Discovering Typical Structures of Documents: A Road Map Approach

Ke Wang
Department of Information Systems and Computer Science
National University of Singapore
www.iscs.nus.edu.sg/~wangk

Huiqing Liu
BioInformatics Center
National University of Singapore
www.bic.nus.edu.sg

Abstract The *structure* of a document refers to the role and hierarchy of subdocument references. Many on-line documents are similarly structured, though not identically structured. We study the problem of discovering "typical" structures of a collection of such documents, where the user specifies the minimum frequency of a typical structure. We will consider structural features of subdocument references such as labeling, nesting, ordering, cyclicity, and wild-card references, like those found on the Web and digital libraries. Typical structures can be used to serve the following purposes. (a) The "table-of-content" for gaining the general information of a source. (b) A road map for browsing and querying a source. (c) A basis for clustering documents. (d) Partial schemas for building structured layers to provide standard database access methods. (e) User/customer's interests and browsing patterns. We present a solution to the discovery problem.

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 are combined from several heterogeneous sources. See [11] for some recent work on semistructured data. Unlike unstructured raw data (such as image and sound), semistructured documents do have some structures. Figure 1 shows a segment of the semistructured document about the soccer league, modified from [7]. Each circle plus the text inside represents a subdocument (e.g., a HTML file) and its identifier (e.g., URL). The arrows and their labels, identifiable by special tags or a grammar, represent subdocument references and roles. In this paper, the term *structure* refers to the role (in the form of labels) and hierarchical relationships of subdocument references. The structure of a document gives a sense of what sort of questions might be answered by a more intensive examination of the document and how the information is represented.

For example, the structure of a person document could tell that the person has labeled fields *Name, Address, Hobby*, and *Friend* and that the *Address* subdocument has labeled fields *Street* and *Zipcode*. Knowing this information can help the user to decide whether a source contains the sort of information he/she wants, such as the personal information in this case.

Unlike structured data (such as relational or object-oriented databases), semistructured documents have no absolute schema or class fixed in advance, and each document contains its own schema or structure. For example, some clubs have more players than others; some fields are missing for some clubs; some players have birth day recorded and some do not; some have nicknames and some do not; etc. This structural irregularity, however, does not imply that there is no structural similarity among documents. On the contrary, it is quite common for documents describing the same type of information to have similar structures. For example, every club document has *Name* label and at least 10 *Player* labels; every player document has *Name* label; 50% player documents have *Nationality* label, etc. Other sources of documents having similar structures are documents about universities, movies, countries, census data, branch information within an organization, etc.

1.2 Main results

We consider the following discovery problem: given a collection of documents, find all "typical" structures that occur in a minimum number of documents specified by the user. We formally define this problem in Section 2. It is worth mentioning that though we use the term structure, it is up to the user to specify what is the structure of a document. For example, if the user wants to find frequent co-occurrences of keywords in several free-running text documents (thus, no structure at all in an usual sense), he/she just needs to specify all keywords as labels. In this case, a typical structure is a set of keywords that co-occur in some minimum number of free-running 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.

1.3 Application

Depending on applications, discovering typical structures 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 typical structure is associated with identifiers (e.g., URL) of documents that contain the structure, so that relevant documents can be retrieved and examined easily. The

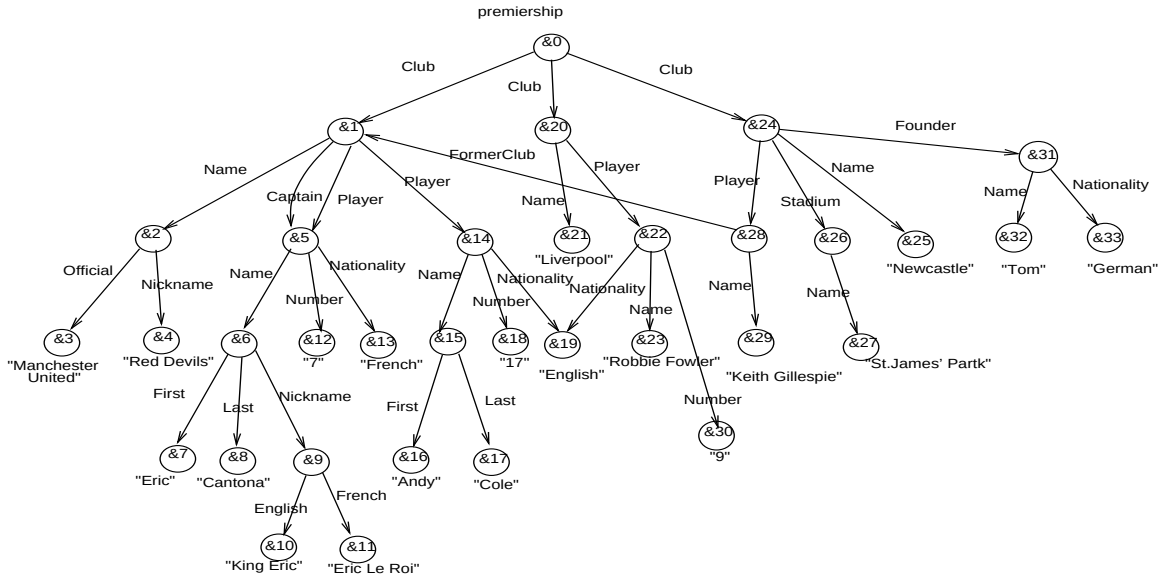


Figure 1: The premiership document

following list gives a taste of applications of the discovery problem.

- **Road maps for querying/browsing a data source**. One limitation of querying and browsing semistructured documents, such as those found on the Web or digital libraries, is the disorientation resulting in the infamous "lost-in-hyperspace" syndrome, due to the lack of external schema information. To formulate any meaningful query, say in WebSQL [5], 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 documents.

- **General information content**. Very often the user may not be looking for anything specific at all but rather may wish to discover the general information content or an overview of a source. A more appropriate search mode for such users would be examining the structure of the source, just like examining the table-of-content if a reader likes to gain a gist of a book. This can be done by requesting to lay out the structure of each document if there are only a few documents, or to lay out 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. The point is that searching or browsing stored typical structures is much faster than searching or browsing actual documents.

- **A guide for building indexes and views**. To speed up information retrieval, it is desirable to construct indexes and views on frequently retrieved, typically occurred structures. Discovering typical structures helps this task. For example, suppose that 98% person documents have Name label and most of time documents are retrieved using this label, indexing Name (e.g., by a B-tree, hash table, or inverted list) will speed up document retrieval.

- **Structure-based document clustering**. A tree-like subdocument reference within a document is usually ignored by traditional clustering methods. In the semistructured view of documents, each subdocument reference is labeled by its topic, and the topic of a document can be represented by a tree-like structure of such topics. In the tree-like structure, the topic of a subdocument is relative to that of its superdocument. By clustering documents based on such topical structures, the search for IBM catalog prices for personal computers will not return things such as an advertisement for an "I Bought Mac" T-shirt and the SIGIR home page.

- **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 documents consisting of hyperliked Web pages accessed in different sessions. Each document is rooted at the entry page of one session. By labeling pages with either topics or site information, a typical structure captures user's interests and access patterns.

Before performing the discovery task, the structure of each document should be extracted. The extraction removes the unstructured data such as image and video that do not contribute to the structure of the source. Some systems provide "wrappers" or one can write a parser or thesaurus for the extraction [11]. In this paper, we assume such extraction has been done.

The paper is organized as follows. Section 2 defines the discovery problem. Section 3 presents the algorithm. Section 4 presents a case study using a real dataset. Section 5 reviews related work. Section 6 concludes the paper.

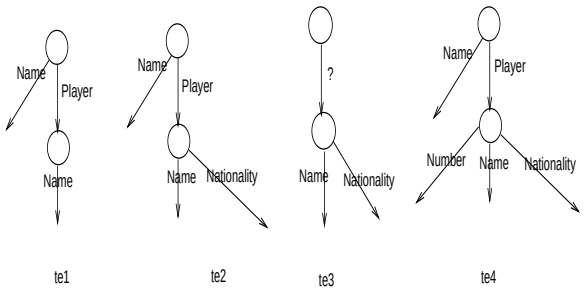


Figure 2: Tree-expressions of club documents

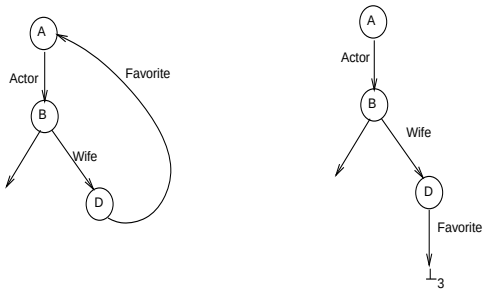


Figure 3: Tree-expressions extended to represent cycles

2 The Problem

We adopt the *Object Exchange Model (OEM)* for representing semistructured documents. For a detailed account of OEM, the interested reader may refer to [1, 4, 8].

Every document o in OEM consists of an *identifier*, denoted $\&o$, and a *value*, denoted $val(\&o)$. The identifier $\&o$ uniquely identifies document o. The value $val(\&o)$ is either an *atomic document*, such as an integer or a string; or a *list document* of the form $< l_1 : \&o_1, \ldots, l_p : \&o_p >$; or a *bag document*, of the form $\{l_1 : \&o_1, \ldots, l_p : \&o_p\}$. $\&o_i$ are identifiers of subdocuments o_i. l_i are *labels* that describe the role of subdocuments o_i. As usual, the order in a bag does not matter, but it does in a list. Repeating of subdocuments is allowed in a bag or a list.

The OEM can be conveniently represented by a labeled and ordered multi-graph: nodes are document identifiers; for each reference $l_i : \&o_i$ in $val(\&o)$, there is an edge $(\&o, \&o_i)$ labeled l_i. If $val(\&o)$ is a list, all outgoing edges at $\&o$ are ordered. If $val(\&o)$ is a bag, all outgoing edges at $\&o$ are unordered. *Root documents*, specified by the user, are those whose structures will be discovered. An OEM is *cyclic* if its graph is cyclic. Indeed, OEM graphs of many Web documents are cyclic.

We need a mechanism for generalizing the structures of several documents. This is defined by the notion of tree-expressions below. For the rest of the paper, symbol ? denotes the *wild-card* that matches with any level, and $\perp$ denotes the *nil schema* containing no structure.

Tree-expressions. First, we consider acyclic OEM graphs. For any label l, l^* denotes either l or the wild-card ?. Assume that te_i are *tree-expressions* of documents o_i, $1 \leq i \leq p$.

- $\perp$ is a *tree-expression* of any document.
- If $val(\&o) = \{l_1 : \&o_1, \ldots, l_p : \&o_p\}$ and $\{i_1, \ldots, i_k\}$ is a subset of $\{1, \ldots, p\}$, $k > 0$, $\{l^*_{i_1} : te_{i_1}, \ldots, l^*_{i_k} : te_{i_k}\}$ is a *tree-expression* of document o.
- If $val(\&o) =< l_1 : \&o_1, \ldots, l_p : \&o_p >$ and $< i_1, \ldots, i_k >$ is a subsequence of $< 1, \ldots, p >$, $k > 0$, $< l^*_{i_1} : te_{i_1}, \ldots, l^*_{i_k} : te_{i_k} >$ is a *tree-expression* of document o.

To ensure that the "terminal" label on a path is not a wild-card, if te_{i_j} is $\perp$, $l^*_{i_j}$ must be l_{i_j}. A tree-expression $\{l_{i_1} : te_{i_1}, \ldots, l_{i_k} : te_{i_k}\}$ or $< l_{i_1} : te_{i_1}, \ldots, l_{i_k} : te_{i_k} >$ has a tree representation in which te_{i_j}'s form the subtrees, each being labeled l_{i_j}.

Example 2.1 Consider Figure 1. $te_1 = \{Player : \{Name : \perp\}, Name : \perp\}$ is a tree-expression of documents &1, &20, &24, so is the result of replacing *Player* with ?. However, replacing any $Name$ with ? does not result in a tree-expression because "terminal" labels cannot be the wild-card. $te_2 = \{Player : \{Name : \perp, Nationaltiy : \perp\}, Name : \perp\}$ is a tree-expression of documents &1 and &20, but not of &24. $te_3 = \{? : \{Name : \perp, Nationality : \perp\}\}$ is shared by clubs &1, &20, &24, that is, some member in these clubs (i.e., either *Player* or *Founder*) has both $Name$ and $Nationality$. Figure 2 shows the trees for te_1, te_2, and te_3. (End)

Cyclic references. For cyclic OEM graphs, tree-expressions defined above may be infinitely large. To solve this problem, we allow a leaf node to be named by special symbols $\perp_i$, $i > 0$. Essentially, $\perp_i$ is the alias of the ancestor that is i nodes above the leaf node in the tree-expression, called the *ith ancestor*. Figure 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 third ancestor A. By treating each $\perp_i$ as a leaf node, we are able to continue dealing with a tree-expression as a tree.

Very often, we are interested in most "informative" substructures. In Figure 2, te_4 is more informative than te_2 which is more informative than te_1. This is compared by the "weaker than" relation between two tree-expressions.

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

- $\{l_1 : te_1, \ldots, l_p : te_p\}$ is *weaker than* $\{l'_1 : te'_1, \ldots, l'_q : te'_q\}$ if every te_i is weaker than some te'_{j_i}, where either $l_i = l'_{j_i}$ or $l_i =$?.
- $< l_1 : te_1, \ldots, l_p : te_p >$ is *weaker than* $< l'_1 : te'_1, \ldots, l'_q : te'_q >$ if every te_i is weaker than some te'_{j_i}, where either $l_i = l'_{j_i}$ or $l_i =$?, and $< j_1, \ldots, j_p >$ is a subsequence of $< 1, \ldots, q >$.

Intuitively, if te_1 is weaker than te_2, all structural information of te_1 is found in te_2.

Definition 2.1 (The discovery problem) Consider a tree-expression te. The *support* of te is the number of root documents d such that te is weaker than d. For a user-specified minimum support $MINISUP$, 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 maximally frequent tree expressions. (End)

If we are interested in all frequent tree-expressions, we can turn off the maximality requirement in the above problem. For the rest of the paper, we consider only maximally frequent tree-expressions.

Example 2.2 In Figure 1, suppose that &1, &20, &24 are the root documents. Refer to Example 2.1 and Figure 2 for te_1, te_2, te_3, te_4. The support of te_1 and te_3 is 3, and the support of te_2 and te_4 is 2. If $MINISUP = 3$, te_1 and te_3 are frequent, but te_2 and te_4 are not. Both te_1 and te_3 are also maximally frequent. Suppose $MINISUP = 2$. Then te_1, te_2, te_3, te_4 all are frequent. But since te_1, te_2, te_3 are weaker than te_4, only te_4 is maximally frequent. (End)

3 The Algorithm

The problem of finding frequent subsets from a collection of sets [2] is related to our problem. Unlike flat sets, however, documents have structures, in the form of labeled hierarchical subdocument references. In addition, we allow the wild-card label that matches with any label. Therefore, the algorithm in [2] is not directly applicable to the problem at hand. This justifies to present a new mining algorithm.

We do not assume that the OEM graph G fits in the memory entirely. 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 (recall that G is a multi-graph), we assume that G contains at most one edge between two nodes and that one set of labels is associated with each edge. $L(w, z)$ denotes the set of labels associated with edge (w, z). The intended use of $L(w, z)$ is as follows: as we traverse a path $w_1, \ldots, w_k$, where w_i's are nodes, we have effectively traversed all paths labeled by the sequences in the cross product $L(w_1, w_2) \times \ldots \times L(w_{k-1}, w_k)$. The information stored for each node w includes addresses of all subnodes, $L(w, z)$ for every subnode z, and positions in $val(w)$ at which each label in $L(w, z)$ appears.

One important property of our algorithm is to traverse simple paths (i.e., paths on which only the first and last nodes can be the same) of G in the depth-first order. 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 on the same disk page and may be entered from different supernodes. To reduce the disk access, frequently referenced nodes, i.e., those with a large in-degree, can be stored in memory and infrequently referenced nodes stored on the disk. The exact implementation on disk is transparent to the presentation of our algorithm.

3.1 Representing tree-expressions

A *k-tree-expression* is a tree-expression containing k leaf nodes (i.e., nodes for $\perp$ or $\perp_i$). A leaf node can be represented by a path of the form $[\top, l_1, \ldots, l_n, \perp]$, where symbol $\top$ represents a generic root document, and l_i are the labels on a simple path in G starting from a root document. $\perp$ is replaced with $\perp_i$ if the last node on the path is identical to the ith ancestor on the path. A k-tree-expression can be constructed by "gluing" a sequence of k paths that are not prefix of each other. First, $\top$ of all paths are glued together to form the root. Recursively, all paths at a node that match the next label are glued together, such that the order of leaf nodes is the order of paths in the sequence. In other words, the k-tree-expression constructed is the "prefix tree" of the k paths that preserves their order in the sequence.

However, the above representation has two problems.

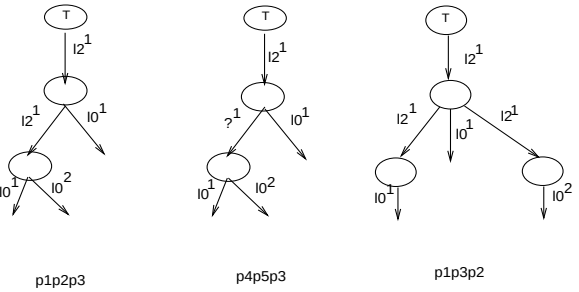


Figure 4: Constructing tree-expressions

First, no repeating labels for subdocuments are constructed because they are always glued together. For example, $\{l : \perp, l : \perp\}$ is a legal tree-expression of document $\{l : \&a, l : \&b\}$, but cannot be constructed using the only path $[\top, l, \perp]$. We solve this problem by distinguishing all repeating labels using different superscripts. In the above example, instead of generating only one path $[\top, l, \perp]$, we generate two paths $[\top, l^1, \perp]$ and $[\top, l^2, \perp]$, where superscripts 1 and 2 correspond to the two occurrences of label l for two sibling documents.

Second, the wild-card label ? is not considered. This problem can be solved by adding the wild-card label ? to the set of labels $L(w, z)$ associated with each edge (w, z). The maximal superscript for ? is the maximal number of occurrences of a label for sibling documents.

Path-expressions. With the above modifications, a k-tree-expression can be represented by a sequence $p_1 \ldots p_k$ of k paths of the form $[\top, l_1^{j_1}, \ldots, l_n^{j_n}, \perp]$ or $[\top, l_1^{j_1}, \ldots, l_n^{j_n}, \perp_i]$, such that l_i are labels on a simple path in G starting at a root document, j_i are superscripts for labels l_i, and no p_i is a prefix of another. The last label l_n must be a non-wild-card, as required for tree-expressions. Paths $[\top, l_1^{j_1}, \ldots, l_n^{j_n}, \perp]$ or $[\top, l_1^{j_1}, \ldots, l_n^{j_n}, \perp_i]$ are called *path-expressions* below.

Example 3.1 Consider the root document t defined as

$$val(\&t) = \{l_0 : \&o_2, l_2 : \&c\},$$
$$val(\&c) = \{l_0 : \&o_1, l_2 : \&a, l_0 : \&o_2, l_2 : \&a, l_0 : \&o_1\},$$
$$val(\&a) = \{l_0 : \&o_1, l_0 : \&o_1\}.$$

The maximal superscript for l_0 is 3, and the maximal superscript for l_2 is 2. Thus, the maximal superscript for ? is 3. Consider two tree-expressions of t:

$$te_1 = \{l_2 : \{l_2 : \{l_0 : \perp, l_0 : \perp\}, l_0 : \perp\}\} \text{ and}$$
$$te_2 = \{l_2 : \{? : \{l_0 : \perp, l_0 : \perp\}, l_0 : \perp\}\}.$$

As shown in Figure 4, te_1 can be constructed by sequence $p_1p_2p_3$, and te_2 by sequence $p_4p_5p_3$, where p_i are path-expressions:

$p_1 = [\top, l_2^1, l_2^1, l_0^1, \perp]$,
$p_2 = [\top, l_2^1, l_2^1, l_0^2, \perp]$,
$p_3 = [\top, l_2^1, l_0^1, \perp]$,
$p_4 = [\top, l_2^1, ?^1, l_0^1, \perp]$,
$p_5 = [\top, l_2^1, ?^1, l_0^2, \perp]$. (End)

For the rest of the paper, a k-tree-expression is represented by a sequence $p_1 \ldots p_k$ of k path-expressions, as defined above.

3.2 The algorithm

Searching the whole space of k-tree-expressions is prohibitive. Fortunately, we do not need to consider a k-tree-expression if some "substructure" of it is known to be infrequent. This is stated below.

Theorem 3.1 Let p_i denote path-expressions. Every frequent k-tree-expression $p_1 \ldots p_k$ is constructed by two frequent $(k-1)$-tree-expressions $p_1 \ldots p_{k-2}p_{k-1}$ and $p_1 \ldots p_{k-2}p_k$.

Following Theorem 3.1, we compute frequent k-tree-expressions in the order of k, starting with frequent 1-tree-expressions, i.e., frequent path-expressions. We examine a k-tree-expression $p_1 \ldots p_k$ only if the two $(k-1)$-tree-expressions $p_1 \ldots p_{k-1}$ and $p_1 \ldots p_{k-2}p_k$ are frequent. On the appearance, Theorem 3.1 looks similar to the subset property in [3]: an itemset $\{i_1, \ldots, i_k\}$ is frequent only if both $\{i_1, \ldots, i_{k-2}, i_{k-1}\}$ and $\{i_1, \ldots, i_{k-2}, i_k\}$ are frequent. This may suggest that [3] can be applied here by treating each tree-expression $p_1 \ldots p_k$ as an itemset $\{p_1, \ldots, p_k\}$. However, this does not work. In Figure 4, $p_4p_5p_3$ is weaker than $p_1p_2p_3$, but $\{p_4, p_5, p_3\}$ is not a subset of $\{p_1, p_2, p_3\}$. Consequently, it is possible that tree-expression $p_4p_5p_3$ is frequent, but itemset $\{p_4, p_5, p_3\}$ is not. Another difference is that, in general, $p_1 \ldots p_k$ and its permutations represent different tree-expressions. For example, $p_1p_2p_3$ and $p_1p_3p_2$ in Figure 4 are different. In addition, we need to deal with wild-card labels that require special treatments.

Since $p_1 \ldots p_k$ being maximally frequent implies that $p_1 \ldots p_{i-1}p_{i+1}$ are *not* maximally frequent (because the latter are weaker than the former), we will perform the pruning after constructing all frequent tree-expressions. For the rest of the paper, F_k denotes the set of frequent k-tree-expressions.

Phase I: Computing F_1

This phase finds all frequent 1-tree-expressions in the form of path-expressions $[\top, l_1^{j_1}, \ldots, l_n^{j_n}, \bot]$ or $[\top, l_1^{j_1}, \ldots, l_n^{j_n}, \bot_i]$. Since the represented 1-tree-expressions depend only on labels l_i's, not on superscripts j_i's, the support of a 1-tree-expression is denoted $sup(l_1, \ldots, l_n, \bot)$ or $sup(l_1, \ldots, l_n, \bot_i)$, defined as the number of root documents from which there is a simple path in G labeled $l_1, \ldots, l_n$.

Here is the computation of *sup*. For each root document t, we depth-first traverse all simple paths in G starting from t. On visiting a node w_n, let $t, w_1, \ldots, w_n$ be the sequence of nodes on the path being traversed.

Acyclic path: w_n is not identical to any w_i, $i < n$. We increment all $sup(l_1, \ldots, l_n, \bot)$ if they were not increased for t, where $(l_1, \ldots, l_n)$ is in the cross product $L(t, w_1) \times L(w_1, w_2) \times \ldots \times (L(w_{n-1}, w_n) - \{?\})$.

Cyclic path: w_n is identical to some w_i, $i < n$. We increment all $sup(l_1, \ldots, l_n, \bot_i)$ if they were not increased for t, where $(l_1, \ldots, l_n)$ is in the cross product $L(t, w_1) \times L(w_1, w_2) \times \ldots \times (L(w_{n-1}, w_n) - \{?\})$.

To find all frequent 1-tree-expressions F_1, for each $sup(l_1, \ldots, l_n, \bot)$ or $sup(l_1, \ldots, l_n, \bot_i)$ not less than MINISUP, all path-expressions of the form $[\top, l_1^{j_1}, \ldots, l_n^{j_n}, \bot]$ or $[\top, l_1^{j_1}, \ldots, l_n^{j_n}, \bot_i]$ are added to F_1. The next example shows the work of computing F_1.

Example 3.2 Consider the OEM graph in Figure 5 for two root documents $\&t_1$ and $\&t_2$ defined as:

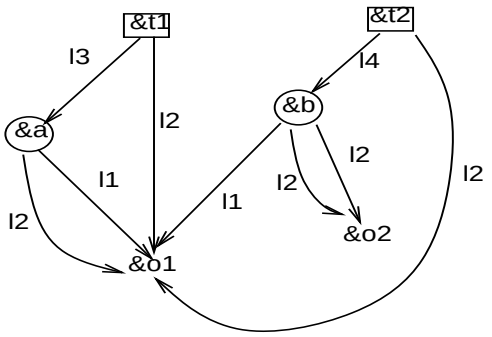


Figure 5: The OEM graph in Example 3.2

$$val(\&t_1) =< l_3 : \&a,\ l_2 : \&o_1 > \text{ and}$$
$$val(\&t_2) =< l_4 : \&b,\ l_2 : \&o_1 >.$$

$\&a$, $\&b$ are defined as

$$val(\&a) = \{l_1 : \&o_1,\ l_2 : \&o_1\} \text{ and}$$
$$val(\&b) = \{l_1 : \&o_1,\ l_2 : \&o_2, l_2 : \&o_2\}.$$

$\&o_1$ and $\&o_2$ are atomic documents. A squared node represents a list document and a circled node represents a bag document. l_2 has repeated twice in $val(\&b)$, its maximal superscript is 2, and so is the maximal superscript of ?. Suppose that the $MINISUP = 2$. 8 frequent path-expressions $p_1, \ldots, p_8$ are found during Phase I, shown in Table 1. For example, $sup(l_2, \bot) = 2$ because each root document has a simple path labeled l_2. From this support, two frequent path-expressions, i.e., p_1 and p_2 are generated. The other frequent path-expressions are similarly generated. (End)

Phase II: Computing F_k

The search space. Following Theorem 3.1, each frequent k-tree-expression $p_1 \ldots p_{k-2}p_{k-1}p_k$ is constructed by two frequent $(k-1)$-tree-expressions $p_1 \ldots p_{k-2}p_{k-1}$ and $p_1 \ldots p_{k-2}p_k$, called a *matching pair* below. We will represent $F_1, \ldots, F_{k-1}$ by a $(k-1)$-*candidate-trie*, denoted Π_{k-1}, that facilitates retrieval of all matching pairs.

Π_{k-1} is a trie of depth $k-1$. Each non-root node in Π_{k-1} represents a frequent path-expression p_i in F_1. Each root-to-leaf path of length $j \leq k-1$, called a *j-candidate-path*, represents a frequent j-tree-expression in F_j, given by the j path-expressions on the root-to-leaf path. Therefore, a matching pair $p_1 \ldots p_{k-2}p_{k-1}$ and $p_1 \ldots p_{k-2}p_k$ is represented by a pair of $(k-1)$-candidate-paths ending at two sibling nodes. Therefore, each leaf node v in Π_{k-1} represents two things: the path-expression at v and the candidate-path ending at v. To begin with, Π_1 has one child for each path-expression in F_1.

Generating candidates. To generate Π_k from Π_{k-1}, for every matching pair $p_1 \ldots p_{k-2}p_{k-1}$ and $p_1 \ldots p_{k-2}p_k$ represented by sibling leaf nodes l and l', we create a new child under l for representing the k-candidate-path $p_1 \ldots p_{k-2}p_{k-1}p_k$. This operation is called *extending l by l'*. Figure 6 shows Π_1, Π_2, Π_3 generated by three frequent path-expressions p_1, p_2, p_3, assuming that all 2-tree-expressions and 3-tree-expressions are frequent.

Counting the support. The support of k-candidate-paths is counted by scanning all root documents. For

path-expressions	supporting root documents	tree-expressions represented
$p_1 : [\top, l_2^1, \bot]$	$\&t_1, \&t_2$	$< l_2 : \bot >$
$p_2 : [\top, l_2^2, \bot]$ (pruned)	$\&t_1, \&t_2$	$< l_2 : \bot >$
$p_3 : [\top, ?^1, l_1^1, \bot]$	$\&t_1, \&t_2$	$<? : \{l_1 : \bot\} >$
$p_4 : [\top, ?^1, l_2^1, \bot]$	$\&t_1, \&t_2$	$<? : \{l_2 : \bot\} >$
$p_5 : [\top, ?^1, l_2^2, \bot]$ (pruned)	$\&t_1, \&t_2$	$<? : \{l_2 : \bot\} >$
$p_6 : [\top, ?^2, l_1^1, \bot]$ (pruned)	$\&t_1, \&t_2$	$<? : \{l_1 : \bot\} >$
$p_7 : [\top, ?^2, l_2^1, \bot]$ (pruned)	$\&t_1, \&t_2$	$<? : \{l_2 : \bot\} >$
$p_8 : [\top, ?^2, l_2^2, \bot]$ (pruned)	$\&t_1, \&t_2$	$<? : \{l_2 : \bot\} >$

Table 1: F_1 in Example 3.2

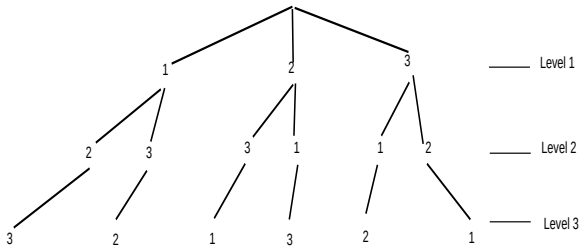


Figure 6: Π_1, Π_2, Π_3

each root document t, we read the hierarchy of t, traverse Π_k and increase the support of a k-candidate path if it is a tree-expression of t. Π_k serves a structure to prune the traversal of subtrees: before reaching a candidate at level k, if the current path $p_1 \ldots p_j$ traversed is not a tree-expression of t, the traversing into the subtree below p_j is pruned. Pruning non-frequent candidates corresponds to deleting leaf nodes at level k having support less than $MINISUP$.

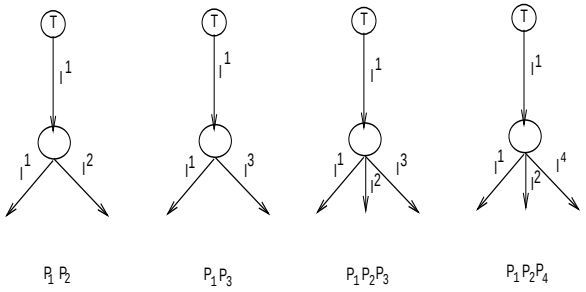


Figure 7: Constructing natural by unnatural

Pruning of search space. Pruning the search space Π_k is crucial to the performance. One observation is that several candidate-paths in Π_k may represent the same tree-expression by ignoring superscripts i of labels l^i and it suffices to consider only one of such candidate-paths. A tree-expression is *natural* if for every non-leaf node with k branches of the same label, the superscripts on these branches are $1, \ldots, k$ from left to right. Otherwise, the tree-expression is *unnatural*. A candidate-path is *natural (unnatural)* if it represents a natural (unnatural) tree-expression. For example, in Figure 7, $p_1p_2p_3$ and p_1p_2 are natural, and p_1p_3 is unnatural. Note that there are far more unnatural candidate-paths than natural ones, just like far more unsorted sequences than the sorted sequence. Essentially, we only need to consider natural candidate-paths.

However, some unnatural candidate-paths are needed to generate natural ones in the construction of Theorem 3.1. In Figure 7, for example, natural $p_1p_2p_3$ is generated by extending natural p_1p_2 by unnatural p_1p_3. Therefore, simply pruning all unnatural candidate-paths does not work. On the other hand, extending an unnatural candidate-path always generates an unnatural candidate-path because of the unnatural prefix, and for the same reason the result of such extension is not useful to generate a natural candidate-path. This gives the first pruning strategy.

Strategy I. A leaf node is extended only if it represents a natural candidate-path. After all extensions in Π_{k-1} are considered, leaf nodes representing unnatural $(k-1)$-candidate-paths can be pruned.

By Strategy I, an unnatural candidate-path is never extended. This is highly effective because there are much more unnatural candidate-paths than natural ones. Some unnatural candidate-paths don't even have to be generated. A tree-expression is *super-unnatural* if at some node the superscripts on branches for the same label are not sorted. A candidate-path is *super-unnatural* if it represents a super-unnatural tree-expression. In Figure 7, the tree-expression for p_1p_3 is unnatural but is not super-unnatural, and p_3p_1 is super-unnatural. An extension involving a super-unnatural candidate-path always generates a super-unnatural candidate-path. Thus, we have

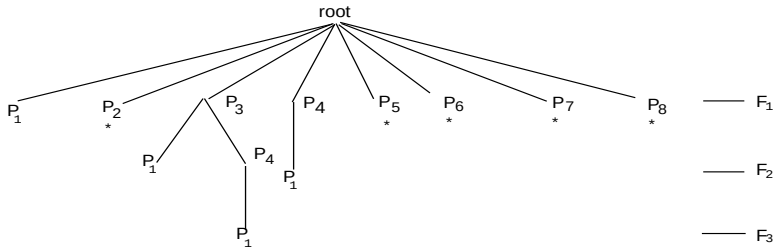
Strategy II. An extension is performed only if it does not generate a super-unnatural candidate-path.

Finally, if extending leaf node l generates a frequent candidate-path, the candidate-path at l becomes non-maximally frequent because it is weaker than the new tree-expression generated. Since the candidate-path at l will not be used in a later pass, it can be pruned.

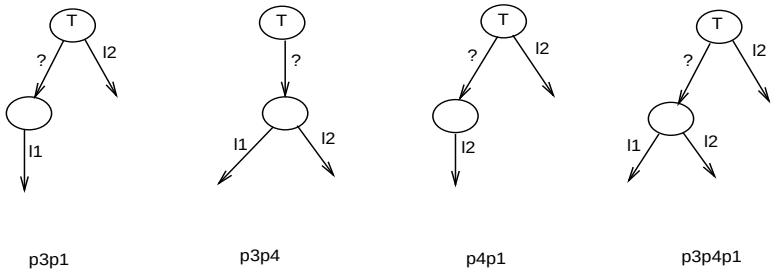
Strategy III. If extending l generates a frequent candidate-path, prune the tree-expression represented by l.

A leaf node l becomes non-leaf after extended. Thus, we can enforce Strategy III by revising F_k to be only the leaf nodes at level k in Π_k. Another strategy is to prune candidate-paths that are reorderings of subnodes of a bag node. We omit the detail.

Example 3.3 Continue with Example 3.2. Figure 8(a) shows Π_1, Π_2, Π_3, corresponding to levels 1, 2, and 3, respectively. 3 frequent 2-candidate-paths and 1 frequent 3-candidate-paths are generated. Candidate-paths not included in Π_i are either not frequent or pruned by our pruning strategies. (Please refer to Table 1 for the actual path-expressions represented by p_i.) For example, all 2-candidate paths p_1p_i are not frequent, where $2 \leq i \leq 8$; p_4p_3 is the same as p_3p_4 (i.e., reordering of subnodes of a bag node); 1-candidate paths p_2, p_5, p_6, p_7, p_8, marked by * in Figure 8(a), are not extended because they are unnatural (thus, pruned from F_1). Figure 8(b,c) shows



(a) Π_1, Π_2, Π_3 in Example 3.3



(b) 2-tree-expressions and 3-tree-expressions

sequences of path-expressions	supporting root documents	tree-expressions represented
F_2		
p_3p_1	$\&t_1, \&t_2$	$<?: \{l_1 : \perp\},\ l_2 : \perp>$
p_3p_4	$\&t_1, \&t_2$	$<?: \{l_1 : \perp,\ l_2 : \perp\}>$
p_4p_1	$\&t_1, \&t_2$	$<?: \{l_2 : \perp\},\ l_2 : \perp>$
F_3		
$p_3p_4p_1$	$\&t_1, \&t_2$	$<?: \{l_1 : \perp,\ l_2 : \perp\},\ l_2 : \perp>$

(c) F_2 and F_3

Figure 8: Example 3.3

F_2 and F_3, and the tree representation of tree-expressions in F_2 and F_3. (End)

Phase III: The maximal phase

Let Π_k be the candidate-trie at the end of Phase II. Phase III prunes all non-maximally tree-expressions represented by leaf nodes of Π_k. One observation is that, for $i > j$, no i-tree-expression can be weaker than a j-tree-expression. For each j, we first find j-candidates 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, we add a j-candidate in M_j to the result only if it is not weaker than any candidate already in the result.

Example 3.4 Continue with Example 3.3. M_1 contains the three tree-expressions for p_1, p_3, p_4. M_2 contains the three tree-expressions for p_3p_1, p_3p_4, p_4p_1. M_3 contains the only tree-expression for $p_3p_4p_1$. All tree-expressions in M_1 and M_2 are weaker than the tree-expression in M_3. Thus, only one maximally frequent tree-expression is found, that is, $p_3p_4p_1$ or $<?: \{l_1 : \perp,\ l_2 : \perp\},\ l_2 : \perp>$. It can be verified from the OEM in Figure 5 that this is the only maximally frequent tree-expression. (End)

If we want to find all frequent tree-expressions, Phase III should be skipped and tree-expressions represented by all non-root nodes of Π_k should be returned.

Testing "weaker than"

To test whether a tree-expression te_1 is weaker than a tree-expression te_2 (as defined in Section 2), we need to search for a "match" of the tree te_1 inside the tree te_2. Recursively, a match is found for a non-leaf node v in te_1 if and only if matches are found for the label of v and for all subnodes of v. Unordered subnodes in te_1 require a complete bipartite match in te_2, and ordered subnodes require a sublist match in te_2. We omit the detail.

4 Discovery from a real dataset

To test if our algorithm acts as expected, we apply the algorithm to the Internet Movies Database (IMDb) at http://us.imdb.com/ for discovering typical structures of movies documents. IMDb currently covers more than 95,000 movies and over 1,300,000 filmography entries. All information about movies are organized into HTML document trees, with an example in Figure 9 for a segment about "Star Wars". We have chosen the top 100 most voted movies as classified by the source. We set

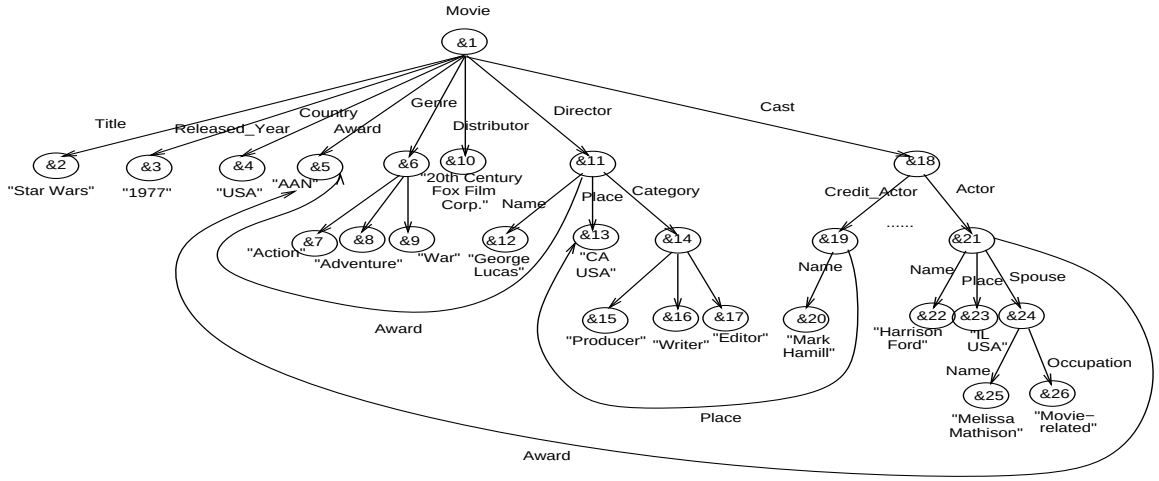


Figure 9: A segment of the "Star Wars" movie document

MINISUP to 15% and found the following three maximally frequent tree-expressions.

The first maximally frequent tree-expression, with 22% support, is given in Figure 10. (We have omitted all $\perp$ in the display of tree-expressions.) The tree-expression tells that one actor received award, four actors had spouses, two of which were doing movie-related jobs (because only movies-related occupations are documented), etc.

The second maximally frequent tree-expression, with the support of 17%, is the same as the first one, except that the last five lines are replaced by

```
?:{Name, Place, Spouse:{Name}, Category}}.
```

The wild-card ? represents any of *Writer*, *Composer*, *Cinematographer*, *Editor*, and *Producer*. This substructure was not discovered in the first maximally frequent tree-expression because no single occupation has the minimum support for this substructure, but the five combined together do. Since *Director* is explicitly captured, ? does not include *Director*.

The third maximally frequent tree-expression, with the support of 16%, is the same as the first one except that the line for *Director* is deleted and the last five lines are replaced with

```
?:{Name, Place, Award, Spouse:{Name,
Occupation}, Category}}.
```

The wild-card ? here covers *Director*, *Writer*, *Composer*, *Cinematographer*, *Editor*, and *Producer*. This substructure was not discovered in the second tree-expression because the separation of *Director* makes it not frequent enough.

These tree-expressions are maximally frequent, so their supports are not very high. Non-maximally frequent tree-expressions have much higher supports. For example, every movie document has labels: *Title*, *Release_Year*, *Country*, and *Director*. Discovered tree-expressions can be stored and retrieved later. For example, 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 the *Nationality* of cast). Often, it is necessary to keep track of the identifiers of movie documents supporting 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.

We have also conducted intensive experiments to study the efficiency of the algorithm for large collections of documents. Due to space limitation, the details are omitted.

5 Related Work

Our work is related to mining association rules from supermarket data [2, 3]. An example of association rules is "if a customer buys diaper, he/she also buys beer with 80% confidence". The core of the association rule problem is to find all subsets that are contained in at least some number of given sets. In [3], constructing larger candidate subsets is done by joining smaller frequent subsets, and counting support is done by set containment test. Our work has some important differences. First, a document contains subdocument references, which can be hierarchical, labeled, ordered, and cyclic. Second, a tree-expression has a tree-like structure, constructing tree-expressions and counting support require more than set containment test and join of flat sets. Third, the search space of structures is much larger than that of sets and requires more effective 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 structures from semistructured data. In [10] we considered discovering typical structures for building a structured layer above semistructured data to provide the benefits of standard database access methods. In the current paper, we exploit typical structures themselves for information retrieval. Also, in [10] no algorithm was given, and cyclic references and ordered references were not considered. [7] considered schema discovery for a single document. We considered the discovery task for a collection of documents, therefore, have to deal with the interestingness issue of structures. [9] has considered a collection of documents, as in our case, and derived a uniform object-oriented database schema for all documents. They first find the hierarchy for each document and merge them into a global schema. We do not construct such a global schema. Instead, we discover "typical" structures of doc-

```
Pattern 1 (support = 22%):
     {Title, Released_Year, Country, Award, Key, Distributor,
      Director:{Name, Place, Award, Spouse:{Name}, Category},
      Cast:{Actor:{Name, Place, Award, Spouse:{Name, Occupation}, Category},
            Actor:{Name, Place, Spouse:{Name, Occupation}, Category},
            Actor:{Name, Place, Spouse:{Name}, Category},
            Actor:{Name, Place, Spouse:{Name}},
            Actor:{Name, Place}},
      Writer:{Name, Place, Category},
      Cinematographer:{Name, Category},
      Composer:{Name, Category},
      Editor:{Name},
      Producer:{Name, Place, Category}}.
```

Figure 10: The first maximally frequent tree-expression, with support of 22%

uments for the purpose of information retrieval.

6 Conclusion

As the amount of documents available on-line grows rapidly, we find that most references to important fields are labeled and hierarchical, sometimes 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 methods treat a text document either as a set of words or as a list of words, and have not explored such structures of documents. We observed that many documents describing the same type of information are similarly structured, though not identically structured. Typical structures shared by a large number of documents can reveal the the general information content and representation in the source, and discovering typical structures is important for both the user and the builder of the source. We have defined the discovery problem and proposed a solution based on a new representation of search space. The effectiveness was evaluated on a real dataset. Traditional information access tends to emphasize the narrowly specified *querying* and the largely dis-oriented *browsing*. Our approach of mining typical structures of documents provides an alternative to information access that can overcome some limitations of these methods.

References

[1] S. Abiteboul, "Querying semi-structured data", ICDE 1997 (http://www-db.stanford.edu/pub/papers/icdt97.semistructured.ps)

[2] R. Agrawal, T. Imielinski, A. Swami, "Mining association rules between sets of items in large databases", SIGMOD 1993, 207-216

[3] R. Agrawal and R. Srikant, "Fast algorithms for mining association rules", VLDB 1994, 487-499

[4] P. Buneman, S. Davidson, G. Hillebrand, D. Suciu, "A query language and optimization techniques for unstructured data", SIGMOD 1996, 505-516

[5] A.O. Mendelzon, G.A. Mihaila, T. Milo, "Querying the World Wide Web", PDIS 1996 (available at ftp://ftp.db.toronto.edu/pub/papers/pdis96.ps.gz)

[6] S. Nestorov, S. Abiteboul, R. Motwani, "Inferring structure in semistructured data", Proceeding of the Workshop on Management of Semistructured Data, Tucson, May 1997, 42-48 (http://www.research.att.com/ suciu/workshop-papers.html)

[7] S. Nestorov, J. Ullman, J. Wiener, S. Chawathe, "Representative objects: concise representations of semistructured, hierarchical data", ICDE 1997 (http://www-db.stanford.edu/pub/papers/representative-object.ps)

[8] Y. Papakonstantinuo, H.Garcia-Molina, and J. Widom, "Object exchange across heterogeneous information sources, ICDE 1995, 251-260

[9] 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", Proceeding of the Workshop on Management of Semistructured Data, Tucson, May 1997, 54-59 (http://www.research.att.com/ suciu/workshop-papers.html)

[10] K. Wang, H.Q. Liu, "Schema discovery from semistructured data", International Conference on Knowledge Discovery and Data Mining, August, 1997, Newport Beach, 271-274

[11] The Workshop on Management of Semistructured Data 1997, Arizona (http://www.research.att.com/~suciu/workshop-papers.html