^1, Cash^1, \perp]$	$\langle Unused : \{Cash : \perp\} \rangle$
$p_7 : [\top, Unused^1, Cash^2, \perp]$ (pruned)	$\langle Unused : \{Cash : \perp\} \rangle$

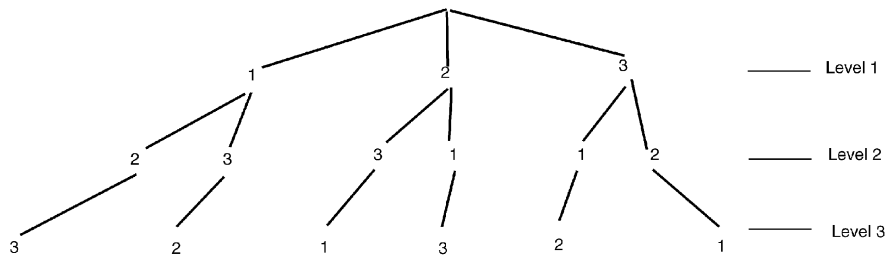


Fig. 7.  $\Pi_1, \Pi_2,$  and  $\Pi_3$ .

$Occur(\&b, Cash) = 2$  because  $Cash$  occurs in  $val(\&b)$  twice;  $Occur(\&t_1, ?) = Occur(\&t_2, ?) = 1$  because there is only one nonatomic object in  $val(\&t_1)$  and  $val(\&t_2)$ . Suppose that  $MINISUP = 2/2$ . Path-expressions  $p_1$  through  $p_7$  are frequent, as shown in Table 1. For example,  $sup(Unused, Cash, \perp) = 2$  because both transaction objects have a simple path labeled  $Unused, Cash$ . From this support,  $p_6$  and  $p_7$  are generated because among all paths of the form

$$\&t_i, Unused, w_1, Cash, w_2,$$

the largest  $Occur(\&t_i, Unused)$  is 1 and the largest  $Occur(\&w_1, Cash)$  is 2 (i.e., when  $\&w_1 = \&b$ ). The other frequent path-expressions are similarly generated.  $p_4$  and  $p_7$  will not be included in the final  $F_1$  by the pruning strategies to be discussed in Section 3.5.

### 3.4 Phase II: Computing $F_k$

**The search space.** Following Theorem 3.1, the storage structure of  $F_{k-1}$  should facilitate efficient retrieval of pairs  $(p_1, \dots, p_{k-2}, p_{k-1})$  and  $(p_1, \dots, p_{k-2}, p_k)$  and in addition, dynamically grow from  $F_{k-1}$  to  $F_k$  without reorganization. We propose the  $(k-1)$ -candidate-trie, denoted  $\Pi_{k-1}$ , to meet these requirements.  $\Pi_{k-1}$  is a trie of maximal depth  $k-1$ . (A trie is a tree in which each nonleaf node has at least one child.) In  $\Pi_{k-1}$ , each nonroot node represents a path-expression  $p_i$  in  $F_1$  and each path  $(root, p_1, \dots, p_j)$  represents a  $j$ -sequence  $(p_1, \dots, p_j)$  in  $F_j$ . Without confusion, we omit the  $root$  node and use the  $j$ -sequence  $(p_1, \dots, p_j)$  to refer to such paths in  $\Pi_{k-1}$ . Consequently, each nonroot node in  $\Pi_{k-1}$  represents two things: the path-expression at the node and the  $j$ -sequence ending at the node. We will freely speak

of terms like “frequent  $j$ -sequences,” “maximally frequent  $j$ -sequences,” “the support of  $j$ -sequences,” and “some  $j$ -sequences weaker than others,” with the obvious understanding that these refer to the tree-expressions represented by the  $j$ -sequences. The following corollary follows from our representation of search space.

**Corollary 3.1.** *The pair  $p_1 \dots p_{k-2} p_{k-1}$  and  $p_1 \dots p_{k-2} p_k$  in Theorem 3.1 is represented by two  $(k-1)$ -sequences ending at sibling leaf nodes in  $\Pi_{k-1}$ .*

**Generating candidates.** Following Corollary 3.1, to generate  $\Pi_k$  from  $\Pi_{k-1}$  we consider every pair of  $(k-1)$ -sequences  $(p_1, \dots, p_{k-2}, p_{k-1})$  and  $(p_1, \dots, p_{k-2}, p_k)$ , ending at sibling leaf nodes  $l$  and  $l'$  in  $\Pi_{k-1}$ , and create a child under  $l$  to represent the  $k$ -sequence  $(p_1, \dots, p_{k-1}, p_k)$ . We say that  $l$  is extended by  $l'$ , or that  $(p_1, \dots, p_{k-2}, p_{k-1})$  is extended by  $(p_1, \dots, p_{k-2}, p_k)$ . We also say that  $(p_1, \dots, p_{k-2}, p_{k-1})$  and  $(p_1, \dots, p_{k-2}, p_k)$  are used in this extension. Fig. 7 shows  $\Pi_1, \Pi_2, \Pi_3$  generated by three path-expressions  $p_1, p_2, p_3$  without any pruning. We will address the pruning of search space shortly.

**Counting the support.** Fig. 8 shows a conceptual computation of the support of  $k$ -sequences in  $\Pi_k$ . For each transaction object  $t$ , we read the hierarchy of  $t$ , examines each  $k$ -sequence and increases its support if it is weaker than  $\&t$ . In implementation, we use  $\Pi_k$  to prune scans of  $k$ -sequences—we traverse  $\Pi_k$  in a depth-first manner, and if  $p_1 \dots p_j$  for the current  $j$ -sequence  $(p_1, \dots, p_j)$  is not weaker than  $\&t$ , further descending into the tree can be pruned. Since this implementation is straightforward, we do not elaborate on it further.

```

foreach transaction object  $\&t$  do
  foreach  $k$ -sequence  $(p_1, \dots, p_k)$  in  $\Pi_k$  do
    if  $p_1 \dots p_k$  is weaker than  $\&t$ 
      then increase the support for  $(p_1, \dots, p_k)$ 
    foreach  $k$ -sequence  $(p_1, \dots, p_k)$  in  $\Pi_k$  do
      if the support for  $(p_1, \dots, p_k)$  is less than  $MINISUP$ 
        then delete the leaf node for  $(p_1, \dots, p_k)$  from  $\Pi_k$ 

```

Fig. 8. Computing the support of  $k$ -sequences.

### 3.5 Pruning of Search Space

Phase II, described above, faces two problems that seriously affect the efficiency and scalability of the algorithm. First, the search space  $\Pi_k$  grows very fast, as illustrated in Fig. 7. Second, all  $k$ -sequences representing the same frequent tree-expression are generated. For example, both  $(p_1, p_2)$  and  $(p_1, p_3)$  in Fig. 9 will be generated, though both represent the same tree-expression, i.e.,  $\{l : \{l : \perp, l : \perp\}\}$ . We now address these issues by pruning the search space. Recall that a  $k$ -tree-expression is constructed by  $k$ -sequence  $(p_1, \dots, p_k)$ , where each  $p_i$  is a path-expression of the form  $[\top, l_1^{j_1}, \dots, l_n^{j_n}, \perp]$  or  $[\top, l_1^{j_1}, \dots, l_n^{j_n}, \perp]$ . Superscripts  $j_i$  serves to create repeating labels for child nodes in a tree-expression; however, once the tree-expression is constructed, superscripts are not useful anymore and can be ignored. As a result, several  $k$ -sequences could construct the same  $k$ -tree-expression (up to ignoring the superscripts of labels), and it suffices to consider only one of these  $k$ -sequences. What we need is a systematic method that refers to those  $k$ -sequences that need to be considered. The idea is to impose certain conditions on superscripts of labels in the tree-expressions constructed. This motivates the following definitions.

Consider a tree-expression  $p_1 \dots p_k$  constructed by  $k$ -sequence  $(p_1, \dots, p_k)$ . A list node is *monotone* if all outgoing labels  $l^i$  for the same  $l$  are strictly ordered by  $i$  from left to right. A bag node is *monotone* if all outgoing labels  $l^i$  are strictly ordered by the lexicographic order of  $(l, i)$  from left to right. In other words, for a list node we order only repeating occurrences of labels, but for a bag node we order both labels and repeating occurrences. A (list or bag) node is *natural* if it is monotone and each outgoing label  $l^i$  is the  $i$ th occurrence of  $l$  from left to right. A

$k$ -sequence  $(p_1, \dots, p_k)$  is *natural* (*monotone*, resp) if every nonleaf node in tree-expression  $p_1 \dots p_k$  is natural (monotone, resp). For example, in Fig. 9,  $(p_1, p_2)$  and  $(p_1, p_2, p_3)$  are natural;  $(p_1, p_3)$  and  $(p_1, p_2, p_4)$  are monotone but nonnatural;  $(p_3, p_1)$  is nonmonotone.

Several observations are useful for the subsequent discussion.

**Observation I.** For every nonnatural  $k$ -sequence, there is a natural  $k$ -sequence that represents the same tree-expression. Consequently, a search is complete if all frequent natural  $k$ -sequences are generated.

**Observation II.** Every prefix of a natural (monotone)  $k$ -sequence is natural (monotone).

**Observation III.** Every permutation of a natural (monotone)  $k$ -sequence is not natural (monotone). This implies that there are much more nonnatural (nonmonotone)  $k$ -sequences than natural (monotone) ones. Therefore, if we can prune all nonnatural or nonmonotone  $k$ -sequences, the search space will be substantially reduced.

However, simply pruning all nonnatural  $k$ -sequences does not work if we use Theorem 3.1 to generate candidate sequences. In fact, some nonnatural  $(k-1)$ -sequences  $(p_1, \dots, p_{k-2}, p_k)$  must be generated in order to generate natural  $k$ -sequences  $(p_1, \dots, p_{k-1}, p_k)$ . For example, in Fig. 9, to generate natural  $(p_1, p_2, p_3)$ , we first need to generate natural  $(p_1, p_2)$  and nonnatural  $(p_1, p_3)$ . On the other hand, from Observation II, extending a nonnatural  $(k-1)$ -sequence  $(p_1, \dots, p_{k-1})$  always generates a nonnatural  $k$ -sequence  $(p_1, \dots, p_k)$ . For a similar reason, the result of such an extension cannot be used to generate a natural  $j$ -sequence,  $j > k$ . This gives us the first pruning strategy, concerning what  $(k-1)$ -sequences should be extended.

**Strategy I.** In the  $k$ th pass, only natural  $(k-1)$ -sequences should be extended. After all extensions in the  $k$ th pass, all nonnatural  $(k-1)$ -sequences can be pruned.

Since there are a lot more nonnatural  $(k-1)$ -sequences than natural ones (Observation III), Strategy I prunes most extensions at each level. Next, we would like to characterize  $k$ -sequences  $(p_1, \dots, p_k)$  that should be generated. First of all, all natural  $(p_1, \dots, p_k)$  should be generated for the completeness of search. Second, a nonnatural  $(p_1, \dots, p_k)$  should be generated if it is useful for extending a natural

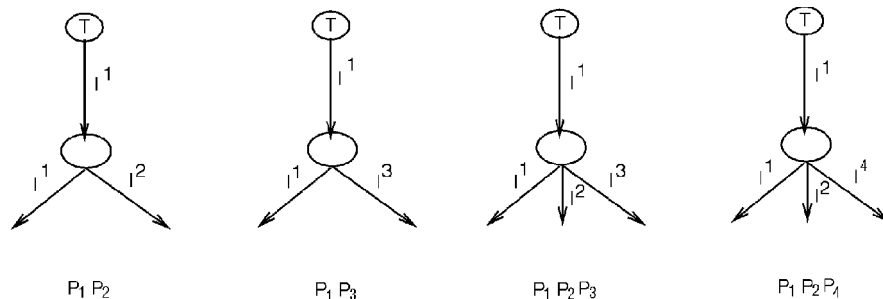


Fig. 9. Constructing natural by nonnatural.

```

generate  $k$ -sequences:
  foreach natural  $(k - 1)$ -sequence  $(p_1, \dots, p_{k-2}, p_{k-1})$  in  $\Pi_{k-1}$  do /* Strategy I */
    foreach  $(k - 1)$ -sequence  $(p_1, \dots, p_{k-2}, p_k)$  in  $\Pi_{k-1}$  do
      if  $p_{k-1}$  and  $p_k$  are not a prefix of each other and
         $k$ -sequence  $(p_1, \dots, p_{k-1}, p_k)$  is near-natural /* Strategy II */
      then extend  $(p_1, \dots, p_{k-2}, p_{k-1})$  by  $(p_1, \dots, p_{k-2}, p_k)$ ;
      delete all leaf nodes representing non-natural  $(k - 1)$ -sequences; /* Strategy I */
compute the support of  $k$ -sequences:
  foreach transaction object  $\&t$  do
    foreach  $k$ -sequence  $(p_1, \dots, p_k)$  in  $\Pi_k$  do
      if  $p_1 \dots p_k$  is weaker than  $\&t$  then increase the support for  $(p_1, \dots, p_k)$ 
    foreach  $k$ -sequence  $(p_1, \dots, p_k)$  do
      if the support for  $(p_1, \dots, p_k)$  is less than  $MINISUP$ 
      then delete the leaf node representing  $(p_1, \dots, p_k)$  from  $\Pi_k$ 
      else mark  $(p_1, \dots, p_{k-2}, p_{k-1})$  and  $(p_1, \dots, p_{k-2}, p_k)$  as used; /* Strategy III */

```

Fig. 10. Generating  $\Pi_k$  from  $\Pi_{k-1}$ .

$(k - 1)$ -sequence. In this case, the prefix  $(p_1, \dots, p_{k-1})$  must be natural because it is shared with a natural  $(k - 1)$ -sequence. Third, a nonnatural  $(p_1, \dots, p_k)$  should be generated if it can be used to generate a natural  $j$ -sequence, in one or more extensions. From Observation II, such  $(p_1, \dots, p_k)$  must be monotone. These three cases are summarized by the notion of near-natural sequences: A  $k$ -sequence  $(p_1, \dots, p_k)$  is *near-natural* if  $(p_1, \dots, p_k)$  is monotone and  $(p_1, \dots, p_{k-1})$  is natural. Every natural  $k$ -sequence is near-natural, but not vice versa. In Fig. 9, all  $k$ -sequences are near-natural; only  $(p_1, p_2)$  and  $(p_1, p_2, p_3)$  are natural; any permutation of these sequences is not nearly-natural (because not monotone). Now, we have the second pruning strategy concerning what  $k$ -sequences should be generated.

**Strategy II.** Only near-natural  $k$ -sequences should be generated.

Observation III implies that Strategy II prunes most extensions at a level because every nonmonotone  $k$ -sequence is not near-natural. Strategies I and II together imply that the only type of extensions that we need to consider are extending a natural sequence with a near-natural sequence. The next pruning strategy applies only to the maximal discovery problem. The idea is to prune a nonmaximally frequent candidate if it is not useful in any later extension. Suppose that a frequent  $k$ -sequence  $(p_1, \dots, p_k)$  is generated by extending

$$(p_1, \dots, p_{k-2}, p_{k-1})$$

by

$$(p_1, \dots, p_{k-2}, p_k).$$

Since both  $(p_1, \dots, p_{k-2}, p_{k-1})$  and  $(p_1, \dots, p_{k-2}, p_k)$  are weaker than  $(p_1, \dots, p_k)$ , they are nonmaximally frequent.

Further, these  $(k - 1)$ -sequences will not be used in any extension after the  $k$ th pass. This gives us the following pruning strategy.

**Strategy III.** For the maximal discovery problem, a  $(k - 1)$ -sequence that is used to generate at least one frequent  $k$ -sequence can be pruned.

This strategy prunes all  $(k - 1)$ -sequences ending at nonleaf nodes because non-leaf nodes are used to generate their child nodes. It also prunes all  $(k - 1)$ -sequences at leaf nodes that are used to extend their sibling nodes. In Section 4, we will experimentally verify the effectiveness of Strategies I, II, and III. Fig. 10 summarizes the generation of  $\Pi_k$  from  $\Pi_{k-1}$ . The following theorem follows from the above discussion.

**Theorem 3.2.** Assume that  $\Pi_k$  is the candidate-trie at the end of Phase II and that  $1 \leq j \leq k$ .

- Let  $F_j$  be the set of  $j$ -sequences in  $\Pi_k$  that are not pruned by Strategies II and III. Then,  $F_j$  contains exactly the  $j$ -tree-expressions for the discovery problem.
- Let  $F_j$  be the set of  $j$ -sequences in  $\Pi_k$  that are not pruned by Strategies I, II, III. Then,  $F_j$  contains all (possibly more)  $j$ -tree-expressions for the maximal discovery problem.

We now show that each  $j$ -sequence in  $F_j$  computed in Theorem 3.2 represents a unique tree-expression. Importantly, this implies that no tree-expression is generated more than once.

**Theorem 3.3.** For any two distinct  $j$ -sequences  $(p_1, \dots, p_j)$  and  $(p'_1, \dots, p'_j)$  in  $F_j$  computed in Theorem 3.2, tree-expressions  $p_1 \dots p_j$  and  $p'_1 \dots p'_j$  are distinct.

**Proof.** First, observe that all  $j$ -sequences in  $F_j$  are natural because nonnatural ones are pruned by Strategy I. Suppose that  $j$ -sequences  $(p_1, \dots, p_j)$  and  $(p'_1, \dots, p'_j)$  in  $F_j$  represent the same tree-expression (after ignoring superscripts of labels). Consider corresponding nodes  $u$  and  $u'$  in  $p_1 \dots p_j$  and  $p'_1 \dots p'_j$ . Let  $l_1^{j_1}, \dots, l_n^{j_n}$ , from left to right, be the outgoing labels at  $u$  and  $l'_1, \dots, l'_n$  be the outgoing labels at  $u'$ . Clearly,  $l_i = l'_i$ , for  $1 \leq i \leq n$ . Since both  $(p_1, \dots, p_j)$  and  $(p'_1, \dots, p'_j)$  are natural, there is only one "natural" superscripting of labels, so  $j_i = j'_i$ , for  $1 \leq i \leq n$ . This implies that path-expressions  $p_i$  and  $p'_i$  are identical for  $1 \leq i \leq n$ , contradicting the assumption that  $(p_1, \dots, p_j)$  and  $(p'_1, \dots, p'_j)$  are distinct.  $\square$

**Example 3.3.** Continue with Example 3.2 where  $MINISUP = 2/2$ . Fig. 11a shows  $\Pi_1, \Pi_2, \Pi_3$ , corresponding to the portion above levels 1, 2, and 3, respectively. Please refer to Table 1 for frequent one-sequences  $p_i$ ,  $1 \leq i \leq 7$ . Here is the generation of  $\Pi_2$  from  $\Pi_1$ .

- Extensions of  $p_1$ :  $p_1$  is not extended because all its extensions are not frequent. In fact, *Cash* does not appear on the left side of any label in either  $val(\&t_1)$  or  $val(\&t_2)$ .
- Extensions of  $p_2$ :  $(p_2, p_1)$  is generated.  $(p_2, p_3)$  and  $(p_2, p_4)$  are not generated because they are nonmonotone (Strategy II).  $(p_2, p_5)$ ,  $(p_2, p_6)$ , and  $(p_2, p_7)$  are not frequent.
- Extensions of  $p_3$ :  $(p_3, p_1)$  and  $(p_3, p_2)$  are generated.  $(p_3, p_4)$ ,  $(p_3, p_5)$ ,  $(p_3, p_6)$ , and  $(p_3, p_7)$  are not frequent.
- Extensions of  $p_4$ :  $p_4$  is not extended because it is nonnatural (Strategy I).
- Extensions of  $p_5$ :  $(p_5, p_1)$  is generated.  $(p_5, p_2)$ ,  $(p_5, p_3)$ , and  $(p_5, p_4)$  are not frequent;  $(p_5, p_6)$  and  $(p_5, p_7)$  are nonmonotone (Strategy II).
- Extensions of  $p_6$ :  $(p_6, p_1)$  and  $(p_6, p_5)$  are generated.  $(p_6, p_2)$ ,  $(p_6, p_3)$ ,  $(p_6, p_4)$ ,  $(p_6, p_7)$  are not frequent.
- Extensions of  $p_7$ :  $p_7$  is not extended because it is nonnatural (Strategy I).

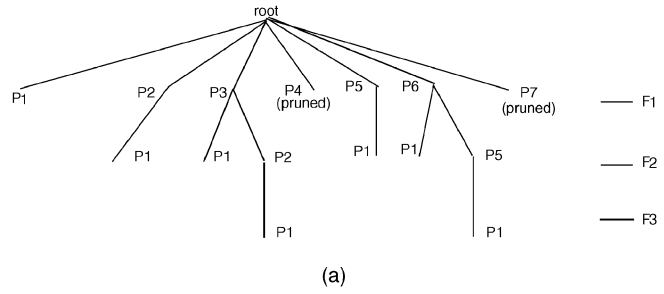
After two-sequences are generated, nonnatural  $p_4$  and  $p_7$  are pruned from  $\Pi_2$  by Strategy I.

The generation of  $\Pi_3$  from  $\Pi_2$  follows as:

- Extensions of  $(p_3, p_1)$ :  $(p_3, p_1, p_2)$  is not frequent (nor near-natural).
- Extensions of  $(p_3, p_2)$ :  $(p_3, p_2, p_1)$  is generated.
- Extensions of  $(p_6, p_1)$ :  $(p_6, p_1, p_5)$  is not frequent (nor near-natural).
- $(p_6, p_5)$ :  $(p_6, p_5, p_1)$  is generated.

Fig. 11b shows  $F_2$  and  $F_3$  and the tree-expressions represented. Fig. 11c shows their tree representations. At this stage,  $F_1, F_2, F_3$  are returned for the discovery problem.

For the maximal discovery problem,  $F_1$  is empty because each one-sequence is either pruned by Strategy I (i.e.,  $p_4$  and  $p_7$ ) or marked as *used* (i.e.,  $p_1, p_2, p_3, p_5, p_6$ ).  $F_2$  contains only  $(p_2, p_1)$  and  $(p_5, p_1)$  because  $(p_3, p_1), (p_3, p_2),$



sequences	tree-expressions
<b>F<sub>2</sub></b>	
$p_2p_1$	$\langle ? : \{Credit : \perp\}, Cash : \perp \rangle$
$p_3p_1$	$\langle ? : \{Cash : \perp\}, Cash : \perp \rangle$
$p_3p_2$	$\langle ? : \{Cash : \perp, Credit : \perp\} \rangle$
$p_5p_1$	$\langle Unused : \{Credit : \perp\}, Cash : \perp \rangle$
$p_6p_1$	$\langle Unused : \{Cash : \perp\}, Cash : \perp \rangle$
$p_6p_5$	$\langle Unused : \{Cash : \perp, Credit : \perp\} \rangle$
<b>F<sub>3</sub></b>	
$p_3p_2p_1$	$\langle ? : \{Cash : \perp, Credit : \perp\}, Cash : \perp \rangle$
$p_6p_5p_1$	$\langle Unused : \{Cash : \perp, Credit : \perp\}, Cash : \perp \rangle$

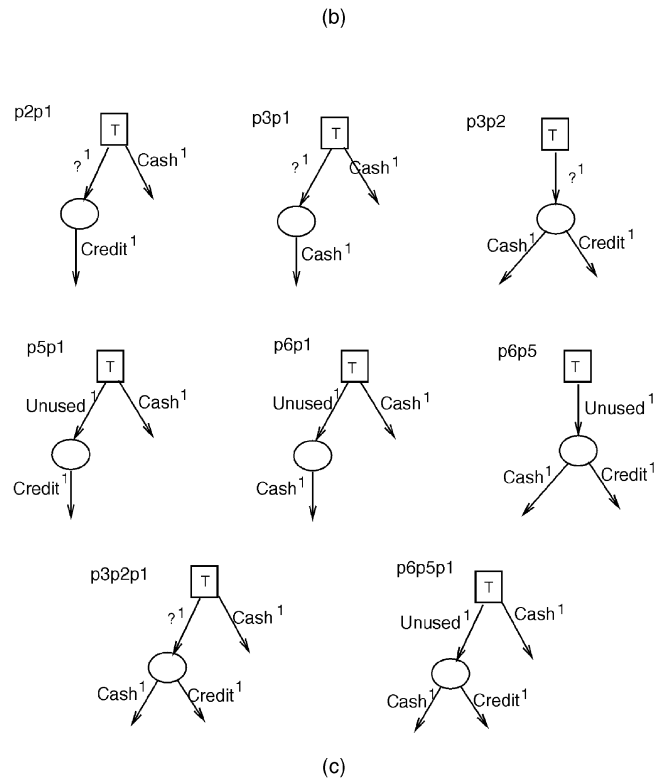


Fig. 11. Example 3.3. (a)  $\Pi_1, \Pi_2,$  and  $\Pi_3$ , (b)  $F_2$  and  $F_3$ , and (c) Tree representation.

$(p_6, p_1)$ , and  $(p_6, p_5)$  are marked as *used*.  $F_3$  contains  $(p_3, p_2, p_1)$  and  $(p_6, p_5, p_1)$ .

### 3.6 Phase III: The Maximal Phase

For the maximal discovery problem, we must remove remaining nonmaximally frequent sequences. One observa-

```

foreach  $j$  from  $k$  to 1 do
  let  $M_j$  be  $F_j$ 
  foreach sequence  $s$  in  $M_j$  do
    if  $s$  is weaker than some sequence in  $M_j$  then remove  $s$  from  $M_j$ 
foreach  $j$  from  $k$  to 1 do
  foreach sequence  $s$  in  $M_j$  do
    if  $s$  is not weaker than any sequence already output then output  $s$ 

```

Fig. 12. The maximal phase.

tion is that, for  $i > j$ , no  $i$ -sequence can be weaker than a  $j$ -sequence. This suggests the following pruning: For each  $1 \leq j \leq k$ , we find  $j$ -sequences in  $F_j$  that are maximally frequent with respect to  $F_j$ . Let this result be  $M_j$ . Then for  $j$ , from  $k$  to 1 in that order, we add a  $j$ -sequence in  $M_j$  to the final result only if it is not weaker than any sequence already in the final result. Fig. 12 shows this computation.

**Example 3.4.** Continue with Example 3.3. From that example,  $F_1$  is empty,  $F_2$  contains  $(p_2, p_1)$  and  $(p_5, p_1)$ , and  $F_3$  contains  $(p_3, p_2, p_1)$  and  $(p_6, p_5, p_1)$ . After removing nonmaximally frequent sequences in  $F_j$ ,  $M_1$  is empty,  $M_2$  contains  $(p_5, p_1)$ , and  $M_3$  contains  $(p_6, p_5, p_1)$ . Since  $(p_5, p_1)$  is weaker than  $(p_6, p_5, p_1)$ , tree-expression  $\langle \text{Unused} : \{ \text{Cash} : \perp, \text{Credit} : \perp \}, \text{Cash} : \perp \rangle$ , represented by  $(p_6, p_5, p_1)$ , is the answer to the maximal discovery problem.

### 3.7 Testing “Weaker Than”

It remains to be seen how to test whether a tree-expression  $te_1$  is weaker than a tree-expression  $te_2$  (as defined in Section 2). Basically, we need to search for a “match” of the tree  $te_1$  inside the tree  $te_2$ , such that the root of  $te_1$  matches the root of  $te_2$ . Recursively, a match is found for a nonleaf node  $v$  in  $te_1$  if matches are found for the label of  $v$  (ignoring superscripts) and for all subnodes of  $v$ . An additional requirement is that a node matches only a node of the same type (i.e., list or bag). For a bag node in  $te_1$ , a complete bipartite match in  $te_2$  is required, whereas for a list node in  $te_1$ , a sublist match in  $te_2$  is required. Since algorithms for finding subtree matches are well-known [11], we omit the detail. Assume that  $te_1$  has  $n$  nodes and  $te_2$  has  $m$  nodes. The time complexity of testing whether  $te_1$  is weaker than  $te_2$  is  $O(nm^{1.5})$  or better, depending on how good an algorithm one has for a complete bipartite matching [11]. This complexity, however, does not affect the I/O cost because the testing is done in memory.

## 4 EFFICIENCY

We now study the efficiency of the algorithm. The efficiency depends not only on database size, but also on factors such as minimum support and pruning strategies. Therefore, it is difficult to derive a closed, tight bound on the computational cost. On the other hand, the worst-case analysis assuming that nearly everything is frequent is far from

typical cases, thus, of little value. We take a more practical approach by analyzing the I/O scan of the database and studying experimentally other factors of the cost for various data characteristics. These factors include: the size of search space expanded, execution time, effectiveness of pruning strategies, and scalability for large databases.

### 4.1 I/O Scan

To analyze the I/O scan, we assume that the OEM graph (i.e., the database) is stored on disk and that the candidate-tree  $\Pi_k$  is stored in memory. The choice of storing  $\Pi_k$  in memory is based on the following reasons: The minimum support that defines a “typical” substructures is specified by the user and is often highly effective in restricting the search space. In the case of a “very small” minimum support, many substructures could become frequent. But this is also the case where the user should question the usefulness of such a large amount of “typical” substructures. Our view is that any substructures that cannot fit in a modern computer memory will not be comprehensible to a human user. If this happens, the user should rise the minimum support to reduce the number of typical substructures.

In Phase I, the hierarchy of each transaction object is read once. Similarly, in each pass of Phase II, the hierarchy of each transaction object is read to compute the support of candidates. Phase III does not read transaction objects. Assuming that  $k$  is the number of passes in Phase II, there are  $k+1$  scans of hierarchies of transaction objects. Our experiments show that  $k$  is typically small, i.e., 3 or 4. Therefore, our algorithm has a linear I/O cost. To reduce the number of page accesses, we can store frequently accessed nodes, called *hot spots*, in memory and leave infrequently accessed nodes on disk. This can be implemented by *pinning* the “hot spots” in memory so that they are not selected for page replacement by the buffer manager. Hot-spots usually have large in-degrees and/or are buried at lower levels in the graph. Another heuristic is to store nodes in an order “close” to the depth-first order in which nodes are traversed in our algorithm, so as to ensure that one page access can bring in several nodes that will be needed subsequently.

### 4.2 Experimental Study

To have a feel of the real performance of the algorithm, we have conducted many experiments for various data

TABLE 2  
Parameters

notation	meaning
$L_i$	number of level- $i$ labels
$N_i$	number of level- $i$ identifiers
$T_i$	average size of $val(\&o)$ for level- $i$ identifiers $\&o$
$I_i$	average size of potentially large sets in $\Gamma_i$
$P_i$	number of potentially large sets in $\Gamma_i$
$m$	maximal nesting level

characteristics and minimum supports. We focus on four indicators of efficiency: size of search space, effectiveness of pruning strategies, execution time, and scalability for large databases. We, also, consider data characteristics such as similarity of objects, number of objects, number of labels, depth of nesting, and size of datasets.

**Dataset generation.** We consider only acyclic datasets because only simple paths of a cyclic OEM graph are traversed. To model similarities of objects, we borrow from [3] the concept of *potentially large sets*. Informally, potentially large sets are itemsets that are more likely to contain common items than a random case. This property is attained by choosing items in a potentially large set in a controlled manner. For more details, please refer to [3]. However, we have to deal with nesting, labeling, and bag and list types of objects, all having impacts on the data mining problem. The idea is to treat subobject references as supermarket items and construct objects at higher levels using bags or lists of such items. At first, all atomic objects are at *level 1*. An object  $o$  is at *level*  $l + 1$  if  $l$  is the maximal level of subobjects of  $o$ . Let  $m$  be the maximal level of nontransaction objects. All transaction objects are at *level*  $m + 1$ .

Documents are generated in a bottom-up manner, from level 1 to level  $m + 1$ . At level  $i$ , we treat each subobject reference  $l : \&o$  at level  $i - 1$  as an item and construct a level- $i$  object as a bag or list (half-half in our case) of such items, as in [3]. This is done by picking several potentially large sets from the pool  $\Gamma_1 \cup \dots \cup \Gamma_{i-1}$ , at least one from  $\Gamma_{i-1}$ , where  $\Gamma_j$  is the set of potentially large sets for level  $j$ . Refer to Table 2 for notation of parameters. As in [3], overlapping of objects is controlled by parameters  $I_i$  and  $P_i$ . Each level- $i$  object constructed is assigned a new identifier. Subobject references  $l : \&o$  at level  $i$  are created by assigning each label  $l$  to some number of level- $i$  identifiers  $\&o$ , determined from the Poisson distribution with mean  $N_i/L_i$ . We then construct the set of potentially large sets  $\Gamma_i$  for level  $i$ . The above processing is repeated until transaction objects at level  $m + 1$  are constructed.

We use the following convention to represent a dataset:

$$(L_1, N_1)(L_2, N_2, T_2, I_2, P_2) \dots \\ (L_m, N_m, T_m, I_m, P_m)(N_{m+1}, T_{m+1}, I_{m+1}, P_{m+1}).$$

The first group  $(L_1, N_1)$  are parameters for level-1 labels and atomic objects. The last group  $(N_{m+1}, T_{m+1}, I_{m+1}, P_{m+1})$  are parameters for transaction objects.  $L_{m+1}$  is not used

because transaction objects have no label.  $(L_i, N_i, T_i, I_i, P_i)$ ,  $2 \leq i \leq m$ , are parameters for level  $i$ . We restrict to datasets in which the setting of  $(L_i, N_i, T_i, I_i, P_i)$  is the same for all  $2 \leq i \leq m$ . The default values of maximal nesting level  $m$  and number of transaction object  $N_{m+1}$  are 4 and 100K, respectively.  $k(L_i, N_i, T_i, I_i, P_i)$  denotes  $k$  repetitions of  $(L_i, N_i, T_i, I_i, P_i)$ . In Table 3,

- @ denotes the default setting

$$(L_i = 1K, N_i = 10K, T_i = 20, I_i = 8, P_i = 200), \\ 2 \leq i \leq m.$$

- & denotes the default setting

$$(N_{m+1} = 100K, T_{m+1} = 20, I_{m+1} = 8, P_{m+1} = 200).$$

For example, dataset II=(1K,10K) 3@& in Table 3a refers to the dataset

$$(1K, 10K)(1K, 10K, 20, 8, 200)(1K, 10K, 20, 8, 200) \\ (1K, 10K, 20, 8, 200)(100K, 20, 8, 200).$$

Let us explain our choices of these default values. For the average number  $T_i$  of subobject references in an object, we choose the default value 20 on the basis that a Web page usually contains a small number of links. For example, the top level of Yahoo! has 13 categories. In order to have nontrivial sharing of low-level objects, we choose the number of level- $i$  objects  $N_i$  ( $i \leq m$ ) to be much smaller than the number of transaction objects  $N_{m+1}$ . Indeed, in many applications there are more transaction objects than nontransaction objects. For example, there are more research papers (i.e., transaction objects) than active authors, their organizations, and research topics (i.e., nontransaction objects); there are more movies than active actors, directors, categories, types of awards; there are more students than available courses and professors, etc. We have also tried (Section 4.2.1) larger  $N_i$  (and larger  $L_i$  as well), but our experiments show that doing so only reduces the search space, rather than increases it. This is expected because more objects at lower levels usually means less sharing of subobjects, thus, less sharing of substructures. For example, as the number of available courses is increased, the probability that two students take the same course will be reduced (assuming that the number of courses a student takes does not change).

Our experiment environment is a Sun Ultra-1 workstation with CPU rate of 167 MHz and memory size of 128 MB. In all experiments, the OEM graph  $G$  is stored in a unix file. The nodes are stored in the depth-first order in which nodes are visited in our algorithm. If a node has already been stored, any later reference to the node is made through its location, rather than storing another copy of the node. For nodes that are frequently accessed, usually those at lower levels, we allow to "pin" them in the memory after

TABLE 3  
Size of  $\Pi_k$

NIMISUP (%)	I=(1K,10K)3@(100K,10,4,200)		II=(1K,10K)3@&		III=(1K,10K)3@(100K,30,16,200)	
	$\Pi_1/\Pi_2/\dots/\Pi_k$	pattern	$\Pi_1/\Pi_2/\dots/\Pi_k$	pattern	$\Pi_1/\Pi_2/\dots/\Pi_k$	pattern
10	10	2	38	3	47/1	3
8	19	4	57/4	4	77/3	5
6	32	5	82/11/1	5	101/39/1	4
4	47/5	3	102/22/2	4	135/53/2	4
2	63/12	6	145/32/3	4	196/73/42/4	6

(a)

MINISUP (%)	I=(500,10K)3(500,10K,20,8,200)/&		II=(1K,10K)3@&		III=(2K,10K)3(2K,10K,20,8,200)/&	
	$\Pi_1/\Pi_2/\dots/\Pi_k$	pattern	$\Pi_1/\Pi_2/\dots/\Pi_k$	pattern	$\Pi_1/\Pi_2/\dots/\Pi_k$	pattern
10	48	5	38	3	13	2
8	77/4	4	57/4	4	22	4
6	93/26/3	5	82/11/1	5	40	5
4	132/46/5	7	102/22/2	4	63/2	3
2	199/52/30/3	7	145/32/3	4	88/25/1	4

(b)

MINISUP (%)	I=(1K,5K)3(1K,5K,20,8,200)/&		II=(1K,10K)3@&		III=(1K,20K)3(1K,20K,20,8,200)/&	
	$\Pi_1/\Pi_2/\dots/\Pi_k$	pattern	$\Pi_1/\Pi_2/\dots/\Pi_k$	pattern	$\Pi_1/\Pi_2/\dots/\Pi_k$	pattern
10	61/4	5	38	3	19	3
8	70/21/1	5	57/4	4	30	4
6	111/39/2	7	82/11/1	5	42	4
4	162/50/17/4	8	102/22/2	4	66/3	5
2	215/76/38/5	8	145/32/3	4	93/4	7

(c)

MINISUP (%)	I=(1K,10K)@&		II=(1K,10K)3@&		III=(1K,10K)5@&	
	$\Pi_1/\Pi_2/\dots/\Pi_k$	pattern	$\Pi_1/\Pi_2/\dots/\Pi_k$	pattern	$\Pi_1/\Pi_2/\dots/\Pi_k$	pattern
10	28	2	38	3	30/1	3
8	41	3	57/4	4	62/3	5
6	59/4	5	82/11/1	5	85/30/4	6
4	86/29/3	5	102/22/2	4	119/40/5	7
2	6	114/37/3	145/32/3	4	179/59/16/2	7

(d)

MINISUP (%)	w/o pruning		pruning	
	$\Pi_1/\Pi_2/\dots/\Pi_k$	pattern	$\Pi_1/\Pi_2/\dots/\Pi_k$	pattern
10	38	3	38	3
8	57/39	4	57/4	4
6	82/78/41	5	82/11/1	5
4	102/134/85	4	102/22/2	4
2	145/199/113	4	145/32/3	4

(e)

they are read for the first time. A hash table can map the location in the file to the location in memory for pinned nodes. For the rest of this section, we examine several factors of efficiency.

#### 4.2.1 Size of Search Space

In this group of experiments, we study how the search space is affected by various data characteristics and minimum supports. We use the number of leaf nodes in  $\Pi_k$  to estimate the size of search space.

1. **Effect of sharing of subobjects.** Larger  $T_{m+1}$  and  $I_{m+1}$  lead to more sharing of subobjects. For datasets I, II, and III in Table 3a, we set

$$(T_{m+1} = 10, I_{m+1} = 4),$$

$$(T_{m+1} = 20, I_{m+1} = 8),$$

$$(T_{m+1} = 30, I_{m+1} = 16),$$

respectively. Table 3a shows the number of leaf nodes in  $\Pi_k$ . For example, for dataset II at  $MINISUP = 2$  percent, there are 145, 32, and 3 leaf nodes in levels 1, 2, and 3, respectively, and there are four maximally frequent tree-expressions, indicated in the pattern column. Other entries are interpreted similarly. Comparison of datasets I, II, and III shows that the search space grows as more subobjects are shared. A similar effect is observed for lower levels  $i \leq m$ .

2. **Effect of number of labels.** In Table 3b, we set the number of labels  $L_i$  at 500, 1K, 2K for datasets I, II, and III, respectively, with other parameters unchanged. Table 3b shows the number of leaf nodes. As expected, a smaller  $L_i$  implies a larger search space, due to more sharing of labels.
3. **Effect of number of objects.** In Table 3c, we set the number of object identifiers  $N_i$  at 5K, 10K, 20K in datasets I, II, and III. Table 3c shows that the number of object identifiers has an effect similar to the number of labels in b above: a smaller  $N_i$  implies a larger search space. Importantly, these trends show that simply having more labels and objects (without increasing the sharing of subobjects) only decreases the search space. For this reason we did not experiment with larger  $L_i$  and  $N_i$ .
4. **Effect of number of levels.** In Table 3d, we set the maximal level  $m$  at 2, 4, and 6 in datasets I, II, and III, while fixing other parameters. Table 3d shows that as  $m$  increases, so does the size of search space.

#### 4.2.2 Pruning Strategies

In Table 3e, we compare the number of leaf nodes generated with and without pruning Strategies I, II, and III. We have shown the result for the default dataset (1K,10K)3@&; other datasets have similar trends. The comparison shows that these pruning strategies lead to a quick drop in the size of search space. This confirms our expectation about the effectiveness of pruning strategies.

#### 4.2.3 Execution Time

The figures a1, b1, c1, d1, and e1 in Fig. 13 show the execution time for the five experiments in Table 3. Two general trends can be observed: 1) As the minimum support decreases, the execution time increases with a maximum of 500 seconds in all cases. 2) The execution time is consistent with the size of search space in Table 3.

#### 4.2.4 Scale up

For each experiment on the left side of Fig. 13, we scale up the number of transaction objects  $N_{m+1}$  from 100K to

1,000K, with other parameters unchanged. The right side of Fig. 13 shows the relative time with respect to the time for the corresponding experiment with  $N_{m+1} = 100K$  on the left side. The time is averaged over the different minimum supports used. All cases show a clear linear growth with the number of transaction objects.

We now summarize these experiments as follows:

- The search space is increased when more subobjects are shared and when the minimum support is reduced (Section 4.2.1a). Simply increasing the number of objects and labels does not intensify the computation, unless the sharing of subobjects and labels are increased (4.2.1b and 4.2.1c).
- There is a clear indication that pruning Strategies I, II, and III are effective. All experiments show a quick drop in the number of level- $k$  leaf nodes as  $k$  increases. The small search space justifies the choice of storing  $\Pi_k$  in memory. Note that we have used small minimum supports, ranging from 2 percent to 10 percent, which generally require a larger search space than large minimum supports do.
- No more than 500 seconds are needed for 100K transaction objects in tested data characteristics. Experiments show that the algorithm scales linearly for larger datasets.
- The number of frequent tree-expressions can be large, especially for a small minimum support. The number of maximally frequent tree-expressions is usually very smaller at most eight in all cases studied. Unlike frequent tree-expressions, reducing the minimum support can add a maximally frequent tree-expression that makes several previous maximally frequent tree-expressions no longer maximally frequent. This explains why the number of maximally frequent tree-expressions sometimes is decreased as the minimum support is reduced.

## 5 THE MOVIE DATASET

We apply the algorithm to the Internet Movies Database (IMDb) at <http://us.imdb.com> to discover typical structures of movies documents. As of June 1998, IMDb covers more than 150,000 movies, over 2,250,000 filmography entries, and over 560,000 people. All information is organized into HTML documents. Fig. 14 shows a segment of the movie document for "Star Wars" at [http://us.imdb.com/Title?Star+Wars+\(1977\)](http://us.imdb.com/Title?Star+Wars+(1977)). The reader can take a quick tour of the source at <http://us.imdb.com/tour>. After randomly inspecting some movie titles, we found that some movies have very little information documented, especially those that are very old or from non-English speaking countries. To get movie titles that are sufficiently documented, we ran a query using condition

$$(1950 \leq Released\_Year \leq 1998) \wedge (Country = USA)$$

at <http://us.imdb.com/list>. In return, we got more than 20,000 movie titles. In the next step, we extracted important fields from these movie documents and built the OEM graph. This requires a large number of automated requests from a remote site. We selected only the first 5,000 of

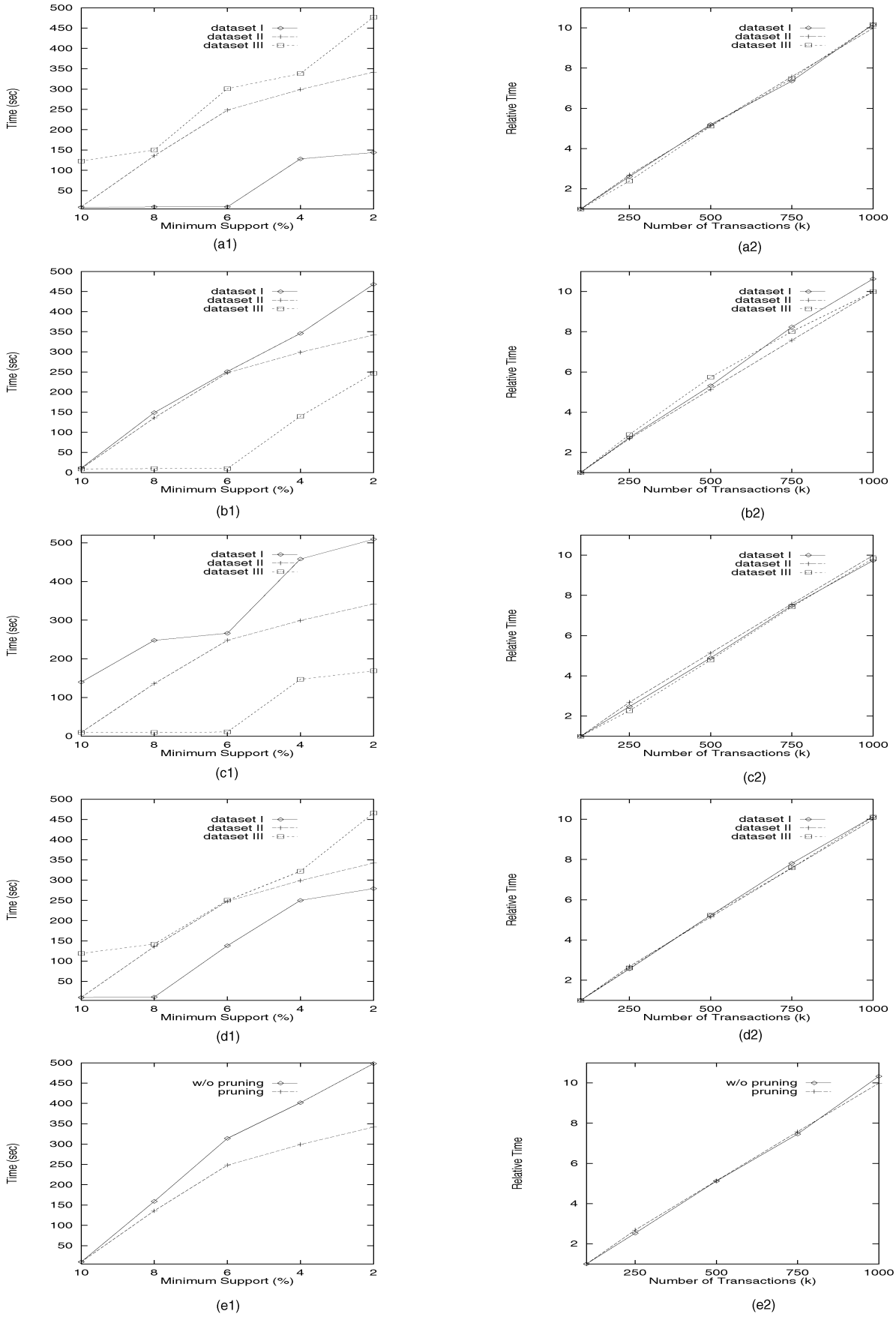


Fig. 13. Left—execution time. Right—scale up.

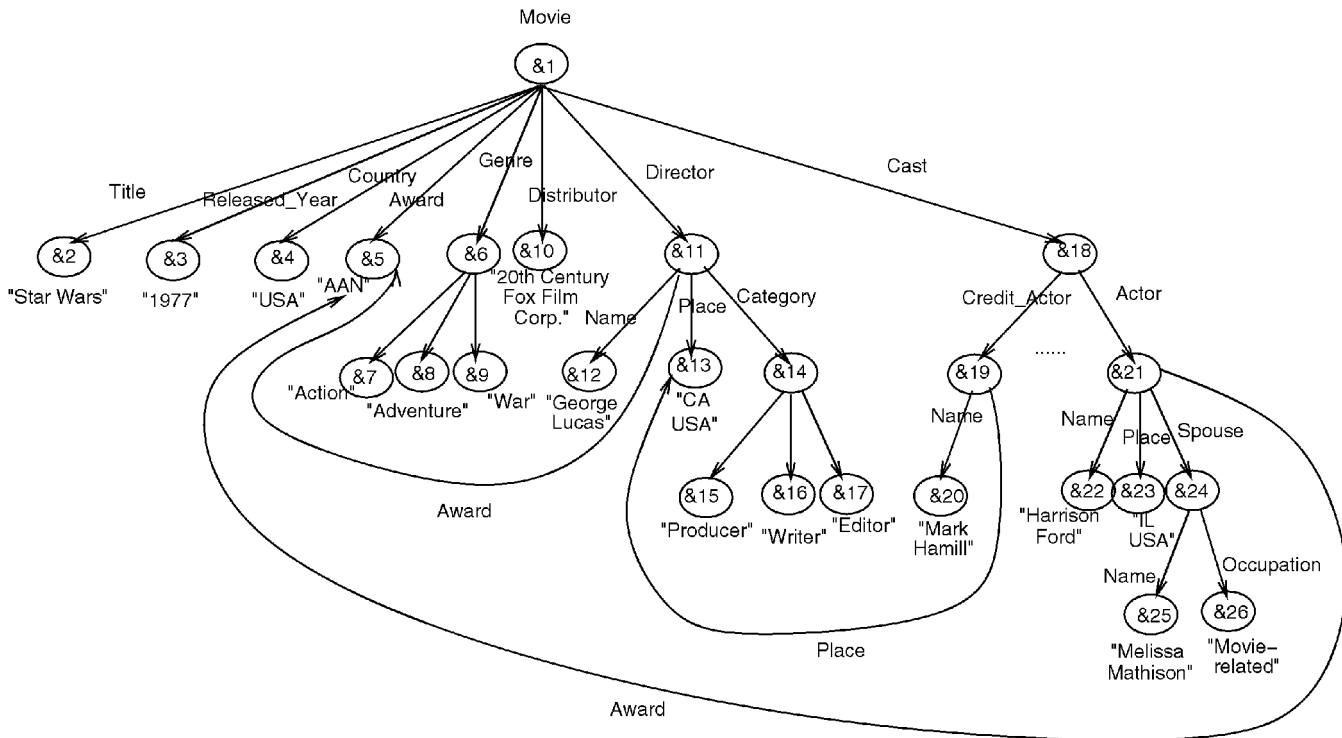


Fig. 14. A segment of the “Star Wars” movie document.

returned movie titles for our experiment. We wrote a profile to tell the extraction program what to extract in a particular context. This is necessary because certain labels can appear in different contexts and at different levels and we do not want all of them. For example, *Title* of movies appears at level 1 as well as within each actor objects, and if we are not interested in the movies in which an actor acts, we can ignore *Title* labels within actor objects. A movie usually has many actors, but we restricted the search to only “active” actors, which we defined as the top 5 actors in each movie (by the way, actors are listed in the order of credits in the source). We ignored certain links such as *Costume\_Design*, *Sound\_Mix*, *Language* and all links to images. The top part of Fig. 15 shows the full structure of a movie document from the perspective of our experiment.

We set *MINISUP* to 50 percent and find the two maximally frequent tree-expressions in Fig. 15 (⊥ is omitted for simplicity). In Pattern 1, none of *Director*, *Producer*, *Writer*, *Editor*, *Composer*, *Cinematographer* individually has enough support for the substructure

$$Bio : \{Born\_Year, Born\_Where\}.$$

In Pattern 2, the wild card label ? matches any of these labels, thus, this substructure is found. There are many nonmaximally frequent tree-expressions and such tree-expressions usually have much higher supports. For example, every movie document has labels *Title*, *Released\_Year*, *Country*, and *Director*, thus

$$\{Title, Released\_Year, Country, Director\}.$$

has 100 percent support. Discovered tree-expressions can be stored and retrieved through a query interface. One can retrieve such information to gain the general information content of the movie source, or to discover the vocabulary

and structure of the source, or to find out statistics of missing or known information (such as *Born\_Year* and *Born\_Where* of actors). Often, it is useful to keep track of the identifiers of movie documents that support each typical structure, i.e., URL addresses in this case. This can be easily incorporated into our algorithm when counting the support of each candidate.

## 6 RELATED WORK

Our work is related to mining association rules from a collection of baskets of items (called transactions) [2], [3]. An example of association rules is “If a customer buys diapers, he/she also buys beer with 80 percent confidence.” The core of the association rule problem is finding all itemsets that are contained in at least some number of baskets. A larger candidate itemset is constructed by joining two smaller frequent itemsets and the support is computed by testing containment of the candidate in baskets. Our work has some important differences. Unlike a flat basket, subobject references in an object can be hierarchical, labeled, ordered, and cyclic; and unlike an itemset, a tree-expression has a tree-like structure, and constructing tree-expressions and counting support require more than joining flat sets and testing set containment. Also, the rich data in our framework requires new pruning strategies. Finally, the use of the wild card label makes our problem very different from the association rule problem.

There are some works on discovering structural information from semistructured data. [8] discovered the type of objects (i.e., sets of labels) based on the relative importance of labels in a larger set and constructs the *type hierarchy*. The type hierarchy is a lattice organization of discovered types ordered by the standard set containment, therefore, very

```

The full structure:
  {Title,
   Country,
   Released_Year,
   Award,
   Production,
   Genre:{Keyword},
   Director:{Category,Bio:{Born_Year,Born_Where,Award,Spouse}},
   Cast:{Actor:{Category,Bio:{Born_Year,Born_Where,Award,Spouse}},
        Actor:{Category,Bio:{Born_Year,Born_Where,Award,Spouse}},
        Actor:{Category,Bio:{Born_Year,Born_Where,Award,Spouse}},
        Actor:{Category,Bio:{Born_Year,Born_Where,Award,Spouse}},
        Actor:{Category,Bio:{Born_Year,Born_Where,Award,Spouse}}},
   Writer:{Category,Bio:{Born_Year,Born_Where,Award,Spouse}},
   Cinematographer:{Category,Bio:{Born_Year,Born_Where,Award,Spouse}},
   Producer:{Category,Bio:{Born_Year,Born_Where,Award,Spouse}},
   Editor:{Category,Bio:{Born_Year,Born_Where,Award,Spouse}},
   Distributor:{Category,Bio:{Born_Year,Born_Where,Award,Spouse}}}

Pattern 1 (support = 61.7%):
  {Title, Released_Year, Country, Genre:{Keyword},
   Director:{Category},
   Cast:{Actor:{Bio:{Born_Year,Born_Where,Spouse},Category},
        Actor:{Bio:{Born_Year,Born_Where},Category},
        Actor:{Category},
        Actor:{Category},
        Actor},
   Producer:{Category}}.

Pattern 2 (support = 50.2%):
  {Title, Released_Year, Country, Genre:{Keyword},
   Cast:{Actor:{Bio:{Born_Year,Born_Where,Spouse},Category},
        Actor:{Bio:{Born_Year,Born_Where},Category},
        Actor:{Category},
        Actor:{Category},
        Actor},
   ?:{Bio:{Born_Year,Born_Where},Category}}.

```

Fig. 15. The full structure and maximally frequent tree-expressions.

different from a tree-expression that generalizes the sub-object relationship in the original data. [9] extracted the schema in a single graph structure. We considered "schemas" that are repeated in a number of graph structures. Consequently, we have to deal with the interestingness of substructures such as confidence and support. [12] derived a uniform object-oriented database schema for multiple objects. They first find the hierarchy for each object and merge them into a global schema. We do not

construct any global schema. Instead, we discover "typical" substructures of objects. Most information extraction systems treat an object as a collection of keywords. We treat an object as a structure of labels, like those found on the Web. Preliminary, versions of our work were reported in [13], [14]. Beyond [13], [14], we have shown that each tree-expression is generated only once (Theorem 3.3), and we have included the full version of the experimental results.

## 7 CONCLUSION

As the amount of data available on-line grows rapidly, most references to important fields are labeled and hierarchical (sometimes also ordered and cyclic). The label of a reference tells the role of the field and the hierarchy of references tells how the information is structured in the source. Traditional data mining methods have treated an object (such as a document) as either a set or a list of items and have not explored internal structures of objects. Our treatment of structures is based on the observation that many objects containing the same type of information are similarly structured, though not identically structured. Typical (sub)structures shared by a large number of objects reveal general information content and representation of the source, and discovering such structures is important for both the end user and the source builder. We have defined the discovery problem and proposed a solution based on a new representation of search space. The efficiency and effectiveness were evaluated on both synthetic datasets and real datasets. Traditional information access tends to emphasize the narrowly specified *querying* and the largely disoriented *browsing* approaches. The approach of mining typical structures of objects provides an alternative to information access.

## ACKNOWLEDGMENTS

The authors would like to thank the anonymous referees for providing many useful comments.

## REFERENCES

- [1] S. Abiteboul, "Querying Semi-Structured Data," *Proc. Int'l Conf. Data Engineering*, 1997. <http://www-db.stanford.edu/pub/papers/icdt97.semistructured.ps>.
- [2] R. Agrawal, T. Imielinski, and A. Swami, "Mining Association Rules Between Sets of Items in Large Databases," *Proc. SIGMOD*, pp. 207–216, 1993.
- [3] R. Agrawal and R. Srikant, "Fast Algorithms for Mining Association Rules," *Very Large Data Bases*, pp. 487–499, 1994.
- [4] P. Buneman, S. Davidson, G. Hillebrand, and D. Suciu, "A Query Language and Optimization Techniques for Unstructured Data," *Proc. SIGMOD*, pp. 505–516, 1996.
- [5] D. Konopnicki and O. Shmueli, "W3QS: A Query System for the World-Wide Web," *Very Large Data Bases*, pp. 54–65, 1995.
- [6] S.B. Huffman and C. Baudin, "Toward Structured Retrieval in Semi-Structured Information Spaces," *Proc. Int'l Joint Conf. Artificial Intelligence*, pp. 751–756, 1997.
- [7] A.O. Mendelzon, G.A. Mihaila, and T. Milo, "Querying the World Wide Web," *Proc. Fourth Int'l Conf. Parallel and Distributed Information Systems*, 1996. <ftp://ftp.db.toronto.edu/pub/papers/pdis96.ps.gz>.
- [8] S. Nestorov, S. Abiteboul, and R. Motwani, "Inferring Structure in Semistructured Data," *Proc. Workshop Management of Semistructured Data*, pp. 42–48, May 1997. See [15].
- [9] S. Nestorov, J. Ullman, J. Wiener, and S. Chawathe, "Representative Objects: Concise Representations of Semistructured, Hierarchical Data," *Proc. Int'l Conf. Data Engineering*, 1997.
- [10] Y. Papakonstantinou, H. Garcia-Molina, and J. Widom, "Object Exchange Across Heterogeneous Information Sources," *Proc. Int'l Conf. Data Engineering*, pp. 251–260, 1995.
- [11] S.W. Reyner, "An Analysis of a Good Algorithm for the Subtree Problem," *SIAM J. Computing*, vol. 6, no. 4, Dec. 1977.
- [12] D.Y. Seo, D.H. Lee, K.M. Lee, and J.Y. Lee, "Discovery of Schema Information from a Forest of Selectively Labeled Ordered Trees," *Proc. Workshop Management of Semistructured Data*, pp. 54–59, May 1997. See [15].

- [13] K. Wang and H.Q. Liu, "Schema Discovery from Semistructured Data," *Proc. Int'l Conf. Knowledge Discovery and Data Mining*, pp. 271–274, Aug. 1997.
- [14] K. Wang and H.Q. Liu, "Discovering Typical Structures of Documents: A Road Map Approach," *Proc. ACM SIGIR Conf. Research and Development in Information Retrieval*, pp. 146–154, Aug. 1998.
- [15] The Workshop on Management of Semistructured Data, 1997. <http://www.research.att.com/suciu/workshop-papers.html>.



**Ke Wang** received the PhD degree from Georgia Institute of Technology. He is an associate professor at School of Computing, National University of Singapore. His current research work include data mining, text mining, and web mining, and applications of data mining. Details can be found at <http://www.comp.nus.edu.sg/~wangk>.



**Huiqing Liu** received the MSc degree from the School of Computing of National University of Singapore (NUS) in 1997. She is a research fellow at Bioinformatics Centre at (NUS). Her current research work is focused on mining from biological data. Please refer to <http://www.bic.nus.edu.sg>.