Discovering Associations in XML Data

Amnon Meisels, Michael Orlov and Tal Maor

Department of Computer Science
Ben-Gurion University of the Negev
Beer-Sheva 84105, Israel

am,orlovm,maort@cs.bgu.ac.il

Abstract—*Knowledge inference from semi-structured data can utilize frequent sub structures, in addition to frequency of data items. In fact, the working assumption of the present study is that frequent sub-trees of XML data represent sets of tags (objects) that are meaningfully associated. A method for extracting frequent sub-trees from XML data is presented. It uses thresholds on frequencies of paths and on the multiplicity of paths in the data. The frequent sub-trees are extracted and counted in a procedure that has $O(n^2)$ complexity.*

The data content of the extracted sub-trees, in the form of attribute values, is cast in tabular form. This enables a search for associations in the extracted data. Thus, the complete procedure uses structure and content to extract association rules from semi-structured data. A large industrial example is used to demonstrate the operation of the proposed method.

◆

1 Introduction

There are three main features, unique to XML files, that can be explored for inferring knowledge:

- The topology of the data — its graph structure
- The frequency of tags and of graph-connected tags
- The values of tags — attributes that are attached to tags

The main idea is that tags that are connected topologically, i.e. belong to some subtree of the XML data, are "meaningfully" related. For example, the tag "**staff-member**" appears frequently with a subtree that includes three branches with tags "**name**" , "**phone**" and "**office**". By discovering the fact that this subtree repeats itself in the data one can infer that it is an object with a meaning. Tags that are topologically connected can be investigated further in two combined directions. One direction looks at the meaning (i.e. natural-linguistic meaning) of the tags found to be related. The other direction is based on the statistical distribution of attributes value, within the set of related tags. Values of Tags typically take the form of attributes and it is common to have more than one attribute per tag.

Our proposed procedure for discovering sets of related tags, is composed of two parts. One part calculates frequencies of tags within paths on the XML data. It prunes its "paths-with-frequencies" by thresholding these frequencies relatively to their topological connections. It also achieves a useful pruning of paths by analyzing the distribution of frequencies of attributes values. As will be seen in Sec. 3, the extraction of attributes and values involves complex steps, sometimes using text strings as key words for comparison of values.

The product of the first part of our proposed procedure is a set of paths of tags that are frequent, their attribute values frequent, and topologically connected on a path. These paths can now be combined into subtrees and the frequency of these subtrees in the XML data can be counted. Our proposed interpretation is that frequent subtrees represent related objects (that contain some knowledge). The issue of counting of subtrees is computationally complex and has been attempted by a few researchers in the last two years. We believe that we improve on former proposals for locating frequent subtrees in XML data, by using the above three features together. Our proposed extraction and counting algorithm behaves like the square of the number of Tags. Former proposed algorithms [10] for counting subtrees, creating all potential frequent k-trees, were clearly exponential.

2 Related work

There are two recent papers that address the search for frequent objects in XML data [9], [10]. Wang and Liu used the same algorithm in both papers. In [10] they searched on HTML data that was constructed from a movies database (IMDB at `http://us.imdb.com`), by an SQL query - "Take the 5000 movies that were made from 1950 to 1998 in the US". The resulting tree is *assumed to be known*, so that [10] counts subtrees that start from a selected node in the tree of records. The present study assumes no knowledge about the structure of the XML data and looks for frequent subtrees that start anywhere on the data tree. Consequently, the definition of the term frequent is different in [9], [10] and in the present study. WL start from frequent paths with one leaf and construct all possible trees with two leaves, three leaves, etc. This constructive algorithm is obviously exponential in the number candidates and counts all possible embeddings of subtrees [10]. In the proposed algorithm of the present study, we construct subtrees bottom up, starting with tags that were found frequent and adding to them more frequent tags (see Sec. 3.3). As a result of this difference, our proposed algorithm is not exponential.

Another recent study [6], [7] builds on top of the frequent tree counting method of [10]. It looks for multi-level association rules, extracted from the frequent trees of [10]. Frequent subtrees have also been derived from HTML data, by [11]. Tags of HTML files are considered both as an attribute and as a value. For example, in the tag `<td>Position</td>`,

`<td>IT Specialist</td>`, the attribute name is *Position* and its value is *IT Specialist*. With this approach, [11] names construct trees of hierarchical classes, where leafs are text between two tags.

Starting from HTML data and removing structures that are assumed to be irrelevant (such as images or sound) Laur et. al. [5] also propose to count frequent subtrees. In [5], trees are constructed from paths (i.e. one leaf) and combined to form trees. The term frequent (similarly to [10]), is defined to be the number of appearances of a path that is higher than a given threshold. Each transaction contributes at most one count of the tree frequency. In a subsequent stage, nodes of frequent trees are organized and indexed according to their levels in the tree.

The most recent published method for find frequent sub-trees in semi- structured data operates on either XML or HTML data [3]. It finds a frequent pattern tree in the original data tree by starting with one frequent node and expanding it recursively. A pattern is defined to be frequent by [3] according to the count of its root instances. Support is defined as the ratio of this count to the number of nodes in the data and a threshold for support is used to extract frequent patterns.

The path of a "web-surf", that follows the list of sites a user moved through, can also be analyzed for frequent sub paths. Here, two somewhat different approaches appeared recently, [8], [4]. One approach generates a tree per user, where all paths generated by a single user over time are combined into a tree [4]. The other approach generates a tree for each instance of "web-surfing", thus creating a very large number of trees (paths) [8]. Chen et. al. [4] look for frequent instances of trees among users. Here, frequent is taken to mean that the number of instances is above a given threshold. Lin et. al. [8], on the other hand, define frequency and support on web paths. Each visit contributes one count to the support of a node in a given path. Frequent paths are combined to construct frequent trees. The method proposed in [8] combines two paths or trees if they differ by at most two nodes.

3 EXTRACTING FREQUENT TREES

The proposed extraction procedure for frequent subtrees starts with the extraction of frequent paths. All statistics use the tags of the XML data and some thresholds on the frequency of attribute values within tags. The paths are pruned and written to a (meta-data) XML file. Next, a pruning tree is constructed from the extracted paths. It includes all of the frequent paths, but its parts do not necessarily exist in their simple form in the data (see Sec. 3.2). The pruning tree is used to prune the data tree. Each frequent path is scanned top-down on the full data tree, to find all paths on the pruning tree that are connected to it. The result of this procedure is a pruned data tree. A tree that actually appears in its full form in the XML input data, which is composed only of frequent paths. The third and final stage is to count frequent subtrees on the pruned data. In this stage the frequencies of subtrees that are composed only of frequent paths and that appear as a whole in the real data are counted. This is done by a low complexity counting algorithm that uses the table of frequent trees created during the second stage. The three stages are described in detail in the following subsections.

3.1 Frequencies of paths

Statistics over tags, their attributes and attributes' values generate our definition of frequent paths as follows:

- For each tag on a distinct path (identified uniquely by the list of tags/nodes from root to the tag being counted), the count of its appearances is computed (for example, tag `book` appears 1530 times on path `/library/shelf/book`);
- The total count is calculated, for each attribute of each tag. For example, the book above could have attributes `@date` and `@ISBN`, the former with count of 1530, and the latter with count of 1310;
- The counts for each value of an attribute, are also calculated. For example, `@date` has the value `1984` forty times, and the value `1899` only a single time.

For the purpose of the statistics above, text nodes are considered as attribute nodes, with automatically generated names (such as `@text-node-1`).

Here is a sample of typical results of the path frequencies statistics (the actual result contains a variety of additional statistics, that are omitted here). Note that the number of occurances of the tag "SPN"; of the attribute "text-node-1" within this tag; and the number of occurances of the specific value "SEL FROM", of this attribute are all equal to 239. This particular (real) example reflects the fact that in an "industrial" XML database it frequently occurs that in all occurances of a specific tag there occurs also the same attribute and the same value for this attribute.

```
<?xml version="1.0" encoding="UTF-8"?>
<frequencies xmlns:kite="http://www.kite.org.il">
  <tag name="SFN" count="239">
    ...
    <path-per-tag name="SFN"
       path="/EIPC/FIGURE/PARTLIST_SECTION/PART/NOM_COL/SFN"
                 count="239">
      <attr-per-path name="text-node-1" count="239">
        <value-per-attr-per-path value="SEL FROM"
                                  count="239"/>
      </attr-per-path>
    </path-per-tag>
    ...
  </tag>
  ...
</frequencies>
```

3.2 Pruning tags and paths

For our final goal we want to combine the discovery of frequent XML subtrees and of interesting relations among the values of attributes of those frequent subtrees. To this end, we must determine in some way which tags and attributes are "interesting". This is best done by using both the topology of the XML data and the distribution of values within its tags. In the present discovery procedure we apply two types of criteria, one looks at the *content of the tag* and the other examines the relative frequency of the tag (i.e. relative to frequencies on its path). The criterion for the content of tags examines the *distribution of values of attributes* and retains values that are more frequent than some threshold. The criterion for frequency of tags uses a threshold on the *ratio of frequencies of tags* along paths. The criteria are applied in the form of a pruning procedure, by the algorithm that searches for frequent tags. The algorithm prunes the frequent tags by using the three following steps:

1. For each tag, the ratio of its count (determined in Sec. 3.1) to its parent tag count is calculated. If the ratio exceeds

a given threshold, the tag is not pruned. Tags that are not pruned are checked (in the following step) for "interesting" values to their attributes. Pruned tags are removed together with the rest of their paths.
Note that frequency ratios of over 100% signal multiplicity of descendants, on the average. If, on the average, for each path `A-B` there is a corresponding path `A-B-C`, it probably means that many `B` on the path `A-B` have a child `C`, and thus `C` is essential for the frequent subtrees counting procedure.

2. Attribute *values* that have a relative frequency (i.e. relative to their total count) exceeding a given threshold are considered "interesting". For example, for a threshold of 2%, the value `1984` in the example of Sec. 3.1 would be considered "interesting", and the value `1899` in the same example would not.
Attributes must have at least *two* "interesting" values in order to be considered "interesting". This is because if an attribute has only a single dominant value, it cannot yield associations (or knowledge). Tags that are not "interesting" are pruned by a recursive procedure that is described as part of the next step.
The reason for this step is that if an attribute has either many "insignificant" values or one value only, it will not generate any meaningful associations. Since we want "interesting" values for each attribute, if there are none, we prune the tag. The meaning of "insignificant" is defined by a suitable threshold.
3. Tags that are not "interesting" are pruned only if they do not posses an "interesting" descendent tag. In other words, tags remain in the data tree (and thus participate in the counting of subtrees) only if there is a descendant tag with "interesting" attribute(s) (or the tag has an interesting attribute itself).

The example in (Fig. 1) demonstrates the results of applying the pruning principles described above. In the figure of the data tree, boxed nodes are nodes that have "interesting" attributes (determined by a threshold on the distribution of their values).

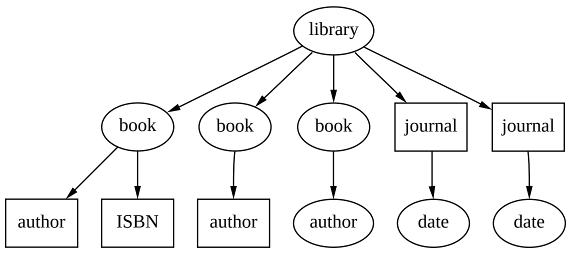


Fig. 1. Visualizing determination of "interesting" tags. Boxed nodes are those that have "interesting" attributes.

Let us assume that the threshold which is used in Step 1 above is 40%. Following the procedure of determining interesting tags, we find that the path `/library/book/ISBN` is below the 40% threshold (this path occurs only once, while its ancestor `/library/book` occurs three times). Next, `/library/journal/date` has no "interesting" attributes (it's not boxed in the figure), and it also has no descendants, therefore, this path, is also "non-interesting". It can be seen that for the other paths, all the requirements hold, and thus the paths that remain are:

- `/library`
- `/library/book`
- `/library/book/author`
- `/library/journal`

From all the paths that were kept as part of frequent tags, by the procedure in Sec. 3.1, a tree is constructed. All "non-interesting" tags and attributes are not included in the tree. In other words, tags that were not found to be "interesting" by the recursive procedure of Sec. 3.2 are pruned together with subtrees that are rooted at those tags. We term the result, the *pruning tree*. The pruning tree for the data tree in Sec. 3.2 is shown in Fig. 2. The algorithm for building the *pruning tree* is Alg. 1.

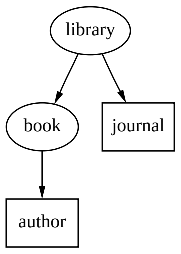


Fig. 2. Pruning tree resulting from uniting the "interesting paths" from the data tree in Sec. 3.2.

Algorithm 1 BUILDPRUNINGTREE(`PATHLIST`)

```
Pre-condition: PathList is a list of all different paths in the data tree
Post-condition: $T$ is the pruning tree that adheres to the principles described in this section
1:  /* Next step before the beginning of the recursion */
2:  $T \leftarrow$ Tree built by uniting the paths in PathList
3:  if $T_{\text{count}}/T^{\text{parent}}_{\text{count}} >$ given threshold then
4:     /* Erase $T$ from its parent's list of child subtrees */
5:  else
6:     for all $T' \in$ child subtrees of $T$ do
7:        BUILDPRUNINGTREE($T'$)
8:        for all $A \in$ attributes of $T$ do
9:           if $A$ doesn't have at least two "interesting" values then
10:             /* Erase $A$ from the list of attributes of $T$ */
11:       if $T$ has no child subtrees $\wedge$ $T$ has no attributes then
12:          /* Erase $T$ from parent's list of child subtrees */
```

The *pruning tree* is a combination of frequent paths which satisfy some requirements. In order to be able to count real subtrees, that actually appear in the data we have to use the original XML tree, which we will call the *data tree*. The *pruning tree* is utilized for the purpose it was built for — pruning nodes in the *data tree*. Alg. 2 presents the way to do it: traverse the *data tree* simultaneously with the *pruning tree*, and throw from the *data tree* all the tags and their subtrees that do not appear in the *pruning tree*. The algorithm needs one pass from top to bottom on the *data tree*, to produce the *pruned data tree*. At the end of this step, all the nodes of the *pruned data tree* contain tags which are "frequent". This means that some of these nodes have "interesting" attributes and/or values, or one of their descendant nodes does.

Algorithm 2 PRUNE-DATA-TREE(DATATREE, PRUNINGTREE)

Pre-condition: `dataTree` is the original *data tree*, `pruningTree` is the *constructed from all frequent paths*.

Post-condition: Pruned data tree is returned. In this tree, each node contains a representation of the subtree rooted at this node, as well as the representation of the values of nodes on that subtree. We use such representations so that order of children in the subtree won't matter, and for the purpose of saving space as well.

```
1: processedTrees ← ∅
2: dataSubTrees ← dataTree.children()
3: for all $c \in$ dataSubTrees do
4:    /* we only consider Element nodes of the XML data tree, ignoring Comment and Text nodes */
5:    if $c$ is an Element node $\wedge$ $c$.node() $\in$ pruningTree.children() then
6:       processedTrees ← processedTrees ∪ {PRUNE-DATA-TREE($c$,$c'$)}
7: return New subtree with corresponding representation and values
```

3.3 Finding frequent subtrees

The counting of subtrees is a complex process that performs two tasks at the same time - counting of subtrees and storing of all appearances of the counted subtrees. The different appearances of the same sub- tree relates to the existence of different values of attributes in the same tags of the two sub-trees. The counting algorithm keeps two global tables that store subtrees in tabular (one row per sub-tree) form. For efficiency, these tables are hash-tables that store and retrieve similar subtrees in $O(1)$. In our implementation, these are standard Java HashMaps, using the standard Java hashCode() for string representation. The first table stores every possible subtree in the *pruned data tree*. It has one row per each different subtree in the pruned data-tree. The second table stores all the appearances of the data of each subtree in the *pruned data tree*. Each subtree that has a count of n in the data, will have n different rows in the second table. This will enable the retrieval of the actual attribute values for algorithms that look for meaningful associations in the frequent sub-trees.

The counting algorithm traverses the pruned data-tree from top to bottom, recursively. Each visited node (tag) causes the algorithm to insert the relevant structure of the sub-tree into the two tables. For example, if the string `{A{B}{C{D}}}` is inserted into the first table, this represents the traversal of a sub-tree in which `B` and `C` are children of `A`, and `D` is a child of `C`. The entry is inserted when the algorithm has traversed this sub-tree and is pointing at node `A` for the second time. If the subtree already exists in the counting table, the counter will be updated. At this point in the run of the algorithm, the subtree's attributes and their values are inserted into the second table. For example, `{id="123"{attribute1="b1",attribute2="b2"}{{no="5"}}}`.

In the description of Alg. 3, the term `SubtreesRegistry` refers to the two global tables described above. One for counting subtrees and the other for lists of subtrees "value-tuples". In the above example, when a subtree is represented as `{A{B}{C{D}}}`, the representation is unique for all topologically equivalent subtrees, the order of children is not important. Assume that in the first table this subtree has the count of 3. In the second table it might have the following list of values:

```
{id="123"{attribute1="b1",attribute2="b2"}{{no="5"}}}
{id="456"{attribute1="b1",attribute2="b3"}{{no=""}}}
{...}
```

In the first list entry, `C` has no value or attribute, while `B` has two attributes. For completeness, a missing attribute value at some tag is stored as the value `none`. Here, the term "missing" relates to attribute values that appear at some of the appearances of a frequent sub-tree and are "missing" in others. This will be useful later, when all attribute values are transferred to some data mining algorithm for extraction of interesting associations and empty fields are problematic.

The worst case complexity of this algorithm is $O(n^2)$ and this happens when the data tree becomes one path. The algorithm scans the data tree bottom-up. When the algorithm starts it picks up a leaf and updates the SubtreesRegistry tables. Its next step is to combine the leaf with the immediate subtree it belongs to. The algorithm updates the SubtreesRegistry accordingly and repeats this step until it reaches the root of the data tree. The worst complexity of each step is $O(n)$, writing the whole data. The number of steps is the depth of the tree which is n at the worst case.

Algorithm 3 COUNT-SUBTREES(PRUNEDDATATREE, SUBTREESREGISTRY)

Pre-condition: `PrunedDataTree` is the pruned data tree (data with eliminated "non-interesting" nodes, and `SubtreesRegistry` is structure with two tables as described before.

Post-condition: The two tables in `SubtreesRegistry`, one with subtrees counts, and another one with subtrees values list, are updated for all subtrees in `PrunedDataTree`.

```
1: PrunedDataSubTrees ← PrunedDataTree.children()
2: for all $c \in$ PrunedDataSubTrees do
3:    COUNT-SUBTREES($c$,SUBTREESREGISTRY)
4: /* rep is this subtree representation, and valueRep is the corresponding subtree values representation */
5: SubtreesRegistry.addCount(rep,1)
6: SubtreesRegistry.addValue(rep,valueRep)
```

4 IMPLEMENTATION AND EXPERIMENTATION

To test the proposed algorithms and their implementation we ran some experiments on XML data. The standard XML datasets are no good for our approach. Take the famous IMDB for example, it's data is composed of a few original large tables translated into XML format. This means that *all subtrees are similarly frequent*. In other words, all subtrees of tags correspond to tuples in an original database table. Correspondingly they all have the exact same frequency for all sub-tuples, hence no “interesting” subtrees (with widely different frequency of appearance in the data).

Our experiments were run on a large industrial XML data file that stores a large manual of an aviation company. This particular dataset, generated probably mechanically, has little hope for extraction of meaningful associations, but is at least a true XML data tree with subtrees widely differing in their frequency of appearance. The input data includes a total of 382,373 instances of 62 different tags, in a tree of depth 8. The parameters of the XML data tree are given in table I.

Depth	8	7	6	5
Num. of Tags	2873	391 25	237, 523	79, 354

Depth	4	3	2
Num. of Tags	21, 287	1983	221

TABLE I
DISTRIBUTION OF TAGS IN THE INPUT XML DATA TREE

The result of running Algorithms 3 for 4 different cases of threshold selections are presented in Table II. For all the cases in Table II, the similarity threshold for identifying the discovered frequent subtrees is 90% and the required minimal number of trees is 5. The table presents the number of subtrees, with different topologies, that were found to be frequent in the data.

case	*attribute threshold*	*tags threshold*	*different subtrees*
1	20%	20%	7
2	20%	5%	12
3	45%	20%	1
4	20%	45%	7

TABLE II
RESULTS OF RUNNING ALGORITHMS 1, 2, 3

The exact frequency distribution of the resulting subtrees (that were found to be frequent) is detailed in Table III. Entries in Table III give the number of instances of frequent subtrees, with a given number of Tags. Tags are counted in leafs of the subtree.

case	*1 tag*	*2 tags*	*3 tags*	*4 tags*
1	19117	10549	8368	175
2	995	9814	8126	242
3	564			
4	19117	10549	8368	175

case	*5 tags*	*8 tags*	*9 tags*	*16 tags*
1			5	
2	6	5		9
3				
4			5	

TABLE III
DISTRIBUTION OF FREQUENT SUBTREES

To illustrate the process, let us take a look at a particular subtree that was found to be frequent (Figure 3). The subtree in Figure 3 has 4 tags/leaf-nodes and 175 instances of this subtree were found in the data. Taking the multiple instances of the data (i.e. the values of its attributes), one can attempt to discover meaningful associations. Analyzing the 175 data tuples of the frequent tree of Figure 3 for association rules and using a support threshold of 20% and a confidence threshold of 90% we get 29 association rules. Examples of association rules are:

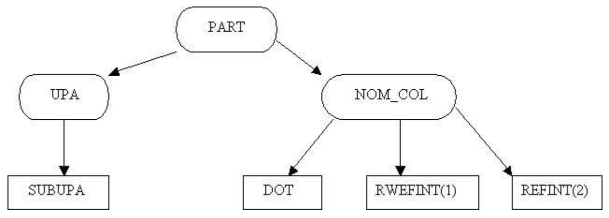


Fig. 3. Example1 description.

```
1. If     PART-->UPA-->SUBUPA=1
   Then  PART-->NOM\_COL-->DOT=.
       support $32.39\%$     confidence $95\%$
2. If     PART-->NOM\_COL-->DOT=.
   Then  PART-->NOM\_COL-->REFINT(2)=null
       support $81.25\%$     confidence $91.67\%$
3.  If     PART-->NOM\_COL-->REFINT(1)=null
    Then  PART-->NOM\_COL-->DOT=.
     &     PART-->NOM\_COL-->REFINT(2)=null
       support $81.25\%$     confidence $94.08\%$
```

The analyzed XML data describes presentation modes of text and graphics of a large manual for some sub systems of an airplane. The attribute DOT describes modes of presentation of parts, whether they are composed of several sub-presentations, that can be zoomed-in, or not. The value “.” means that the presentation has no zoom-in capabilities. The first rule means that when only one subpart is present, then the presentation does not include capabilities for presenting sub-parts. The second rule means that when the presentation is “simple”, then no pointer to a second sub-part exists. The third rule is similar to the first and second one, but, connects the fact that there is no sub-part with the “simple” mode of presentation and the non existence of a second pointer, to a second sub-part.

A much more frequent subtree that was found, has 8126 instances in the data has 3 tags (leaf-nodes) and is presented in Figure 4. Using a threshold support of 20% and confidence of 90% we get from the 8126 instances of 3-tuples of data 8 rules. As can be seen, the 2 example rules are very similar to the previous 3 rules.

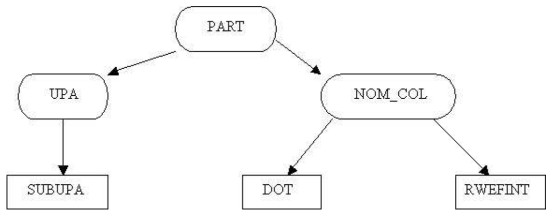


Fig. 4. Example2 description.

```
1. If     PART-->UPA-->SUBUPA=1
   Then  PART-->NOM\_COL-->REFINT=null
       support $23.38\%$     confidence $93.09\%$
2. If     PART-->NOM\_COL-->DOT=.
   Then  PART-->NOM\_COL-->REFINT=null
       support $82.66\%$     confidence $96.25\%$
```

5 Conclusions

The top-down scheme of our frequent trees extraction method and its resulting value-tuples for mining associations can be listed as follows:

1. Perform counts of all tags and their paths on the XML file and create a new XML file that includes the frequencies of paths and tags. This generates useful *meta-data*.
2. Prune "non-interesting" attribute values. Attributes that are either "rare" or "too-frequent".
3. Use thresholds on relative frequencies to prune tags/paths. This will prune subtrees that have low multiplicity (less frequent instances).
4. Combine frequent paths into subtrees, generating a "pruning tree" from the data.
5. Prune the data tree, retaining only subtrees that are in the pruning tree.
6. Count subtrees in the pruned data-tree, using an $O(n^2)$ algorithm.
7. Construct tables of values of the frequent objects (tuples), thereby enabling procedures of data mining (on the attribute values) to be performed.

The above scheme for extracting and counting frequent subtrees in XML data differs from former studies in two aspects:

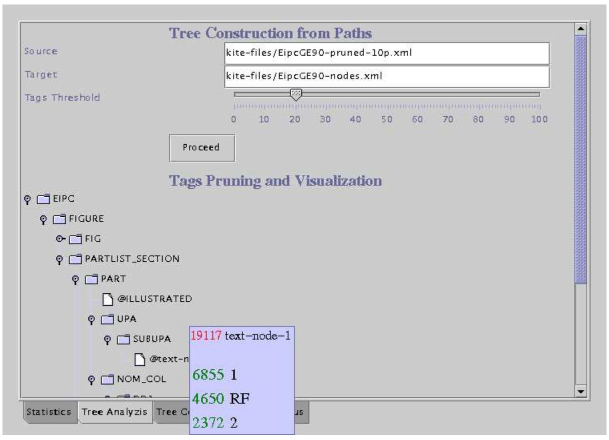
1. The extraction and counting procedure has low computational complexity.
2. The selection criteria for "interesting" sub-trees involves considerations that relate to the statistics of the *content* of the data in the XML tags.

For the counting of sub-trees, a viable comparison of the proposed method is with that of WL [10]. WL construct and keep entire paths and the subtrees that are constructed from the frequent paths. This procedure is clearly exponential in the number of data nodes. In the proposed method, subtrees are constructed from the bottom up. Two paths are combined if their leaves are brothers in the XML tree. As explained in section 3.2, this computation is linear in the input XML data.

The counting of sub-trees in the proposed method also differs from that of WL. Here, the number of appearances of combined paths in the XML tree, is calculated and its frequency determined. If it is frequent, it is inserted into the table of frequent trees. This procedure is performed recursively until a maximal sub tree is achieved. The algorithm of WL removes sub trees that can in principle be just partial trees to other (frequent) trees or frequent trees with a different order of some branches [10].

The second major difference of the present extraction method from former studies, is its consideration of discovery [2]. An important part of the pruning procedure relates to the *content* of the XML tags. Using a threshold on the distribution of *values of attributes*, is clearly a non structural criterion. Its rational is based on our assumption that the finding of frequent sub-trees is part of a wider goal - discovering forms of knowledge in semi structured data. Our specific choice is to attempt to discover association rules (cf. [1], [2]) in the extracted data that was found to be frequent (i.e. frequent sub-trees). The proposed method assumes that the values of attributes are the content of the XML data. This is the reason for pruning paths and potential sub-trees that have a value distribution of attributes that will not yield any meaningful associations.

The method proposed in the present paper has been implemented in Java and has an interactive part that enables the user to select the relevant thresholds. It also outputs the resulting pruning trees and frequent sub-trees, for the user to inspect and interact with. The graphical user interface of the implemented system for extracting frequent tags and constructing the pruning tree is shown in Fig. 5. As can be seen, the frequencies of attribute values are presented on the screen. The pruning tree appears on the screen as a combined set of frequent paths that resulted from the thresholds selected by the user (in the top part of the form). This pruning tree will be combined with the data, in order to produce real counts of subtrees in the data, as described in Sec. 3.3.


Tree Construction from Paths

Source: kite-files/EipcGE90-pruned-10p.xml

Target: kite-files/EipcGE90-nodes.xml

Tags Threshold

Tags Pruning and Visualization

```
EIPC
  FIGURE
    FIG
    PARTLIST_SECTION
      PART
        @ILLUSTRATED
        UPA
          SUBUPA
            @text-n
        NOM_COL
```

19117 text-node-1

6855 1

4650 RF

2372 2

Fig. 5. Graphical User Interface showing a pruning tree.

References

[1] Rakesh Agrawal and Tomasz Imielinski and Arun N.Swami, *Mining Association Rules between Sets of Items in Large Databases*. Proceedings of the 1993 ACM SIGMOD International Conference on Management of Data, pp.207–216, 1993

[2] Rakesh Agrawal and Ramakrishnan Srikant, *Fast Algorithms for Mining Association Rules*. Proc. 20th Int. Conf. Very Large Data Bases, VLDB, pp.487–499, 1994

[3] Tatsuya Asai et. al., *Efficient Substructure Discovery from Large Semi-structured Data, Proc. 2nd SIAM Intern. Conf. on Data Mining (SDM'02)*, pp. 158-174, Arlington, VA., April 2002.

[4] Ming-Syan Chen and Jong Soo Park and Philip S. Yu, *Efficient Data Mining for Path Traversal Patterns*, *Knowl. Data Eng.*, vol 10, pp. 209-221, 1998.

[5] P.A. Laur, F. Masseglia, P. Poncelet and M. Teisseire. *A General Architecture for Finding Structural Regularities on the Web (PS)*, *Proc. 9th Intern. Conf. on Artif. Intell.* (AIMSA'2000), Varna, Bulgaria, September 2000.

[6] K. Maruyama and K. Uehara *Mining Association Rules from Semi-Structured Data. ICDCS Workshop on Knowledge Discovery and Data Mining in the World-Wide Web 2000: F23-F30*.

[7] K. Maruyama and K. Uehara *Knowledge Integration of Rule Mining and Schema Discovering*, *Proc. Third Intern. Conf. on Discovery Science (DS2000)*, pp.285-289, 2000.

[8] X. Lin, C. Liu, Y. Zhang, and X. Zhou, *Efficiently Computing Frequent Tree-Like Topology Patterns in a Web Environment* , *31st Tool's Asia*, IEEE cs press, pp. 440-447, 1999.

[9] Ke Wang and Huiqing Liu. *Discovering typical structures of documents: a road map approach* . *Proc. 21st ACM SIGIR Conf. on Res. Dev. in Inform. Retrieval* (SIGIR'98), pp. 146-154, Melbourne, Austrailia, August 1998.

[10] K. Wang and H. Liu. *Discovering Structural Association of Semistructured Data* . *IEEE Trans. Knowl. Data Engin.*, vol. 12, pp. 353-371, 2000.

[11] Wai-ching Wong and Ada Wai-Chee Fu, *Finding Structure and Characteristics of Web Documents for Classification*. ACM SIGMOD Workshop on Research Issues in Data Mining and Knowledge Discovery, pp. 96-105, 2000.