Control and Cybernetics

vol. **40** (2011) No. 3

XCleaner: A New Method for Clustering XML Documents by Structure

by

D. Brzeziński, A. Leśniewska, T. Morzy, and M. Piernik

Abstract: With the vastly growing data resources on the Internet, XML is one of the most important standards for document management. Not only does it provide enhancements to document exchange and storage, but it is also helpful in a variety of information retrieval tasks. Document clustering is one of the most interesting research areas that utilize XML's semi-structural nature. In this paper, we put forward a new XML clustering algorithm that relies solely on document structure. We propose the use of maximal frequent subtrees and an operator called Satisfy/Violate to divide documents into groups. The algorithm is experimentally evaluated on real and synthetic data sets with promising results.

1. Introduction

XML is a standard for developing numerous web applications that deal with document retrieval and storage. It is also widely used for annotating Web resources like articles, movies, and web services. An XML document consists of a structure and content, and in this way differs from traditional data. This semi-structural nature of XML allows to model a wide variety of databases as XML documents. There is a number of papers (see Widom, 1999; Florescu and Kossmann, 1999; Chawathe, 1999) that have raised the issue of processing, storing, and management of documents in this format. However, in the field of XML processing, data mining is still a new topic. Clustering represents one of the most interesting trends in this research area (see Lian et al., 2004; Lesniewska, 2009; Costa et al., 2004; Tran et al., 2008).

Clustering aims at grouping together similar objects, in this case documents, usually according to a similarity measure. Because text document clustering algorithms ignore the structural information in the data, they are considered to be inappropriate for the purposes of grouping XML documents (see Dalamagas et al., 2004). Thus, there is a need to develop new clustering algorithms specifically for XML documents.

XML clustering methods can be categorized into three main groups: methods based on content, methods based on structure, and methods that analyze both content and structure. In this article we are focusing on the structural similarity of documents. This approach has many real-world applications in a wide variety

of domains. Identification of documents with similar structure can prove useful for systems that extract DTDs and XSDs from XML data sets. Structure-based clustering can also help to solve the problem of recognition of different sources that provide the same kind of information. Alternatively, it can be used in the structural analysis of a Web site. Moreover, since the XML language can encode hierarchical data, structure-based methods can be exploited in the discovery of structurally similar macromolecular tree patterns, encoded as XML documents, in bioinformatics.

The main contribution of this paper is a new methodology for clustering XML documents by structure. Our approach is based on a divisive algorithm that uses maximal frequent subtrees to split documents into groups. To perform the clustering, we introduce a subtree matching algorithm and a new operator called *Satisfy/Violate*. The proposed methodology is tested against heterogeneous and homogeneous sets of data. The preliminary results of the research indicate that our algorithm can provide high quality clustering for both types of data sets.

The structure of this paper is as follows: Section 2 discusses relevant work on the issue of clustering XML documents by structure; Section 3 contains the description of our method with examples; Section 4 presents the experimental environment and achieved results; Section 5 contains conclusions and shows directions of further research.

2. Related Work

One of the earliest approaches to clustering XML documents based on their structure is the *XClust* algorithm (see Lee et al., 2002), where authors propose to cluster documents by using Document Type Definitions (DTDs). This approach uses a similarity measure based on the semantic and structural information of elements in the documents' corresponding DTDs. However, this approach is not suitable for environments where documents do not have DTDs.

Most of the existing representations of XML documents are based on a labeled tree (see Candillier et al., 2005; Nayak and Iryadi, 2006) as it is a natural model for XML's hierarchical structure. An XML document can be easily transformed into a rooted labeled tree, but solutions based on tree-like structures require complex computations. Lian et al. showed (see Lian et al., 2004) that calculating the similarity between two documents represented as rooted ordered labeled trees, using the tree edit distance, is not efficient. Therefore, they proposed a graph summarization of XML documents called *s-graph*. To capture structural similarities, the distance between two s-graphs is computed according to the number of common element-subelement relationships. A similar solution, based also on a tree structure, was proposed by Dalamagas et al. (see Dalamagas et al., 2004). This approach introduced different structural summaries and improved computational efficiency without compromising cluster quality.

A different approach was presented in the *XProj* algorithm (see Aggarwal et al., 2007). The authors propose to use rooted ordered labeled trees to represent XML documents. The XProj algorithm uses a set of frequent substructures (sequences of tree edges) as the representatives of document clusters. The clustering algorithm is a partition based algorithm, which constructs groups that maximize the structural similarities among the documents within a group.

Our solution is based on an assumption, similar to that in XProj, that frequent substructures preserve more information than simple element-subelement relationships (see Dalamagas et al., 2004) or nodes and edges (see Lian et al., 2004). In contrast to XProj, we propose a hierarchical method which uses maximal frequent subtrees as patterns that represent clusters. Our solution is also unique in the use of an operator called *Satisfy/Violate* that determines the proper cluster for a document.

3. The XCleaner Methodology

Our approach is based on a real-world observation that similar objects can be characterized by a set of unique features. It is unnecessary to compare all objects in a set if we know the characteristic features of groups we want to create. For example, for a set of shapes it is unnecessary to compare all figures if we know it contains only triangles and rectangles. The easiest solution is to check whether a shape is a rectangle or triangle and assign it to the proper cluster based on the answer. This approach, called *clustering by patterns*, is the basis of our algorithm. We extract characteristic patterns from the input data set and use them to group objects by simple comparison.

In the following subsections we present the main components of our approach. In Section 3.1 we describe the *document representation* that is used in our algorithm. Then, in Section 3.2, we define the *patterns* that will represent clusters and the algorithm to obtain them. In Sections 3.3 and 3.4 we present the main components of the XCleaner algorithm: the *subtree matching algorithm* and the *Satisfy/Violate* operator. Finally, in Section 3.5, we present the XCleaner algorithm, which uses the *Satisfy/Violate* operator to perform XML document clustering by patterns. Additionally, Section 3.6 discusses the computational complexity of the presented approach.

3.1. Document Representation

In our work we use a tree-based document representation proposed by Zaki in (see Zaki, 2002). This representation relies on mapping the set of all XML tags in the document database into integers and then coding each document tree as a string. The string representation $\mathcal{S}_{\mathcal{T}}$ of document tree $\mathcal{T}$ is constructed as follows: Add vertex labels to $\mathcal{S}_{\mathcal{T}}$ in a depth-first preorder traversal of $\mathcal{T}$ and add symbol -1 whenever backtracking from a child to its parent. For example,

the tree in Fig. 1 would be coded as 0 1 3 1 -1 -1 -1 2 2 -1 3 -1 -1. The numbers in brackets next to the tree nodes in Fig. 1 show the depth-first traversal order.

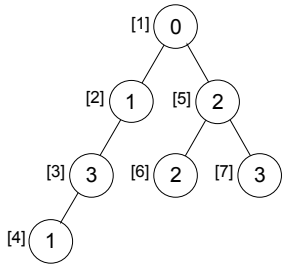

Figure 1. Tree representation of an XML document.

3.2. Pattern Mining

One of the most important components of our approach is the pattern mining algorithm. To describe it we have to present some necessary definitions first. A rooted, ordered, labeled tree $\mathcal{T}'$ is a *subtree* of a tree $\mathcal{T}$ if there exists a one-to-one mapping between each element and edge of $\mathcal{T}'$ and $\mathcal{T}$. A subtree $\mathcal{F}$ is called *frequent* in the set of document trees at a user defined minimum support level *minsup*, if it is a subtree of *minsup* percent or more document trees. A frequent subtree $\mathcal{M}$ is called *maximal* if there does not exist any other frequent subtree $\mathcal{T}'$ for which $\mathcal{M}$ would be a subtree. In this paper we will often refer to maximal frequent subtrees as *patterns*. To find patterns in a data set of XML documents we use the *CMTreeMiner* proposed by Chi et al. (see Chi et al., 2005).

Not all frequent subtrees provide valuable information for clustering. Patterns that occur in all documents do not allow to differentiate documents from each other. Therefore, we use an additional parameter *maxsup* that defines the maximal percentage of documents that a pattern can occur in. By decreasing the value of this parameter we reduce the maximal number of documents that can be assigned to one cluster.

3.3. Subtree Matching Algorithm

The role of the subtree matching algorithm is to determine, in an efficient way, whether a document contains a given pattern or not. The algorithm finds all the occurrences of the pattern root in the document by performing a depth-first traversal and tries to expand the pattern from each of those points. The algorithm terminates when the first occurrence of the pattern is found or when it reaches the end of the document. The exact steps are given in Algorithm 1.

Algorithm 1 Subtree Matching Algorithm

Input: d - an XML document in string format; p - a pattern in string format
Output: *true* if p is a subtree of d, *false* otherwise

```
1:  docPos ← 0;
2:  startPos ← null
3:  calculate depth level for each label p_i ∈ p and remove all return marks −1
    from p;
4:  docLevel ← 0;
5:  if startPos ≠ null then
6:    docPos ← startPos;
7:  if startPos > last occurrence of p[0] in d then
8:    return false;
9:  startPos ← null;
10: for i = {0, ..., legth(p) − 1} do
11:   while docPos < docLength and (p[i] ≠ d[docPos] or docLevel ≠
      depth(p[i])) do
12:     if i > 0 then
13:       if d[docPos] ≠ −1 then
14:         docLevel ← docLevel + 1. ;
15:       else
16:         docLevel ← docLevel − 1;
17:         if IsLevelCrossed(p, docLevel, i) then
18:           i ← index of the parent of p[i];
19:       if d[docPos] = p[0] and startPos = null then
20:         startPos ← docPos;
21:     docPos ← docPos + 1;
22:   if d[docPos] = p[0] and startPos = null then
23:     startPos ← docPos;
24:   if docPos > docLength then
25:     goto 5;
26:   else if i = legth(p) − 1 then
27:     return true;
28:   else
29:     docLevel ← docLevel + 1;
30:     docPos ← docPos + 1;
```

We will illustrate the algorithm with an example depicted in Fig. 2. Let us define the symbols we will use:

- *docPos*: indicates the current position in the document's string representation,
- *docLevel*: indicates the depth of the current node relative to the pattern root found in the document,
- *startPos*: stores the next root occurrence if the algorithm does not manage to find the pattern match in a single pass.

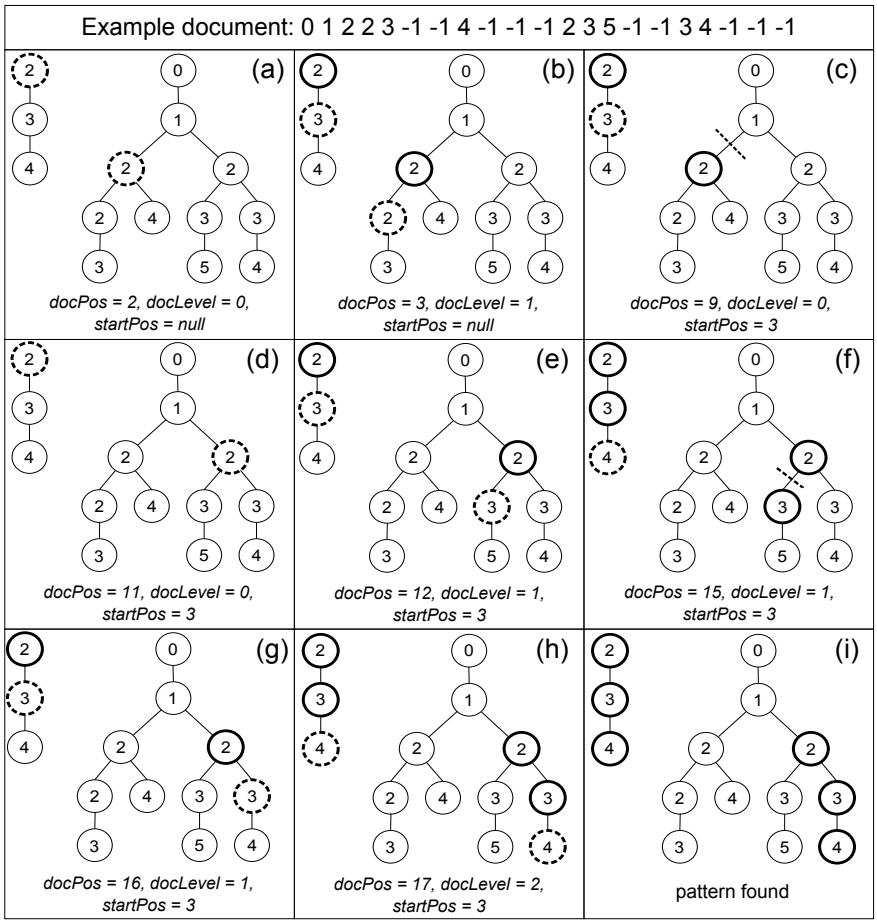

Figure 2. Subtree matching example.

Let d be an XML document and p be a pattern. We begin with transforming p from its string format (2 3 4 -1 -1) to a label[level] format (2[1] 3[2] 4[3]). Next, we seek for the first occurrence of the pattern root (2) (Fig. 2(a)). Once we have

found it, we continue traversing the tree in search for the next pattern element (3). While searching for the second pattern element we find an element with a label identical to that of the patterns root, so we remember its position as *startPos* (Fig. 2(b)). Continuing with the depth-first traversal of the document we find the second pattern element (label 3 at position 4), but the *docLevel* of this element is 3, while we are looking for the element at level 2. Thus, we have to continue searching for the second pattern element, which finally leads us a level higher than the found pattern root (Fig. 2(c)). This forces us to search once again for the pattern root. In Fig. 2(d-f) we see another attempt at matching p to d this time from $docPos = 11$. While searching for the third pattern element we reach the level of its parent and have to search for it again (Fig. 2(f)). Fig. 2(g-i) shows the last three steps in matching pattern p to document d.

3.4. Satisfy/Violate Operator

The *Satisfy/Violate* operator divides a set into two subsets according to a user-specified rule. More formally, for a set of objects $\mathcal{D}$ and a rule r the *Satisfy/Violate* operator divides $\mathcal{D}$ into subset $\mathcal{S}$ containing objects which satisfy r and subset $\mathcal{V}$ containing objects which do not satisfy r.

The idea standing behind the *Satisfy/Violate* operator is much broader than just our application. It can be used for example in divisive clustering or decision tree construction algorithms as a split operator. The rule behind the operator can be anything from a simple value comparison to a decision rule. In our example, a set of XML documents serves as an input set and the patterns serve as a set of rules. Additionally, we add a sensitivity parameter α which indicates the percent of patterns that a single document has to contain to be assigned to the satisfy set. Consequently, $(1 - \alpha)$ is the percent of patterns that a single document cannot contain to be assigned to the violate set. The process of applying the operator in our application is outlined in Algorithm 2.

3.5. The XCleaner Algorithm

Given the above definitions we can now describe the XCleaner methodology. Algorithm 3 outlines the procedure of clustering XML documents by using the *Satisfy/Violate* operator with maximal frequent subtrees.

The algorithm begins with searching the document data set for patterns with support greater than or equal to $minsup$. Next, it clusters the patterns into k groups with the *AHC* algorithm (see Johnson, 1967; Jain et al., 1999), where k is a user defined number of clusters. Although, there are several statistical methods for determining the number of clusters automatically (see Milligan and Cooper, 1985), to simplify the algorithm description we decided that k is given a priori. To perform this step, the algorithm needs to compute the similarity matrix for all pairs of patterns. This is done by applying the *Satisfy/Violate* op-

Algorithm 2 Satisfy/Violate operator for a set of patterns and XML documents

Input: $\mathcal{D}$ - set of documents; $\mathcal{P}$ - set of rules/patterns; α - sensitivity factor
Output: $\mathcal{S}$ - Satisfy set; $\mathcal{V}$ - Violate set

```
1: for all documents d ∈ D do
2:   matchCount ← 0;
3:   for all patterns p ∈ P do
4:     if Matches(d, p) then
5:       matchCount ← matchCount + 1;
6:   if matchCount ≥ α · count(P) then
7:     S ← S ∪ {d};
8:   else
9:     V ← V ∪ {d};
```

Algorithm 3 The XCleaner Algorithm

Input: $\mathcal{D}$ - set of documents to cluster; k - number of clusters; $minsup$ - minimal support for patterns; $maxsup$ - maximal support for patterns; α - sensitivity of the Satisfy/Violate operator
Output: $\mathcal{C}$ - a set of XML document clusters

```
1: P ← TreeMiner(D, minsup, maxsup);
2: P_groups ← AHC(P, k);
3: C ← k empty clusters each defined by one pattern cluster p_c ∈ P_groups;
4: candidateSet ← D;
5: for all c ∈ C do
6:   c ← Satisfy/Violate(candidateSet, p_c, α).S;
7:   candidateSet ← Satisfy/Violate(candidateSet, p_c, α).V;
```

erator on the whole document set $\mathcal{D}$ for each pattern and then using the number of common documents as the similarity between patterns. After preliminary experiments we decided to use *complete-linkage* to calculate the distance between clusters. After obtaining k clusters, each document is tested against cluster definitions represented by pattern sets and assigned to the first one it satisfies, according to the *Satisfy/Violate operator*. This approach has the advantage of producing groups that do not overlap. If documents were not assigned to the first matching pattern set, they could belong to many groups thus producing interleaving clusters.

After clustering the documents according to patterns, there still might be some documents left in the candidate set - documents that did not satisfy any cluster definition. Those documents have to be either assigned with a different method or treated as outliers.

According to the above algorithm description, the order of applying pattern sets in clustering the documents is relevant. If a document satisfies more than one pattern set, it will be assigned to the first presented. This order dependency is a consequence of the definition of the *Satisfy/Violate* operator, which has a binary output - a document satisfies a set or it does not. If the algorithm was to use an operator with a scalar output, all patterns would have to be analyzed for each document. On the other hand, by using the *Satisfy/Violate* operator the proposed algorithm, on an average, needs to analyze only half of the pattern sets. From this point of view, the α parameter can be used to manipulate the rate at which the documents are clustered. The lower the α value, the easier it is for a document to satisfy a pattern set and thus the less pattern sets need to be analyzed.

Now we will present the main steps of XCleaner in an example. The clustering will be performed on documents presented in Fig. 3. We will search for $k = 2$ clusters with patterns with $minsup = 40\%$ and $maxsup = 100\%$.

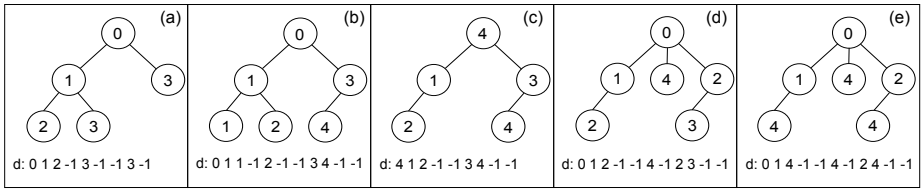

Figure 3. Document trees with their string encoded representations.

Step 1: Obtaining Patterns We apply the CMTreeMiner algorithm on the document set to obtain patterns which are presented in Fig. 4.

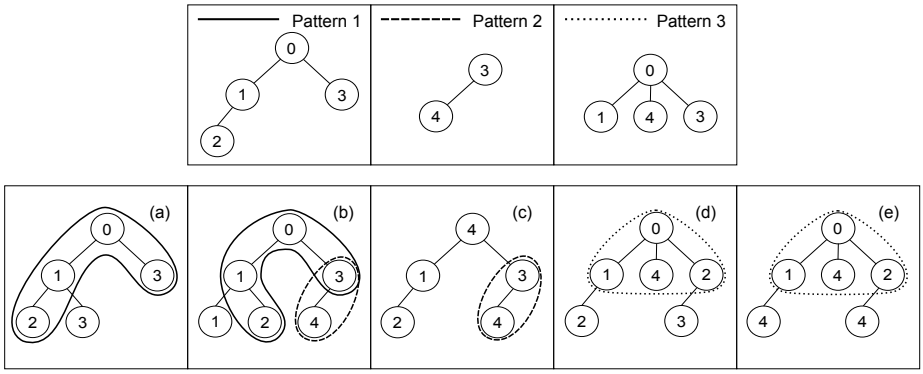

Figure 4. Patterns and their occurrences in the documents.

Table 1. Pattern similarity matrix.

	P1	**P2**	**P3**
P1	-	1	0
P2	1	-	0
P3	0	0	-

Step 2: Pattern Clustering We apply the *Satisfy/Violate* operator on the document set for each pattern. According to Fig. 4, *Pattern 1* co-occurs with *Pattern 2* in one document, so their similarity equals 1, while other pattern pairs do not co-occur with each other and their similarity equals 0. Table 1 presents the full similarity matrix. Since the number of clusters k is 2, the *AHC* algorithm will output two groups: the first will contain *Pattern 1* and *Pattern 2* while the second *Pattern 3*.

Step 3: Document Clustering We use the groups of patterns to cluster the documents. To achieve this goal we take the first group, containing *Patterns 1* and *2*, and apply the *Satisfy/Violate* operator on the entire set of documents. As a result we get documents a, b and c, which form the first cluster. Next, we take the second group, containing *Pattern 3*, and apply the *Satisfy/Violate* operator on the *violate set* from the previous step. This results in assigning the documents d and e to the second cluster. This is the end of the algorithm. The documents are clustered into two groups: one containing documents a, b and c, and the second one containing documents d and e.

3.6. Complexity analysis

Let us analyze the worst-case complexity of each step of our approach. The cost of transforming a single XML document into the chosen representation is equal to the cost of depth-first traversal, that is $O(v + e)$, where v is the number of nodes in the document's tree and e is the number of edges in the document's tree. Therefore, transforming a set D_n of n documents requires $O(n{\cdot}max(v{+}e))$ operations, where $max(v{+}e)$ is the largest sum of the number of edges and vertices found in a single document in D_n. Pattern mining is the most expensive step of our algorithm. According to Chi et al. (see Chi et al., 2005), the cost of mining maximal frequent subtrees is linearly proportional to depth h of each document and exponentially proportional to its width w, giving an overall complexity of $O(2^w)$. The next step, pattern clustering, uses *AHC*, which is a quadratic algorithm, which means that for p patterns the complexity of this step is $O(p^2)$. In the last stage the documents are clustered. This step uses the pattern matching algorithm, which in the worst-case scenario restarts from every possible pattern root found in a document. Therefore, for a single document and pattern, the subtree matching algorithm requires $O((v + e)^2)$ passes. Since

in the worst case scenario each document is compared with each pattern, the complexity of the document clustering step is $O(p \cdot n \cdot max(v+e)^2)$. Given the above, the overall worst-case complexity of our algorithm equals $O(2^{w_{max}})$, where w_{max} is the width of the "widest" document in a data set.

It is easy to notice that the acquired complexity depends mainly on the pattern definition. Maximal frequent subtrees offer comprehensive structural information but imply high computational costs in the pattern mining phase. In practice, the described algorithm performs in acceptable time for documents which are similar in size to those used in the experiments. However, our approach presents a more general idea - a framework which does not imply the use of a specific pattern definition. By using different objects as patterns one can achieve lower overall complexity. For example, narrowing the subtree pattern definition to edges results in a pessimistic complexity of $O(n \cdot e_{max})$ for the pattern mining algorithm, where e_{max} is the maximum number of edges found in a single document in a data set of n objects.

4. Experimental Results

In our experiments we compared the XCleaner to the XProj algorithm (see Aggarwal et al., 2007) and the *Tag only* approach (see Doucet and Ahonen-Myka, 2002). Unfortunately, we were unable to acquire the source code for XProj, so we decided to use the same data sets to make the comparison possible. We took the same real data set, which is the XML SIGMOD database, and for generating the synthetic data sets we used the same DTDs. For the evaluation of the *Tag only* approach we used our implementation of the algorithm with the AHC algorithm for clustering, the *cosine distance* between documents and *complete-linkage* distance between clusters.

4.1. Test Environment

The XCleaner algorithm was implemented in C# and used a C++ implementation of the CMTreeMiner, which served us for maximal frequent subtree mining. The experiments took place on a machine equipped with an Intel Pentium Dual Core E2140 @ 1,60 GHz processor and 3,00 GB of RAM.

4.2. Data Sets

4.2.1. Real Data Sets

The real data set consists of 140 XML documents from the *SIGMOD Record*, which can be downloaded from `http://www.sigmod.org/publications/sigmod-record/xml-edition`. The documents correspond to two DTDs: *IndexTermsPage.dtd* and *OrdinaryIssuePage.dtd* (70 XML documents for each DTD).

4.2.2. Synthetic Data Sets

We used three synthetic data sets: one homogeneous and two heterogeneous. To generate these sets we used the *ToXgene* framework (see Barbosa et al., 2002) and the same DTDs as Aggarwal et al. in (see Aggarwal et al., 2007). The homogeneous set consists of 300 documents grouped into 3 clusters, each containing 100 documents. The *MaxRepeats* parameter, determining the maximum number of times a node will appear as a child of its parent node, was set to 3 for this data set. The heterogeneous data sets, denoted by Heterogeneous_3 and Heterogeneous_6, both contain 1000 XML documents and were generated from 10 different real DTDs, each of which was used to generate 100 documents. The MaxRepeats parameter was set to 3 for Heterogeneous_3 and 6 for Heterogeneous_6.

4.3. Results

To evaluate our clustering method we used the *precision* and *recall* measures (see Aggarwal et al., 2007), defined as follows:

$$precision = \frac{\sum_i s_i}{\sum_i s_i + \sum_i v_i}, \quad recall = \frac{\sum_i s_i}{\sum_i s_i + \sum_i m_i}, \quad i = 1..k$$

where k is the number of clusters, s_i is the number of documents correctly assigned to a cluster C_i, v_i the number of documents incorrectly assigned to C_i and m_i the number of documents which should be, but were not assigned to C_i. Although the described measures originate from supervised learning, we can use them to evaluate our experiments, because all of the data sets we use contain already labeled documents. It is worth noticing that such a situation is unusual, since clustering belongs to the group of unsupervised learning problems. Therefore, in most cases it is necessary to use other evaluation measures such as Cohesion and Separation (see Tan et al., 2005), which require the definition of document similarity.

The selection of the *minsup* value is not an easy task because it highly depends on the data and its complexity. However, if the number of clusters is given a priori (this is one of the assumptions of the algorithm) and the density of clusters is suspected to be uniformly distributed, we suggest to set the *minsup* value to $\frac{1}{k}$, where k is the number of clusters. The *maxsup* value has a smaller impact on the clustering result and for data considered to be easy (such as heterogeneous data sets) it can be set to 1. Manipulating this parameter can be necessary for more difficult data (such as homogeneous data sets). In this case, if we assume, like with *minsup*, that the number of clusters is known and the density of clusters is uniformly distributed, then *maxsup* can be lowered to values close to the *minsup* value. This could help getting better precision but may negatively affect the recall measure.

Considering the above, for mining the patterns in each data set, we set the *minsup* to $\frac{1}{k}$, where k is the number of clusters. Therefore for the SIGMOD data

set *minsup* was equal 50%, for the homogeneous data set 33%, and for both heterogeneous data sets 10%. For the heterogeneous document sets we used the same α and *maxsup* parameters (0% and 100% respectively). To better distinguish similar documents in the homogeneous data set, we set *maxsup* to 34% and α to 30%.

Table 2. Precision and recall for real and synthetic data sets.

Data Set	**Precision**			**Recall**		
	Tags	XProj	XCleaner	Tags	XProj	XCleaner
SIGMOD	1.00	1.00	1.00	1.00	-	1.00
Heterogeneous_3	0.56	1.00	1.00	0.56	1.00	1.00
Heterogeneous_6	0.52	1.00	1.00	0.52	1.00	1.00
Homogeneous_OT	0.51	1.00	1.00	0.51	1.00	0.90
Homogeneous	0.51	1.00	1.00	0.51	1.00	1.00

As Table 2 shows, all compared algorithms present the same quality (according to precision and recall measures) in clustering data from the SIGMOD record. The comparison of the results from synthetic heterogeneous sets shows that the tag only based approach (Tags) identifies only 56% of the smaller and 52% of the larger data set clusters correctly, while both the XProj and the XCleaner still present the best possible precision and recall. It is worth noting that for this type of source, the CMTreeMiner found very few maximal frequent subtrees (5 for SIGMOD and 14 for synthetic sets) and this small amount of information was enough to correctly identify all the documents.

Unfortunately, we were unable to use the same data set as in the XProj for the homogeneous source because of the exponential complexity of the frequent subtrees mining problem. With the MaxRepeats parameter set to 6 we were unable to obtain patterns for *minsup* lower than 65% before the CMTreeMiner ran out of memory. For this reason we used the same DTDs, but unlike the XProj, where the MaxRepeats parameter was set to 6, we used it set to 3. This simplifies the mining of maximal frequent subtrees, but complicates the clustering problem, because larger documents can present higher structural complexity, which helps to distinguish documents from each other. Because of this fact we cannot treat our results as corresponding to the XProj in 100%. Nevertheless, our results for clustering homogeneous documents also presented the highest precision, but 10% (30/300) of the documents were left unassigned. As mentioned earlier in this paper, we propose two approaches to deal with this problem: treating the unassigned documents as outliers or assigning them using some measure of similarity. We tested both approaches and denoted them as: Homogeneous_OT (unassigned documents were treated as outliers) and Homogeneous (unassigned documents were added to clusters using the tag only algorithm). All the results are presented in Table 2.

5. Conclusions and Summary

In this article we have shown a new approach to clustering XML documents by patterns. We employed the idea of the *Satisfy/Violate* operator to data mining purposes and introduced a new clustering algorithm based on that operators. The implemented algorithm has shown good preliminary results on heterogeneous and homogeneous data sets along with small clustering complexity. Our approach was as accurate as the XProj algorithm and outperformed the tag only method. This shows that the *Satisfy/Violate* operator can be successfully used for document clustering.

The presented idea defines a methodology for clustering XML documents. The reusability of this approach opens a new field for further research. The components at each step of the algorithm can be modified or even replaced. As future work, we intend to explore the use of different patterns and grouping algorithms as parts of the methodology. Furthermore, we plan to test the XCleaner on data sets from the INEX contest and evaluate its performance in comparison with other competing algorithms.

Acknowledgments This work was partly supported by the Polish Ministry of Science and Higher Education under Grant No. N N516 365834.

References

Aggarwal, C. C., Ta, N., Wang, J., Feng, J. and Zaki, M. J. (2007) Xproj: a framework for projected structural clustering of xml documents. In: P. Berkhin, R. Caruana and X. Wu, eds., *KDD*. ACM, 46–55.

Barbosa, D., Mendelzon, A. O., Keenleyside, J. and Lyons, K. A. (2002) Toxgene: a template-based data generator for xml. In: M. J. Franklin, B. Moon, and A. Ailamaki, eds., *SIGMOD Conference*. ACM, 616.

Candillier, L., Tellier, I. and Torre, F. (2005) Transforming xml trees for efficient classification and clustering. In: N. Fuhr, M. Lalmas, S. Malik and G. Kazai, eds., *INEX*. LNCS 3977, Springer, 469–480.

Chawathe, S. S. (1999) Describing and manipulating xml data. *IEEE Data Eng. Bull.*, 22 (3), 3–9.

Chi, Y., Xia, Y., Yang, Y., and Muntz, R. R. (2005). Mining closed and maximal frequent subtrees from databases of labeled rooted trees. *IEEE Trans. Knowl. Data Eng.*, 17 (2), 190–202.

Costa, G., Manco, G., Ortale, R., and Tagarelli, A. (2004) A tree-based approach to clustering xml documents by structure. In: J.-F. Boulicaut, F. Esposito, F. Giannotti and D. Pedreschi, eds., *PKDD*. LNCS 3202, Springer, 137–148.

Dalamagas, T., Cheng, T., Winkel, K.-J., and Sellis, T. K. (2004) Clustering xml documents by structure. In: G. A. Vouros and T. Panayiotopoulos, eds., *SETN*, LNCS 3025, Springer, 112–121.

Doucet, A. and Ahonen-Myka, H. (2002) Naïve clustering of a large xml document collection. In: N. Fuhr, N. Govert, G. Kazai and M. Lalmas, eds., *INEX Workshop*. ERCIM, 81–87.

Florescu, D. and Kossmann, D. (1999) Storing and querying xml data using an rdmbs. *IEEE Data Eng. Bull.*, 22 (3), 27–34.

Jain, A. K., Murty, M. N., and Flynn, P. J. (1999). Data clustering: A review. *ACM Comput. Surv.*, 31 (3), 264–323.

Johnson, S. (1967). Hierarchical clustering schemes. *Psychometrika*, 32 (3), 241–254.

Lee, M.-L., Yang, L. H., Hsu, W., and Yang, X. (2002). Xclust: clustering xml schemas for effective integration. In: *CIKM*. ACM, 292–299.

Lesniewska, A. (2009) Clustering xml documents by structure. In: J. Grundspenkis, M. Kirikova, Y. Manolopoulos and L. Novickis, eds., *ADBIS (Workshops)*. LNCS 5968, Springer, 238–246.

Lian, W., Cheung, D. W.-L., Mamoulis, N., and Yiu, S.-M. (2004) An efficient and scalable algorithm for clustering xml documents by structure. *IEEE Trans. Knowl. Data Eng.*, 16 (1), 82–96.

Milligan, G. and Cooper, M. (1985) An examination of procedures for determining the number of clusters in a data set. *Psychometrika*, 50, 159–179. 10.1007/BF02294245.

Nayak, R. and Iryadi, W. (2006) Xmine: A methodology for mining xml structure. In: X. Zhou, J. Li, H. T. Shen, M. Kitsuregawa and Y. Zhang, eds., *APWeb*. LNCS 3841, Springer, 786–792.

Tan, P.-N., Steinbach, M., and Kumar, V. (2005) *Introduction to Data Mining*. Addison Wesley.

Tran, T., Nayak, R., and Bruza, P. (2008) Combining structure and content similarities for xml document clustering. In: J. F. Roddick, J. Li, P. Christen and P. J. Kennedy, eds., *AusDM*. CRPIT 87, Australian Computer Society, 219–226.

Widom, J. (1999) Data management for xml: Research directions. *IEEE Data Eng. Bull.*, 22 (3), 44–52.

Zaki, M. J. (2002) Efficiently mining frequent trees in a forest. In: *KDD*. ACM, 71–80.