Discovering Frequent Substructures from Hierarchical Semi-structured Data

Gao Cong, Lan Yi, Bing Liu, Ke Wang[†]

Abstract: Frequent substructure discovery from a collection of semi-structured objects can serve for storage, browsing, querying, indexing and classification of semi-structured documents. This paper examines the problem of discovering frequent substructures from a collection of hierarchical semi-structured objects of the same type. The use of wildcard is an important aspect of substructure discovery from semi-structured data due to the irregularity and lack of fixed structure of such data. This paper proposes a more general and powerful wildcard mechanism, which allows us to find more complex and interesting substructures than existing techniques. Furthermore, the complexity of structural information of semi-structured data and the usage of wildcard make the existing frequent set mining algorithms inapplicable for substructure discovery. In this work, we adopt a vertical format for the storage of semi-structured objects, and adapt a frequent set mining algorithm for our purpose. The application of our approach to real-life data shows that it is very effective.

Keywords: frequent substructures mining, hierarchical semi-structured data, wildcard, web mining.

1 Introduction

Semi-structured data arise in many application areas. The emergence of XML further increases the availability of semi-structured data. See [1] for an excellent survey on semi-structured data. Figure 1 shows a segment of a semi-structured movie object, “God Father” from http://us.imdb.com/ Title?0068646. The root node represents the movie object and the other nodes represent its sub-objects. The links and their labels denote the sub-object references and their roles. The ***structure*** or ***schema*** of objects refers to the hierarchy of

[†] School of Computing, National University of Singapore, 3 Science Drive 2, Singapore 117543, {conggao, yilan, liub, wangk}@comp.nus.edu.sg.

references and roles. Note that the sub-object references of *writer* are omitted in Figure 1 because they are the same as those of *director*.

Semi-structured data by definition are characterized by the lack of rigid and fixed schema in advance. For example, some movie objects may not have *cast* label but only have *leading actor* label; some sub-objects may be missing in some movie objects, etc. Despite the structural irregularity, semi-structured data typically do possess some structures [11]. Such structures implicit in semi-structured data can serve the following purposes: optimizing query evaluation, obtaining general information contents, facilitating the integration of data from several information sources, improving storage, assisting in building indexes and views, and making it possible for structure-based document clustering [3][24].

Schema (or substructure) discovery for semi-structured data is a rather new and dynamic topic and by necessity incomplete [3]. The focus of schema discovery evolves with the progress of the research on semi-structured data. The system Lorel [2] and Tsimmis adopt OEM graphs as their semi-structured model and propose DataGuides [15] to explore and to represent the implicit schema of semi-structured data. [8][10][17] discover schema by classifying or clustering objects according to their structures.

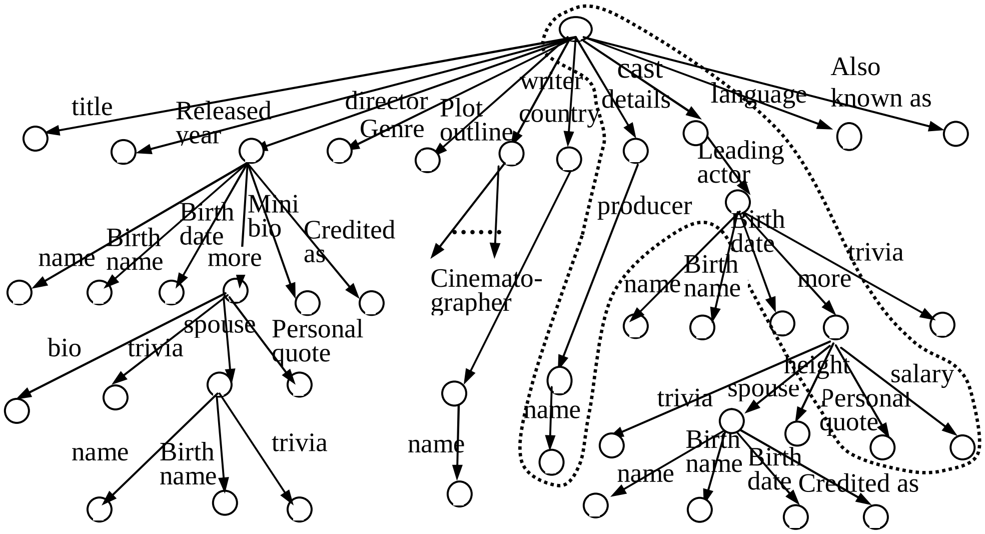


Figure 1: A segment of “God Father ” movie object

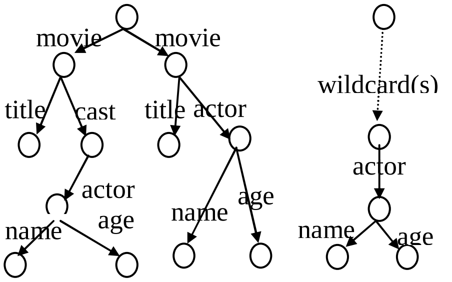
Recently, some researchers investigated the application of relational techniques to semi-structured data [26]. STORED [14] discovers the schema of semi-structured data and transforms it into relational schema. The part of data that fits the relational schema is stored in relational tables and the other part is stored with OEM graphs. [21] stores XML data entirely in relational tables with the assumption that the XML data conforms to a schema. One premise of the application of relational technology to semi-structured data is that semi-structured objects of the same type usually have similar structures.

The key in representing semi-structured data with relational tables is to discover the typical structures implicit in the data. Unfortunately, the hierarchical structure of semi-structured data makes the problem rather complex. Wildcard is widely used in semi-structured data query because of the data irregularity. Many interesting queries over semi-structured data necessarily involve wildcards [1] and the conjunction of path expression with wildcard can give the full power of path expression in query [2][3]. The importance of wildcard in structural discovery is the same as that in data query. However, the use of wildcard further complicates the problem of schema discovery.

In XML, although DTDs (Document Type Definitions) can be used to restrict what attributes and elements that XML documents can contain, attributes and elements included in DTD need not actually appear in the document. DTDs cannot specify restriction on the types of elements referenced by IDREF attributes [26]. Thus, frequent structures of XML documents may not be inferred from their DTDs.

[24] proposes a modified frequent set mining algorithm to discover frequent substructures from a collection of semi-structured data represented with OEM graphs. [14] adopts the approach of [24] to find the frequent structures from XML data. Because of the definition and usage of wildcard, [24] uses a tree match algorithm to count supports of candidate substructures. The efficiency of tree matching algorithm becomes worse and even prohibitive as the number of wildcards in candidate substructures increases.

Moreover, the usage of wildcard in [24] cannot fully explore the structure of irregular semi-structured data. For example, both movie objects in Figure 2a) contain the substructure as shown in Figure 2b). Given the task of finding substructures that occur in both movie objects in Figure 2a), [24] cannot discover substructures in 2b). The reasons are as follows:



a) two movie objects b) a substructure

Figure 2 some movie objects and their substructures

A wildcard (denoted by ?) in [24] matches only one label on a label path. With such wildcards, if the wildcard of 2b) is composed of two "?", 2b) is matched with the left movie object of Figure 2a); if the wildcard of 2b) is composed of only one "?", 2b) is matched with the right movie object of Figure 2a). In both cases, 2b) can only occur in one movie object. Therefore, [24] cannot find 2b). [24] cannot discover typical substructures that exist in the lower part or the middle part of semi-structured objects when the upper part of objects has different number of labels.

The introduction of wildcard in [24] may cause over-generation of paths with wildcard (to form substructures) because it generates paths with wildcard by replacing one or several non-terminal labels on each label path with the corresponding number of "?". This will generate a large number of useless paths with wildcard, and increase the search space drastically.

Our goal in the paper is to discover frequent structures that occur in a minimum number of a collection of semi-structured objects specified by the user. The collection of objects usually contain the same type of information and are similarly structured. Examples of such kind of semi-structured objects are movies, universities, census data, online merchandise, etc. In this sense, our work is related to those of [13][24]. However, our approach differs in important ways. The main characteristics of our approach are as follows:

1) With a more powerful wildcard mechanism, our approach can overcome the shortcomings of [24] as mentioned above, thus exploring the structure of irregular semi-structured data more effectively than previous techniques. E.g. our approach can discover substructures in Figure 2.

2) We propose a new approach to compute the frequencies of substructures (with wildcard or not) in a collection of semi-structured data. The approach is characterized by the new features: semi-structured objects are represented with paths and corresponding *tidlist*s in database; the database is expanded with another two components---paths with

wildcard and a category of substructures; a special *tidlist* format for the two components is adopted. Therefore, Our approach avoids expensive computation of tree matching.

3) We propose an effective way to introduce path with wildcard that avoids over-generation of paths with wildcard (used to construct candidate substructures).

4) We propose an adapted mining algorithm to meet the new requirements of substructure discovery problem: the hierarchical structure of semi-structured objects, the usage of wildcard and the special *tidlist* format used in our approaches.

The algorithms for discovering frequent itemsets from vertical format databases, such as [9][16][19][20][27], are related to our approach. These algorithms, however, cannot be directly applied to objects having structures in the form of labeled hierarchical sub-object references, much less to discover substructures with wildcards. The technique in [22] on pattern discovery from semi-structured data cannot be used either as it does not consider the hierarchical structure of semi-structured objects. In addition, our search space includes substructures containing wildcards, which [22] does not handle. Another related work is [12] that accurately estimates the number of matches of a small tree in a large node-labeled tree. [12] does not deal with wildcards.

The paper is organized as follows: Section 2 defines our discovery problem. Section 3 presents the proposed algorithm. Section 4 evaluates the efficiency of algorithm and applies our algorithm to real-life semi-structured datasets. Section 5 concludes the paper.

2 Problem Description

Section 2.1 introduces the scheme for representing transaction objects for our discovery problem. Another two components are presented in Section 2.2 to expand database. Section 2.3 discusses how to construct substructure by combining components in database and define *path-set* to represent substructures (as well as transaction objects) and *weaker than* relationship to compare the informativeness of path-sets. Section 2.4 defines our discovery problem and gives several examples.

2.1 Representing transaction objects

Our objective in this work is to discover structural similarity of a collection of semi-structured objects. These objects are also called *transaction objects*. Each object can be an acyclic or cyclic graph. A cyclic graph can be transformed into an acyclic graph and an acyclic graph can be equally represented with a tree through replicating shared sub-objects [24]. Assume that the transaction set T consists of m tree objects, $t_1, t_2, \ldots, t_m$. The structure of a tree is represented with a tree of labels, called *tree-representation* below.

Definition 2.1 (tree-representation): For a leaf (or terminal) node, its tree-representation is *null*. For a non-leaf node which has h sub-objects $o_1, o_2,\ldots, o_h$ at the next level with the labels $l_1, l_2,\ldots, l_h$, its tree-representation is $\{l_1\text{: } tr_1, l_2\text{: } tr_2, \ldots, l_h\text{: } tr_h\}$, where tr_j is the tree-representation of o_j ($1\leq j \leq h$).

Example 2.1: The tree-representation of the part of movie object enclosed by the dotted line in Figure 1 is: {*Details*: {*Producer*: {*Name*}}, *Cast*: {*Leading actor*: {*Birth date*, *More*: {*Personal quotes*, *Salary*}}}}.

Definition 2.2 (label path): A label path p is a path starting from the root of the tree and is represented with a sequence of its labels, $[l_1, l_2, \ldots,l_n]$, where l_j is a label and n is the length of p.

Definition 2.3 (pre-paths of a label path): A label path $p = [l_1, l_2, \ldots,l_n]$ has n-1 pre-paths ($n > 2$), where each pre-path is of the form, $[l_1,\ldots, l_s]$, where $1 \leq s \leq n - 1$.

In the proposed technique, each transaction object t ($1 \leq t \leq m$) is represented and stored with all its paths $P_t = \{P_{t,leaf}, P_{t,pre}\}$, where $P_{t,leaf}$ is the set of label paths from the root to all leaf nodes, and $P_{t,pre}$ is the set of pre-paths of all paths in $P_{t,leaf}$. Note that such a representation may result in the loss of information when some children have the same label. The problem is solved in Section 3.1.

Table 1: Representation of dataset in example 2.2

No	Label Path	Tidlist
p1	Details	1,2
p2	Details, Producer	1
p3	Details, Producer, Name	1
p4	Cast	1
p5	Cast, Leading actor	1
p6	Cast, Leading actor, Birth date	1
p7	Cast, Leading actor, More	1
p8	Cast, Leading actor, More, Personal quote	1
p9	Cast, Leading actor, More, Salary	1
p10	Details, Cinematographer	2
p11	Details, Cinematographer, Name	2
p12	Director	2
p13	Director, Birth date	2
p14	Director, More	2
p15	Director, More, Personal quote	2
p16	Director, More, Salary	2

Let $P = \bigcup_{t=1,\ldots,m} P_t$. Using the paths in P, we produce a database D. Each path p in P forms a tuple in our database D, and is represented with <*p*, *tidlist*>, where $p \in P$ and *tidlist* is the set of transaction objects or trees (represented with their *id*'s) that contain p. Because our transaction objects are similarly structured, the representation of objects with label paths will usually save a lot of space than representing them as trees.

Example 2.2: Let transaction object t_1 be the tree-representation in Example 2.1, and transaction object t_2 = {*details*: {*Cinematographer*: {*Name*}}, *Director*: {*Birth date*, *More*: {*Personal quote*, *Salary*}}}. Table 1 shows how the two objects are stored in the database. For example, t_1 contains paths p1-9, which are shown in their *tidlist*s. For t_1, the paths p3, p6, p8 and p9 are in $P_{t_1,leaf}$ and paths p1, p2, p4, p5 and p7 are in $P_{t_1,pre}$.

2.2 Two Other Components for constructing substructures

The paths in database D alone are not sufficient for constructing some interesting substructures with wildcard (we will see that most of the interesting substructures in examples of Section 3 & 4 contain wildcards). Two other necessary components for constructing substructures with wildcard are single paths with wildcards, and a special kind of subtrees composed of paths with wildcard, which are discussed below.

Single paths with wildcard. Instead of using ? to represent a wildcard as in [24] that only matches one label, we use symbol * to represent wildcard that can match one or more labels in the upper part of label paths. For a label path $p = [l_1, l_2, \ldots, l_n]$, where $p \in P$ and n is the number of labels in path p, * can replace any of its pre-path p_j to form a new path p'. But a wildcard cannot replace the last label l_n of path p because the use of wildcard is to meet the requirements: 1) To find something common in the lower parts of semi-structured objects while ignoring their upper parts. 2) To find something common in the middle parts of some semi-structured objects from pre-path set while ignoring their upper parts. p' is represented with $[*, l_{j+1}, \ldots, l_n \mid \{u_1, u_2, \ldots, u_e\}]$, where e is the number of actual paths in D covered by p' and u_k ($1 \le k \le e$) is a path covered by *. Let U be the set of u_k, $1 \le k \le e$. We require $e > 1$ because wildcard is meaningless if p' does not cover more than one path.

All generated paths with wildcard are added to the database D. Note that we do not generate paths with wildcard by simply replacing every pre-path with a wildcard, which will result in a large number of useless paths for substructure discovery. The paths with wildcard have a special *tidlist* format in database. The method of generating paths with wildcard and the representation of paths with wildcard in database are presented in Section 3.1.

Example 2.3: Continue with Example 2.2. After the introduction of wildcard, we obtain the following paths with wildcard: p17 = [*, *Name* | {p2, p10}], which covers p3 and p11. p18 = [*, *Birth date* | {p5, p12}], which covers p6 and p13. p19 =[*, *More* | {p5, p12}], which covers p7 and p14. p20 = [*, *More*, *Personal quote* | {p5, p12}], which covers p8 and p15. p21 = [*, *More*, *Salary* | {p5, p12}], which covers p9 and p16. The * in p17 covers p2 and p10. The * in p18, p19, p20, p21 covers p5 and p12.

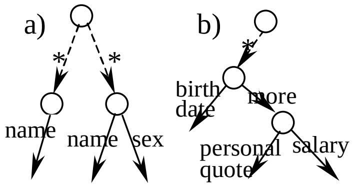


Figure 3: Subtrees with wildcard in transaction obiects

The introduction of paths with wildcard as an addition to the original database *D* is essential to construct some interesting substructures. But it is still not sufficient. Considering the case: a single path with wildcard appears multiple times in a substructure, e.g., in Figure 3a), the path [*, name] appears twice in the substructure. We do not permit self-joining of paths to form substructures because it will increase search space dramatically. Therefore we cannot construct such kind of substructure just by combining paths with wildcards. In order to construct such kind of substructures, we need another component, a kind of subtrees composed of paths with wildcard. Figure 3b) shows an example of such a subtree, which is composed of three paths from example 2.3: p18, p20 and p21.

Special subtrees (or *ST* in short). Intuitively, the *subtree* in the form of Figure 3b) requires that * of the component paths represent the same path in the same transaction object (tree). For different transaction objects, the * may represent different paths. In addition, it is meaningless to generate a subtree by combining paths whose label sequence contains one another, e.g., p19 and p20 in example 2.3. Formally, consider *m* (*m* > 1) paths with wildcard p_1'= [*, β_1 | U_1], p_2'= [*, β_2 | U_2], …, p_m'= [*, β_m | U_m], where β_i may represent multiple labels and β_1,.., β_m can be combined into a tree. Let $U_f = \cap_{i=1,\ldots,m} U_i$ and $s_f = |U_f|$. If $s_f > 1$ and any label sequence β_i is not the upper part of the other label sequence β_j, the *m* paths can be combined to generate a subtree in the form of Figure 3b) and the * in the tree covers paths in U_f. If $s_f = 1$, * covers only one path. Then, there is no need to use wildcard. For the rest of the paper, *ST* is used to denote subtree in the form of Figure 3b).

Example 2.4: Continue with example 2.3. paths p18 and p19 can form a *ST* tr_1 = {*: {*Birth date*, *More*}} because |p18.*U* ∩ p19.*U*| = |{p5, p12}| > 1. In the same way, *ST* tr_2 ={*: {*Birth date*, *More*: {*Personal quote*, *Salary*}}}, as shown in Figure 3, can also be formed. Although |p19.*U* ∩ p20.*U*| = |{p5, p12}| > 1, they cannot form a *ST* because the label sequence of p19 is contained in that of p20.

With the three categories of components, namely, paths without wildcard (the original tree paths), paths with wildcard, and *ST*s, we obtain the final database for substructure discovery. Next, we introduce how to combine the three types of components to form candidate substructures. Note that determining whether a substructure (containing *ST*s or paths with wildcard) is contained in a transaction object is a nontrivial problem and require paths with wildcard and *ST*s to have a special format of *tidlist* in database. This problem is discussed in Section 3.

2.3 Constructing Substructure

Because a single path without wildcard contains the structural information of its pre-paths, a path with wildcard covers several paths without wildcard and a *ST* consists of a number of paths with wildcard, the structural information contained in the three kinds of components are partly overlapping. As a result, not any combination of them is meaningful, and the complete set of combinations of them will result in over-generation of candidate substructures. For instance, in example 2.2 - 2.4, the combination of p5 and p6 is not meaningful because p5 is a pre-path of p6, and neither is the combination of tr_1, p6 and p13 because the component p18 of tr_1 can only represent p6 or p13, i.e., the structural information of tr_1 contains that of p6 or p13 in any transaction object.

The challenge is how to choose a meaningful combination of paths and *ST*s while considering the computational complexity. Intuitively, we require that the structural information of one component should not contain that of the others. For example, for substructure {*: {*Name*}, *: {*Birth date*, *More*}}, composed of two components: a path with wildcard and a *ST*, if the substructure is contained in some transaction object, then paths represented by [*, *Name*] and paths represented by the *ST* should not contain each other. We define *path-set* to stipulate and represent substructures discovered in our approach. We need the operator $\overset{\backslash}{\neq}$ to explain the concept of *path-set*. Two paths p_1 and p_2 satisfy $p_1 \overset{\backslash}{\neq} p_2$ if p_1 is not equal to p_2 and p_1 is not a pre-path of p_2, and vice versa.

***k*-path-set.** It is a set $ps = \{P_o, P_w, S_{st}\}$ with the size k, where P_o, P_w, S_{st} are three sets and meet the following requirements:

- P_o is a set of single paths without wildcard. Suppose there are x (≥ 0) such paths $p_{o1}, \ldots, p_{ox}$ in ps.
- P_w is a set of single paths with wildcard. Suppose there are y (≥ 0) such paths $p_{w1}, \ldots, p_{wy}$ in ps.
- S_{st} is a set of *ST*s. Each st_i in S_{st} is now represented with $\{p_{i1}, p_{i2}, \ldots, p_{il} \mid U_i\}$, where p_{ij} ($1 \leq j \leq l$) is a path that forms st_i and U_i is the set of paths covered by *. Suppose there are z (≥ 0, $x+y+z = k$) such *ST*s $st_1, \ldots, st_z$ in ps.
- There exists a set $\{p_{o1}, \ldots, p_{ox}, u_1, \ldots, u_y, c_1, \ldots, c_z\}$, where $u_i \in p_{wi}.U$ ($1 \leq i \leq y$) and $c_j \in st_j.U$ ($1 \leq j \leq z$), such that any two paths α, β of the set meet the requirements: $\alpha \overset{\backslash}{\neq} \beta$.

We should point out that the concept of path-set imposes a restriction on the components of a substructure than necessary. But any relaxation of the restriction would cause the complexity of the discovery problem to increase drastically. The adverse effect of our restriction is that we ignore one kind of combination: the paths represented by wildcard contain one another, but the corresponding full paths represented by paths with wildcard do not. Substructures from such kind of combinations are uncommon and structural information reflected by them is usually contained in some other discovered substructures. See [13] for the detailed analysis. Even if we ignore such kind of combinations, our approach can discover more substructures than existing techniques.

A tree can be constructed from a path-set by recursively combining paths and *ST*s sharing the same next label l_s into a branch labeled l_s [24]. Note that the wildcards in different components of the same path-set represent different label sequence so that these wildcards are regarded as different labels. Our approach does not rely on tree-representation, which is used for easy understanding. Instead, both transaction objects and discovered substructures are represented with path-sets in our algorithm. Example 2.5 shows how to represent transaction objects and substructures with path-sets.

Example 2.5: In example 2.2, t_1 is expressed as a 4-path-set {p3, p6, p8, p9} and t_2 is also expressed as a 4-path-set {p11, p13, p15, p16}. In example 2.4, the 1-path-set of tr_1 is ps_1 = {{p18, p19 | {p5, p12}}}, and the 1-path-set of tr_2 is ps_2 = {{p18, p20, p21 | {p5, p12}}}. p17 and p18 can form 2-path-set ps_3 = {p17, p18} and its tree-representation is tr_3 = {*: {*Name*}, *: {*Birth date*}}. p18 and p19 can also form a 2-path-set ps_4 ={p18, p19}, whose tree-representation is tr_4 = {*: {*Birth date*}, *: {*More*}}, where if the first * represents p5, the second * can only represent p12; if the first * represents p12, the second can only represent p5.

Given some path-sets, we are interested in the most "informative" one. For example, ps_2 is more informative than ps_1. The "weaker than" relationship below compares the informativeness of path-sets. Intuitively, A is weaker than B if all the structural information of A can be found in B.

Weaker than: Any path-set ps is weaker than itself.

1) For two single paths p and q: If both p and q are paths without wildcard, path q is weaker than p if q is a pre-path of p; If both p and q are paths with wildcards, path q is weaker than p if the label sequence of q is a prefix of that of p; If q is a path with wildcard and p is an original path, q is weaker than p if q covers p. In example 2.3, p1 is weaker than p2, p19 is weaker than p20, and p20 is weaker than p8.

2) For two 1-path-set ps_p = {p} and ps_q ={q}, where $p = \{p_1', p_2', \ldots, p_k'|U_k\}$ and $q = \{q_1', q_2', \ldots, q_l'|U_l\}$ are *ST*s: ps_q is weaker than ps_p if q_i' is weaker than some p_{j_i}' for each $1 \le i \le l$, and $\{p_{j_1}', p_{j_2}', \ldots, p_{j_l}'\}$ is a subset of $\{p_1', p_2', \ldots, p_k'\}$. In example 2.5, ps_1 is weaker than ps_2.

3) For path-set $ps_p = \{p_1, p_2, \ldots, p_m\}$ and 1-path-set $ps_q =\{q\}$, where $q =\{q_1', q_2', \ldots, q_l'|U_l\}$ is *ST*: ps_q is weaker than ps_p if q_i' is weaker than some p_{j_i} for each $1\le i \le l$, $\{p_{j_1}, p_{j_2}, \ldots, p_{j_l}\}$ is a subset of $\{p_1, p_2, \ldots, p_m\}$ and $p_{j_1}, p_{j_2}, \ldots, p_{j_l}$ have some common pre-path $u \in U_l$. In example 2.5, ps_1 and ps_2 are weaker than transaction t_1 and t_2 respectively.

4) For two path-sets $ps_p = \{p_1, p_2, \ldots, p_m\}$ and $ps_q =\{q_1, q_2, \ldots, q_n\}$: ps_q is weaker than ps_p if for each $1 \le i \le n$, q_i is weaker than some subset of ps_q and these subsets do not overlap. In example 2.5, ps_3 is weaker than transaction t_1 and t_2. But ps_4 is not weaker than any transaction object.

2.4 The Discovery Problem

Consider a set of transaction objects T and a substructure expressed as a path-set ps. When ps is weaker than the path-set of a transaction object t from T, then t becomes an element of a set S. The support of ps is the percentage of $|S|$ over $|T|$. A substructure is *frequent* if its support is not less than the user-defined *Minsup* (minimum support). A substructure is *maximally frequent* if it is frequent and is not weaker than any other frequent substructures. The *substructure* (*schema*) *discovery problem* is defined as finding all frequent substructures contained in the set of transaction objects T. The *maximal discovery problem* is to find all maximally frequent substructures. Using the discovered frequent path-set, one can derive association rules about substructures of objects.

Example 2.6: Continue with Example 2.2. Consider the path-set {p1, p4} representing {*Details*, *Cast*}, its support is 1/2. Given *Minsup* = 2/2, only path-set {p1} is frequent with support 2/2.

Example 2.7: Continue with Example 2.5. The supports of path-set ps_1, ps_2, ps_3 are all 2/2. They are frequent given *Minsup* = 2/2. Because ps_1 is weaker than ps_2, ps_1 is not maximal frequent. ps_2 and ps_3 are not maximal frequent because they are weaker than path-set {p17, p19, p21, p22} whose tree-representation is {*: {*Name*}, *: {*Birth date*, *More*: {*Personal quote*, *Salary*}}} with support 2/2. {p17, p19, p21, p22} is maximal frequent. The support of ps_4 is 0.

3 The Algorithm

The Section presents the algorithm for our discovery problem defined in Section 2.3. We first discuss the preparation of database for substructure discovery, which includes encoding label path, and generating paths with wildcard. Then the algorithms for finding both frequent *ST*s and frequent final substructures are presented.

3.1 Preprocessing

The section proposes the methods of coding label paths and introducing paths with wildcard and the special format of *tidlist*s for paths with wildcard.

Coding label: The labels of paths are coded with integers. By replacing each label with its corresponding integer and using "," as the delimiter to separate the code of each label, we represent each label path with a delimited string. Then, the "weaker than" check between two paths can be done by string matching. In order to make our algorithm easier to understand, we still use label texts instead of integers in the rest of the paper.

Example 3.1: Figure 4 shows the structural information of three student homepages. Table 2 shows the path codes. Path [*English version*, *Contact*] is weaker than path [*English version*, *Contact*, *Email*] because path code "10, 12" is a prefix of path code "10, 12, 9".

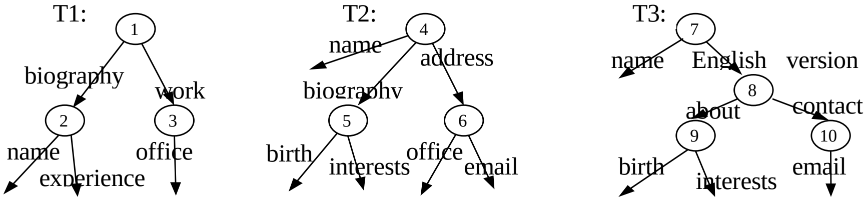


Figure 4: The structural trees of three student homepages

Table 2: The path code of paths in figure 4.

No	Path code	Label path	Tidlist
p1	1	[biography]	{1, 2}
p2	1, 2	[biography, birth]	{1, 2}
p3	10, 11, 2	[English version, about, birth]	{3}
p4	1, 3	[biography, experience]	{1}
p5	4	[work]	{1}
p6	4, 5	[work, office]	{1}
p7	8, 5	[address, office]	{2}
p8	6	[name]	{2, 3}
p9	1, 7	[biography, interests]	{2}
p10	10, 11, 7	[English version, about, interests]	{3}
p11	8	[address]	{2}
p12	8, 9	[address, email]	{2}
p13	10, 12, 9	[English version, contact, email]	{3}
p14	10	[English version]	{3}
p15	10, 11	[English version, about]	{3}
p16	10, 12	[English version, contact]	{3}

When a node has several repeating children nodes, its children with repeating labels are distinguished by suffixing the label code. These children form a new transaction set and the paths containing labels below these children are removed from the set *P* and database *D*. For example, for the part of tree-representation of a movie object: {*cast*: {*actor*: {*name*, *birth*}, *actor*: {*name*, *sex*}}}, assume that label *cast* is coded with 5 and *actor* is coded with 6. The *actor* sub-objects will form a new transaction set to discover frequent substructures for *actor*s. The paths containing labels below *actor* (i.e., *name*, *sex* and *birth*) are removed from the movie dataset. The two children *actor* labels remain in the movie dataset and are coded as 6.1 and 6.2 to distinguish them from each other.

Introducing wildcard: A wildcard is used to match the upper part of some paths in *P* (or *D*) with a common lower part. We introduce wildcard as follows: If several paths in *P* have the same terminal label, these paths will form a bin. If at least 2 paths, a proper subset of the bin, have longer common lower part, the subset forms a new bin. The new bin is checked until there is no proper subset with at least 2 paths and longer common lower part. The above process can be implemented with a modified bin sorting algorithm. After sorting, those paths with the same lower part will be in a bin. Those bins containing only one path are discarded as no wildcard is needed in the case. From each bin containing more than 1 path, we obtain a candidate path with wildcard in the form of [*, the common lower part of paths in the bin]. Considering three paths, p_1 = [1, 4, 5], p_2 = [2, 4, 5] and p_3 = [3, 6, 5]. Two bins are generated and two candidate paths with wildcards are p_4 = [*, 5] and p_5

=[*, 4, 5], where p_4 covers p_1, p_2 and p_3, and p_5 covers p_1 and p_2. If p_3 does not exist, only p_5 is formed.

The support of a candidate path (formed from a bin) is the size of the union set of *tidlist*s of paths in the bin. If a candidate path is frequent, a path with wildcard is generated. Otherwise, it is discarded.

***Tidlist*s of paths with wildcard:** Let P' be the set of paths with wildcard. All paths in P' are frequent. Each path p' in P' has a corresponding *tidlist*. Each *tid* in the *tidlist* of p' appears in one or several *tidlist*s of paths (without wildcard) covered by p'. Such information is essential for determining whether a substructure is contained in a transaction object and will be recorded in the *tidlist* of p'. The *tidlist* of p' is expressed as $tidlist' = \{t_1':H_1', t_2':H_2', \ldots, t_m':H_m'\}$, where m is the number of *tid*'s in $tidlist'$ and each H_i' ($1< i < m$) is the set of $p'.u_k$ ($p'.u_k \in p'.U$), where the *tidlist* of the original (without wildcard) path corresponding to each $p'.u_k$ contains t_i'. All paths in P' and their *tidlist*s produce a database D'.

Example 3.2: Continue with example 3.1. After sorting, paths p2 = [*biography*, *birth*] and p3 = [*English version*, *About*, *Birth*] are in the same bin with common lower part labeled "*birth*". Given *Minsup* = 2/3, A new path p17 = [*, *birth* | {p1, p15}] is introduced. See Table 3 for the other generated paths.

Table 3: Generated paths with wildcard

No	Label path	Tidlist	Support
p17	[*, birth | {p1, p15}]	{1:{p1}, 2:{p1}, 3:{p15}}	3/3
p18	[*, office |{p5, p11}]	{1:{p5}, 2:{p11}}	2/3
p19	[*, interests | {p1, p15}]	{2:{p1}, 3:{p15}}	2/3
p20	[*, email |{p11, p16}]	{2:{p11}, 3:{p16}}	2/3

After pre-processing, we obtain two databases: D for paths without wildcard and D' for paths with wildcards. Note that P is the set of paths in D and P' is the set of paths in D'.

3.2 Overview of the algorithm

The core of the algorithm is to compute all frequent *k*-path-sets. Our technique is based on the partition algorithm for association rule mining given in [19]. However, what we find are substructures, not itemsets without structure. [19] divides the horizontal database into a number of non-overlapping partitions and scans database only twice. Once for generating a set of potential frequent itemsets, and once for gathering their supports.

Since our intention is to investigate how to solve the schema discovery problem, in this paper we focus on discovering frequent structures in one partition. The method for support counting given in this paper can also be applied to compute the global support when all partitions are merged to generate global frequent substructures.

In order to find all frequent final substructures, we should first find all frequent *ST*s from the set of paths with wildcard, P'. The discovered *ST*s (1-path-set) and paths in P' (or D'), together with their *tidlist*s, are added back to the original database D. With the new database D, we discover all frequent final substructures.

One important property that forms the foundation of our algorithm is the **downward closure property:** If a *k*-path-set $\{p_1, p_2, \ldots, p_k\}$ is frequent, then any *k*-1 subset of $\{p_1, p_2, \ldots, p_k\}$ is also frequent. This property holds because a subset is weaker than its superset and because the "weaker than" relationship is transitive.

3.3 Generating frequent *ST*s

The objective of this step is to discover all frequent *ST*s from D'. We use *r*-path-structure ($r > 1$) to represent these *ST*s. A *r*-path-structure ($r > 1$) denotes a *ST* composed of *r* paths with wildcard. Note that a 1-path-structure ($r = 1$) denotes a path with wildcard, but not a *ST*.

Because all paths in *P*′(or *D*′) are frequent, the set of frequent 1-path-structures is *P*′. Let *Q* be the set of all frequent *r*-path-structures (including both paths with wildcard and *ST*s). Each *q* in *Q* is expressed as {*Pno* | *U*}, where *Pno* is the set of paths forming *q*, and *U* is the set of paths covered by the wildcard. Each *q* has a *tidlist* that is the set of *tid*'s that *q* is weaker than. The format of *tidlist* of *q* is the same as that of a single path with wildcard.

The set of frequent *r*-path-structure is denoted by F_r. The detailed algorithm is given in Figure 5. Lines 3 - 7 show the candidate generation process. Lines 9-12 show how to generate the *tidlist* of candidate *c*. The prune step (line 8) is performed from two aspects: One is done based on the downward closure property, the other is that we require that a *r*-path-structure candidate form a tree with *r* leaf nodes. When *r* = 2, a 2-path-structure candidate is pruned if there exists "weaker than" relation between $f_1[1]$ and $f_2[1]$. When *r* > 2, there is no need for such a check because it has been done during the check for *r* = 2.

```
1) $F_1 = P'$
2) for $(r = 2; F_r \neq \Phi; r++)$ do
3)    forall $path-structures$ $f_1 \in F_{r-1}$ do
4)       forall $path-structures$ $f_2 \in F_{r-1}$ do begin
5)          $U = f_1 \cdot U \cap f_2 \cdot U;$   $n = |U|$
6)          if $(n > 1) \wedge (f_1[1] \cdot pno = f_2[1].pno \wedge \Lambda \wedge f_1[r-2] \cdot pno = f_2[r-2].pno \wedge f_1[r-1] \cdot pno < f_2[r-1].pno)$ then begin
7)             $c = \{f_1[1], \Lambda, f_1[r-1], f_2[r-1]\}$
8)             if $c$ $cannot$ $be$ $pruned$ then begin
9)                for each $t \in f_1 \cdot tidlist \wedge t \in f_2 \cdot tidlist$ do
11)                  if $t.H_{f_1} \cap t.H_{f_2} \neq \Phi$ then $add$ $t$ $to$ $c.tidlist;$   $add$ $t.H_{f_1} \cap t.H_{f_2}$ $to$ $c.tidlist[t].H$
12)               if $|c.tidlist| / |T| \geq Minsup$ then $F_r = F_r \cup \{c\}$
13)            endif
14)         endif
15)   endfor
```

Figure 5: Algorithm for generating frequent *ST*s

Example 3.3: Continue with example 3.2. Let *Minsup* = 2/3. F_1 = {p17, p18, p19, p20}. Please refer to Table 3 for F_1. See [13] for the generation process of F_2 (shown in Table 4).

No	Path-set	Tree-representation	Tidlist	Support
p21	{{p17, p19|{p1, p15}}}	{*: {birth, interests}}	{2:{p1}, 3:{p15}}	2/3

Table 4: F_2 in example 3.3

All elements in *Q* (containing both single paths with wildcard and *ST*s), together with their *tidlist*s, are added to the original database *D*. Each element in *Q* forms a tuple in the new database *D* (containing *D*′).

3.4 Generating frequent final substructures

This step generates all frequent final substructures, expressed as path-sets, from the new database *D* (obtained at the end of Section 3.3). The set of frequent *k*-path-set is denoted by F_k'. The frequent 1-path-set, i.e., F_1' includes the paths in *P* that have minimum support and all *q* in *Q*. The detailed algorithm is given in Figure 6. Lines 9-11 show the process of counting support. For a candidate *c*, generated from two (*k*-1)-path-sets, its components are divided into two parts c_p ($c_p \subseteq P$) and c_q ($c_q \subseteq Q$). Let $m = |c_p|$ and $n = |c_q|$.

If *n* = 0 (line 9), *c*'s *tidlist* is obtained by intersecting the *tidlist*s of f_1 and f_2. The support of *c* is the size of its *tidlist*.

If *n* > 0, we do not generate *tidlist* for *c* and only count its support because its *tidlist* is useless for later algorithm. To count support, we need to determine whether the *k*-path-set represented by *c* is contained in a transaction object *t*, i.e., *c* is weaker than *t*. If a transaction object *t* does not appear in all *tidlist*s of *c*'s component, *c* is not weaker than *t*. However, a *t* appearing in all *tidlist*s of *c*'s components does not mean that *c* is weaker than *t* because a path-set is not a simple set of components. The count is handled in the following two cases:

1) If $n = 1$ (line 10), one of f_1 and f_2 does not contain any path with wildcard, i.e., $c_p = f_1$ or f_2. If any $u \in c_q.U$ does not satisfy $p_i \not\succeq u$ ($p_i \in c_p$, $1< i <m$), *u* becomes an element of set *V* and is removed from $c_q.U$. Then, an element of $c_q.tidlist[t].H$ is deleted if it appears in *V*, where *t* is any *tid* in $c_q.tidlist$. If $c_q.tidlist[t].H$ becomes *null*, *t* is deleted from $c_q.tidlist$. The support counting is computed by intersecting $c_p.tidlist$ and the adjusted $c_q.tidlist$ (after deletion).

2) If $n > 1$ (line 11), for each transaction *t* involved in the *tidlist* of all components of *c*, we can enumerate $q_i.tidlist[t].H$ ($q_i \in c_q$), i.e., paths covered by wildcard for each q_i, to see whether there is a combination $p_1, \ldots, p_n, q_1.u_{q_1}, \ldots, q_m.u_{q_m}$ that meets the requirements of *k*-path-set (since *n* and the size of $q_i.tidlist[t].H$ are usually small, the enumeration is fast). If so, *c* is weaker than *t*.

```
1) $F_1' = \{frequent \quad 1-path-sets\}$
2) for ($k = 2; F_k' \neq \Phi; k++$) do
3)   forall $path-sets \quad f_1 \in F_{k-1}'$ do
4)     forall $path-sets \quad f_2 \in F_{k-1}'$ do begin
5)       if $f_1[1] \cdot pno = f_2[1].pno \wedge \Lambda \wedge f_1[k-2] \cdot pno = f_2[k-2].pno \wedge f_1[k-1] \cdot pno < f_2[k-1].pno$) then begin
6)         $c = \{f_1[1], \Lambda, f_1[k-1], f_2[k-1]\}$
7)         $divide \quad c \quad into \quad c_p, c_q (c_p \subseteq P, c_q \subseteq Q); n = |c_q|; m = |c_q|$
8)         if $c \quad cannot \quad be \quad pruned$ then begin
9)           if $n = 0$ then $c.tidlist = f_1 \cdot tidlist \cap f_2 \cdot tidlist; num = |c.tidlist|$
10)            elseif $n = 1$ then $num = count(c_p \cdot tidlist \overline{\cap} c_q.tidlist)$
11)            else $num = count((c_p[1] \cdot tidlist \cap \Lambda \cap c_p[m] \cdot tidlist) \overline{\cap} c_q[1] \cdot tidlist \overline{\cap} \Lambda \overline{\cap} c_q[n] \cdot tidlist)$
12)            if $num / |T| \geq Minsup$ then $F_k' = F_k' \cup \{c\}$
13)          endif
14)       endif
15)    endfor
```

Figure 6: Algorithm for generating all frequent substructures

The following pruning strategies are applied to the algorithm (line 8): 1) If any subset of a candidate is not frequent, the candidate can be pruned. 2) If there exists "weaker than" relation between two paths without wildcard in a candidate, the candidate can be pruned.

Example 3.3: Continue with example 3.2. All elements in F_1, F_2, together with their *tidlists* are added back to the original database in Table 2. The frequent 1-path-set $F_1' =$ {p1, p2, p8, p17, p18, p19, p20, p21}. See [13] for the generation process of $F_3'$and F_2', which is rather involved. Table 5 shows F_2' and F_3'.

In Table 5, {p8, p20, p21} and {p2, p18} are maximally frequent, and the others are not. In addition, the algorithms in [7][9][27] can be adapted to our maximal discovery problem. See [13] for details about the maximal discovery problem and some optimizations to our algorithm.

4 Experiment Results and Application

We applied our system to two internet websites: one is the Internet Movies Database (IMDb) at us.imdb.com, and the other is the World Travel Guide (WTG) at www.wtgonline.com. IMDb catalogs all kinds of information on over 250,000 movies plus even more on over 900,000 people who helped make them. WTG contains detailed information of many countries and cities. We extract representative structural information from HTML document trees of the two websites to test our algorithm.

4.1 Performance Analysis

We ran a query using condition (*title*=*love*)^(*from_year*=1930)^(*to_year*=2010) at IMDb to get 3000 movie titles, from which information was extracted to form our movie dataset. One of the extracted semi-structured movie objects is shown in Figure 1. We choose information extracted from 400 links at www.wtgonline.com/navigate/region/AtoZ.asp and duplicate the 400 transaction objects to 3000 transaction objects to form our travel dataset.

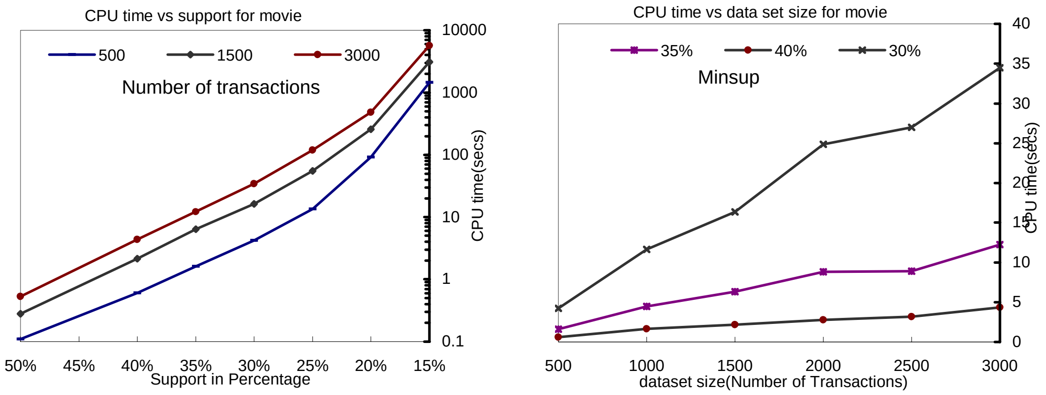
	Path-set	Tree-representation	Weaker than	Support
F_2'	{p1, p18}	{biography, *: {office}}	{1, 2}	2/3
	{p2, p18}	{biography: {birth}, *: {office}}	{1, 2}	2/3
	{p8, p17}	{name, *: {birth}}	{2, 3}	2/3
	{p8, p19}	{name, *: {interests}}	{2, 3}	2/3
	{p8, p20}	{name, *: {email}}	{2, 3}	2/3
	{p8, p21}	{name, *: {birth, interests }}	{2, 3}	2/3
	{p17, p18}	{*: {birth}, *: {office}}	{1, 2}	2/3
	{p17, p20}	{*: {birth}, *: {email}}	{2, 3}	2/3
	{p19, p20}	{*: {interests}, *: {email}}	{2, 3}	2/3
	{p20, p21}	{*: {email}, *: {birth, interests }}	{2, 3}	2/3
F_3'	{p8, p17, p20}	{name, *: {birth}, *: {email}}	{2, 3}	2/3
	{p8, p19, p20}	{name, *: {interests}, *: {email}}	{2, 3}	2/3
	{p8, p20, p21}	{name, *: {email}, *: {birth, interests }}	{2, 3}	2/3

Table 5: $F_2'$and F_3' in example 3.3

Our experiment environment is a 600MHz PC with 128M of memory. Both datasets can be run with a single partition. Figure 7 shows the execution time results on the movie dataset and Figure 8 shows the execution time results on the travel dataset. The y-axis in Figure 7(a) and 8(a) uses a logarithmic scale to show the running time. Two general trends can be observed: 1) As the minimum support decreases, the execution time increases in all cases; 2) the execution time is linear in the size of data set. Table 6 lists the number of frequent substructures and maximal frequent substructures discovered at different support levels from the movie and travel datasets with the size of 3000.

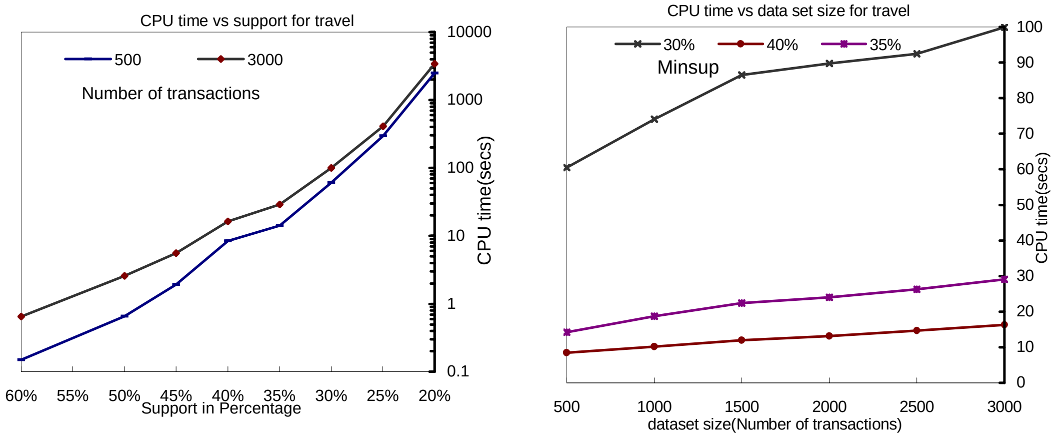
Compared with the approach in [24], our technique has the ability to find more frequent substructures. We also believe that our algorithm is more efficient than that in [24] because we avoid over-generation of paths with wildcard and using expensive tree matching in counting support. However, the execution time of our algorithm is not directly comparable with that of [24] as our technique is much more powerful (i.e., it is able to generate more interesting substructures than the approach in [24]). The analysis below gives an indication of the inefficiency of tree matching.

Assume that there are n transaction objects, each of which has m nodes. Consider a substructure (tree) s without wildcard has k nodes and l leaf nodes. To test whether s is weaker than the set of transaction objects, the time complexity of the tree match algorithm used in [24] is $O(km^{1.5}n)$. The time complexity of our approach is only $O(ln)$ in the worst case, where l is much smaller than k. It is not meaningful to directly compare the time complexity of computing the frequency of a substructure with wildcard between our approach and tree matching because we adopt a more powerful wildcard mechanism than [24]. However, when a substructure contains wildcards, the efficiency of tree matching becomes much worse, while our approach is not much affected as we use an efficient association mining technique and a special *tidlist* format for paths with wildcard.



(a) (b)

Figure 7: Computational performance on the movie data set



(a) (b)

Figure 8: Computational performance on the travel data set

Dataset	Movie				Travel			
Minsup	15%	30%	40%	50%	20%	30%	40%	50%
The number of discovered substructures	578,405	9,983	2,564	563	625,635	105,547	36,474	6,250
The number of discovered maximal substructure	7,666	377	115	44	9,767	3,685	1,598	547

Table 6: The number of substructures found

4.2 Example Substructures

We choose top 250 movies (http://us.imdb.com/ top_250_films) to make up our dataset. We set *Minsup* to 40%. The *Minsup* is usually set high for substructure discovery of semi-structured data as substructures with too small support is of no use, e.g., for database storage and query. Figure 9 shows three example interesting substructures discovered by our system. All of them are maximally frequent.

In substructure 1, there is no wildcard involved. If we had not used wildcard, we could only find such kind of substructures. We observe that none of the *director*, *writer*, *leading* *actor*, *producer* and *cinematographer* individually has enough support for the substructure *Spouse*: {*Name*, *Birth date* (*location*), *Trivia*}, which is discovered in substructure 2 containing wildcard.

In substructure 2, the wildcard * can represent any of the following label sequences: [*director*], [*writer*], [*Details*, *producer*], and [*Details*, *Cinematographer*], but not [*Cast*, *leading actor*]. The wildcard in [24] matches only one label. Therefore, [24] cannot discover frequent substructures from all person objects because *leading actor*, *producer* and *cinematographer* are at level 2, *director* and *writer* are at level 1. [24] cannot discover substructure 2.

In substructure 3, the two wildcards can cover *director*, *writer*, or *leading actor*, but not *producer* and *cinematographer*. Moreover, the two wildcards must represent two different label sequences in a movie object. Such substructures give us more possibility to fully explore the schema of semi-structured data. [24] cannot discover such substructures because its wildcard matches only one label and must represent all matched labels, but not its subset. It is computational prohibitive to compute the support of such substructures with a tree matching algorithm. In addition, two new attributes appear in substructure 3: "*salary*" and "*personal quotes*", which are not in substructure 1 and 2.

Substructure 1

{Title, Released year, Genre, Plot outline, Country, Language,
 Director: {Name, Birth date (location), Show more},
 Writer: {Name, Birth date (location)},
 Cast: {Leading actor: {Name, Birth date (location), Mini bio, Show more: {
 Mini bio, Spouse, Trivia}}},
 Details: {Producer: {Name}, Cinematographer: {Name}}}

Substructure 2

{Title, Released year, Genre, Plot outline, Country, Language,
 Cast: {Leading actor: {Name, Birth date (location), Show more: {Height, Spouse, Trivia}}},
 *: {Name, Birth Name, Birth date (location), Show more: {
 Mini bio, Spouse: {Name, Birth date (location), Trivia}, trivia}}}

Substructure 3

{Title, Released year, Genre, Plot outline, Country, Language,
 *: {Name, Birth Name, Mini bio, Show more: {
 Mini bio, Spouse: {Name, Birth date (location)}, Trivia, Personal quotes}},
 *: {Name, Birth date (location), Show more: {Mini bio, Trivia, Salary}}
 Details: {Producer: {Name}, Cinematographer: {Name}}}}

Figure 9: Some typical frequent substructures

5 Conclusions

As the amount of data available on-line grows rapidly, more and more hierarchical semi-structured data are becoming available. The motivation of the paper is the observation that semi-structured objects describing the same type of information are often similarly structured, but not identically structured. Discovering typical substructures that are shared by a large number of semi-structured objects is important for both data management and end users. We have defined the discovery problem and proposed a technique to solve the problem based on a new representation of semi-structured objects and a powerful wildcard mechanism. The effectiveness and efficiency of the proposed technique were evaluated on real-life datasets.

References

[1] S. Abiteboul. Querying semi-strucutured data. In Proc. of ICDT, 1997.

[2] S. Abiteboul, D. Quass, J. McHugh, J. Widom, and J. Wiener. The lorel query language for semi-strucutured data. In International Journal on Digital Libraries, 1(1):68-88, 1997

[3] S. Abiteboul, P. Buneman P and D. Suciu. Data on the web: from relations to semi-structured data and XML. Morgan Kaufmann Publishers, 2000.

[4] R. Agrawal, R. Srikant, Fast Algorithm for Mining Association rules. In Proc. of the 21th VLDB, 1994.

[5] R. C. Agarwal, C. C. Aggarwal, V. Prasad. Depth First Generation of Long Patterns. In Proc. of KDD 2000, Boston, USA.

[6] G. Arocena and A. Mendelzon. WebOQL: Restructuring Documents, Databases, and Webs. In Proc. of ICDE'98, Orlando, February 1998.

[7] R. J. Bayardo. Efficiently Mining Long Patterns from Databases. In Proc. of ACM SIGMOD, 1998.

[8] E. Bertino, G. Guerrini, I. Merlo, and M. Mesiti. An Approach to Classify Semi-Structured Objects. In Proc. 13th European Conference on Object-Oriented Programming, 1999

[9] D. Burdick, M. Calimlim, and J. E. Gehrke. MAFIA: A Maximal Frequent Itemset Algorithm for Transactional Databases. In Proc. of the 17th ICDE, Heidelberg, Germany, April 2001

[10] P. Buneman, S. Davison, M. Fernandez, and D. Suciu. Adding structure to unstructured data. In Proc. of ICDT, 1997.

[11] P. Buneman and B. Pierce. Union types for semi-structured data. In technical report MS-CIS-99-09, Dept. of CIS, university of Pennsylvania.

[12] Z. Chen, H.V. Jagadish, F. Korn, N. Koudas, R. Ng, S. Muthukrishnan, D. Srivastava. Counting Twigs in a Tree, In Proc. of ICDE 2001

[13] G. Cong, L. Yi, B. Liu, and K. Wang. Discovering Frequent Substructures from Hierarchical Semi-structured Data. Technical report, National Univ. of Singapore.

[14] A. Deutsch, M. Fernandez, and D. Suciu. Storing semi-structured data with STORED. In Proc. of ACM SIGMOD, Philadelphia, Pennyslyvania, 1999.

[15] R. Goldman and J. Wildom. DataGuides: Enabling query formulation and optimization in semistructured databases. Proc. of the 23rd VLDB, 1997.

[16] M. Holsheimer, M. Kersten, H. Mannila and H. Toivonen. A perspective on databases on data mining. In Proc of KDD, Aug. 1995.

[17] S. Nestorov, S. Abiteboul, and R.Motwani. Extracting schema from semistrucutred data. In Proc. of ACM SIGMOD, 1998.

[18] S. Nestorov, S. Ullman, J. Wiener, and S. Chawathe. Representative Objects: Concise Representation of Semistructured, Hierachical Data. In Proc. of the 13th ICDE, 1997.

[19] A. Savasere, E. Omiecinski, S. Navathe. An efficient algorithm for mining association rules in large databases. In Proc. of the 21th VLDB Conference, 1995.

[20] S. Sarawagi, S. Thomas, R. Agrawal. Intergrating association rule mining with relational database systems: alternatives and implications. In Proc. of ACM SIGMOD, 1998.

[21] J. Shanmugasundaram, K. Tufte, G. He, C. Zhang, D. DeWitt, and J. Naughton. Relational Databases for Querying XML Documents: Limitations and Opportunities. In Proc. of the 25th VLDB, Edinburgh, Scotland, 1999.

[22] L. Singh, B. Chen, R. Haight, P. Scheurmann. An Algorithm for Constrained Association Rule Mining in Semi-structured Data. In Proc. of PAKDD, 1999.

[23] H. Toivonen. On knowledge discovery in graph-structured data. In workshop on knowledge Discovery from Advanced Databases, 1999.

[24] K. Wang and H.Q. Liu. Discovering Structural Association of Semistructured Data. In IEEE Transactions on knowledge and data engineering, 12(3), pages 353-371, May/June, 2000.

[25] K. Wang and H.Q. Liu. Schema Discovery from Semistructured Data. In Proc of the 3st KDD, 1997, California, USA.

[26] J. Widom. Data Management for XML - Research Directions. IEEE Data Engineering Bulletin, Special Issue on XML, 22(3):44-52, September 1999.

[27] M. Zaki, S. Parthasarathy, M..Ogihara, and W. Li. New Algorithms for Fast Discovery of Association Rules. In Proc. of the 3st KDD, 1997, also in TR 651, CS Dept, Univ. of Rochester.