**Noname manuscript No.**
(will be inserted by the editor)

# Information-Theoretic Approaches for Measuring the Structural Similarity of Semistructured Documents

**Sven Helmer, Nikolaus Augsten, and Michael Böhlen**

the date of receipt and acceptance should be inserted later

**Abstract** We propose and experimentally evaluate different approaches for measuring the structural similarity of semistructured documents based on information-theoretic concepts. Common to all approaches is a two-step procedure: first we extract and linearize the structural information from documents and then we use similarity measures that are based on, respectively, Kolmogorov complexity and Shannon entropy to determine the distance between the documents. Compared to other approaches, we are able to achieve a linear run time complexity and demonstrate in an experimental evaluation that the results of our technique in terms of clustering quality are on a par with or even better than those of other, slower approaches.

## 1 Introduction

Detecting similarities within a group of unstructured text documents is widely used to, e.g., cluster documents by topic [6], and similarity measures for unstructured text have been researched since the 1970s. Today, however, more and more information is stored in semistructured documents, be it in HTML, XHTML, or XML, and ever larger collections of these documents can be searched online. The advent of semistructured documents brought about new opportunities and challenges for similarity measures, since the structure carries important information about the similarity of documents.

Semistructured documents can exhibit a lot of diversity, due to a potentially arbitrary number of optional and repeating elements. For example, even if two XML documents follow the same schema, some elements might be present in one document only and the documents may have completely different sizes. Nevertheless, we want such documents to be similar in terms of structure. This poses a major challenge for standard similarity measures that operate on trees (such as a tree edit distance), since these measures usually assume that similar trees are alike in shape and size. As a consequence these measures perform poorly on semistructured documents.

Our goal in this paper is to find and evaluate a structural similarity measure that is effective for semistructured documents and can be computed efficiently. We investigate information-theoretic approaches for measuring the structural similarity of semistructured documents. Measures based on information theory have been applied successfully to a range of application domains such as genomics, literature, music, and astronomy [19]. Their main strength is their universality, i.e., these techniques make few assumptions about underlying probability distributions or models describing the creation process of data objects. Existing similarity measures often map objects into the Euclidean space to measure distances, which is not always the most appropriate way to determine similarity as the Euclidean distance may differ from a user's perception [43].

Since we want to measure similarity based on structure and not content, we propose a new two-step approach. In the first step we extract the structural information from a document by stripping away the content (text and attribute values in XML) and representing the document by a structure vector. In the second step, we use information-theoretic approaches to measure the similarity between the structure vectors and thus between the structure of the documents.

We use our similarity measure to cluster XML documents by their schema and show that our technique outperforms the state-of-the-art approaches and has a very

Address(es) of author(s) should be given

good time complexity (linear in the document size). In more detail our contributions are the following:

– We introduce a *structure vector* to represent the structure of an XML document. We discuss six different approaches to compute a structure vector, each capturing different structural aspects.
– We present two information-theoretic approaches, the normalized compression distance (NCD) and the crossparsing distance (CPD), to compute the distance between structure vectors, i.e., XML documents. NCD is based on Kolmogorov complexity, whereas CPD is based on Shannon entropy.
– *Crossparsing* comes out on top for the information-theoretic measures, while *pq-grams* do so for the structure extraction techniques. Their combination, *crossparsed pq-grams*, is an excellent technique for measuring structural similarity. Furthermore, this measure can be computed in linear time.
– We provide a *benchmark* for structural similarity measures with ten real world and synthetic datasets. In our extensive experiments we benchmark our own solution and five other state-of-the-art techniques. Crossparsing, combined with certain structure extraction methods (among them *pq*-grams), outperforms all competitors in terms of both effectiveness and efficiency.

Applications in which it is relevant to measure the structural similarity include schema extraction [1,26, 37,40], the integration of heterogeneous data sources [8, 24,35], retrieving information from semistructured data sources [1,22], data cleaning [11,15,29], and systems that support approximate queries or querying by example [2,12]. As an example, consider the extraction of schema or document type information from a document collection [1,26,40]. The more homogeneous a group of documents is, i.e., the fewer out-of-place documents are contained in it, the better these extraction algorithms work. This holds for the run-time of the algorithms as well as the quality of their output (a user is rarely interested in obtaining one very general schema applicable to every single document in the collection). Thus, it is usually a good idea to first cluster the documents and then apply the extraction algorithms to each cluster in turn [37].

The remainder of this paper is structured as follows. Section 2 discusses related work. In Section 3 we provide basic definitions needed for introducing the concepts used in later parts of the paper. We describe the two phases of our approach (structure extraction and similarity measure) in a formal way in Sections 4 and 5, while Section 6 covers the implementation. In Section 7 we introduce competing approaches that are experimentally compared to our solution in Section 8. We conclude the paper with a summary and outlook in Section 9.

## 2 Related Work

The earliest attempts for detecting structural similarity go back to computing tree-edit distances [44,46,50,53, 56]. These approaches have two major disadvantages when it comes to semistructured documents: they have at least quadratic complexity and they operate on a node basis, i.e., the basic unit of editing is one node. Chawathe et al. [16,17] have enhanced their tree-editing algorithms by subtree editing operations, but they have not improved on the quadratic complexity. Nierman and Jagadish [41] developed an algorithm that better reflects the properties of XML documents, but still does not improve the run-time. Dalamagas et al. [23] improved the performance of the tree-edit distance computation by precomputing tree structural summaries (eliminating element repetition and nesting) and then using these summaries for the comparison. Although this speeds up the similarity computation, the underlying asymptotic complexity does not change. In summary, tree-edit distances suffer from two key drawbacks: many of the approaches are inefficient, and it is difficult to define the subtree editing operations properly.

Flesca et al. [25] take a completely different approach by quantifying the structures of two documents and interpreting the results as time series. The two time series are then analyzed and compared using the Discrete Fourier Transformation (DFT), resulting in an algorithm with a complexity of $O(N \log N)$ for comparing two documents (where $N$ is the size of the larger document). This approach works well for distinguishing repeated elements within documents, but is less effective when documents consist of a large proportion of unique elements.

Buttler [13] uses an approach that is based on path shingles. This approach reduces the paths in a document to hash values, which can be compared to those of another document using set union and set intersection operators. They improve the quadratic performance, but in the worst case the performance is still not linear.

Cherukuri and Candan propagate data that describes the relationships between different element names along the parent-child axes in semistructured documents. The relationships are represented by vectors and the technique is named Propagation-Vectors for Trees (PVT) [18]. In order to reduce the space complexity only the vectors for the root nodes of each document are kept as a summary and these summaries are used for measuring the structural similarity. The memory requirements for computing the summaries are still impractical, however.

Cilibrasi and Vitányi applied information-theoretic concepts to measure the similarity between data objects [19]. Directly applying their approach to determine the similarity between XML documents yields poor results since the structure of the semistructured documents is not considered adequately. Based on their approach Helmer [28] has applied an entropy-based approach for measuring the similarity of semistructured documents, which is the first linear algorithm for measuring structural similarity. We extend this work by combining information theoretic similarity measures with a sophisticated structure extraction method, called *pq*-grams, pursued by Augsten et al. [5]. Up to now, *pq*-grams have been compared using a simple Dice coefficient, which does not provide the same quality in terms of clustering. Augsten et al. extend their work in [4], showing that the *pq*-gram distance can be used to approximate a fanout weighted tree edit distance.

Theobald and Weikum developed a similarity operator in the query language XXL to find element names similar to query terms using an ontology [51]. For example, a search for "drama" is expanded to include "play". This work is mainly concerned with mapping element names and uses context information, such as an ontology and the surrounding structure, to determine semantic similarities of labels.

It is also worthwhile to look at concepts of similarity in data-centric settings, where the order of siblings is not relevant. Windowed *pq*-grams [3], which extend *pq*-grams for data-centric XML, operate on a sorted tree and use a window to simulate permutations of the sibling. Similar to Theobald and Weikum [51], Mesiti et al. [39] mainly look at the structure of XML documents on a local level, while we are interested in capturing the overall structure.

Finally, there is ongoing work on detecting similarity not between XML documents, but between documents and DTDs [10]. This plays a role for example in the context of schema validation, but is not within the scope of this paper.

## 3 Preliminaries

Hierarchical data, such as XML documents, can be represented as rooted, ordered, labeled trees. A *tree* $\mathbf{T}$ is a directed, acyclic, connected graph with *nodes* $N(\mathbf{T})$ and edges $E(\mathbf{T})$. An *edge* is an ordered pair $(\mathsf{p}, \mathsf{c})$, where $\mathsf{p}, \mathsf{c} \in N(\mathbf{T})$ are nodes, and $\mathsf{p}$ is the *parent* of $\mathsf{c}$. A node can have at most one parent, and nodes with the same parent are *siblings*. We denote the parent of a node $\mathsf{x}$ as $\mathrm{par}(\mathsf{x})$, and the size of a tree with $|\mathbf{T}| = |N(\mathbf{T})|$.

An order $\leq$ is defined on the nodes, and this order is total among siblings. Siblings $\mathsf{s}_1 \leq \mathsf{s}_2$ ($\mathsf{s}_1 \neq \mathsf{s}_2$) are *contiguous* if $\mathsf{s}_1$ and $\mathsf{s}_2$ have no sibling $\mathsf{x}$ ($\mathsf{s}_1 \neq \mathsf{x} \neq \mathsf{s}_2$) with $\mathsf{s}_1 \leq \mathsf{x} \leq \mathsf{s}_2$. Let $\mathrm{presib}(j, \mathsf{v})$ denote $\mathsf{v}$'s preceding sibling at position $j$. The preceding siblings of a node are ordered backwards, i.e., the immediate sibling is at position 1. Formally, for two preceding siblings $\mathsf{s}_1$ and $\mathsf{s}_2$ of a node $\mathsf{v}$ with $\mathsf{s}_1 \leq \mathsf{s}_2 \leq \mathsf{v}$ ($\mathsf{s}_1 \neq \mathsf{s}_2 \neq \mathsf{v}$), $\mathsf{s}_1 = \mathrm{presib}(j, \mathsf{v})$, and $\mathsf{s}_2 = \mathrm{presib}(j', \mathsf{v})$ we have $j > j'$.

Let $\mathrm{ch}(\mathsf{x})$ denote the list of all children of $\mathsf{x}$ in $\leq$-order and $\mathrm{ch}(\mathsf{x}, i)$ denote the $i$-th child of $\mathsf{x}$. The number of $\mathsf{p}$'s children is its *fanout* $f_{\mathsf{p}}$. The node with no parent is the *root* node $\mathsf{r} = \mathrm{root}(\mathbf{T})$, and a node without children is a *leaf*. Each node $\mathsf{a}$ in the path from the root node to a node $\mathsf{v}$ is called an *ancestor* of $\mathsf{v}$. If there is a path of length $k > 0$ from $\mathsf{a}$ to $\mathsf{v}$, then $\mathsf{a}$ is the ancestor of $\mathsf{v}$ at distance $k$, denoted by $\mathsf{a} = \mathrm{anc}(k, \mathsf{v})$. The parent of a node is its ancestor at distance 1. $\mathsf{d}$ is a *descendant* of $\mathsf{v}$ if $\mathsf{v}$ is an ancestor of $\mathsf{d}$. The *level* of a node $\mathsf{v}$, $\mathrm{level}(\mathsf{v})$, is the length of the path from the root to $\mathsf{v}$ (with the root node itself being located on level 0). The *depth* of a tree $\mathbf{T}$, $\mathrm{depth}(\mathbf{T})$, is the length of the longest path from the root to any one of the leaves.

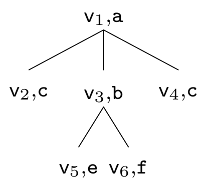
A *label* is a symbol $\sigma \in \mathbf{\Sigma}$, where $\mathbf{\Sigma}$ is a finite alphabet. Each node $\mathsf{v} \in N(\mathbf{T})$ is assigned a label $\mathrm{l}(\mathsf{v})$. A node $\mathsf{o}$ with the label $\mathrm{l}(\mathsf{o}) = *$ is a *dummy node*.[1] If we apply $\mathrm{l}(.)$ to a list of nodes $\mathsf{x}_1, \mathsf{x}_2, \ldots, \mathsf{x}_n$ (e.g., all children of a node), it returns a sequence of labels for these nodes: $\mathrm{l}((\mathsf{x}_1, \mathsf{x}_2, \ldots, \mathsf{x}_n)) = (\mathrm{l}(\mathsf{x}_1), \mathrm{l}(\mathsf{x}_2), \ldots, \mathrm{l}(\mathsf{x}_n))$.

In the graphical representation of trees we represent nodes as (identifier, label)-pairs, the edges are lines between the nodes, and siblings are ordered from left to right. Whenever possible, we omit the identifiers of nodes to avoid clutter. A node in the tree represents an XML element (or attribute). The node label is the name of the element (or attribute). An edge connects an element node with each of its subelements (or attributes). The attributes of an element precede its subelements. The order of sibling nodes is defined by the order of the respective elements in XML. For attributes, XML does not define an order, and we sort attributes lexicographically by their name. Since we are only interested in the structure of XML documents, text nodes and attribute values are ignored.

*Example 1* Figure 1(a) shows a tree with $N(\mathbf{T}_1) = \{\mathsf{v}_1, \mathsf{v}_2, \mathsf{v}_3, \mathsf{v}_4, \mathsf{v}_5, \mathsf{v}_6\}$, $E(\mathbf{T}_1) = \{(\mathsf{v}_1, \mathsf{v}_2), (\mathsf{v}_1, \mathsf{v}_3), (\mathsf{v}_1, \mathsf{v}_4), (\mathsf{v}_3, \mathsf{v}_5), (\mathsf{v}_3, \mathsf{v}_6)\}$, and the order $\mathsf{v}_2 \leq \mathsf{v}_3 \leq \mathsf{v}_4$, $\mathsf{v}_5 \leq \mathsf{v}_6$. The root node of $\mathbf{T}_1$ is $\mathrm{root}(\mathbf{T}_1) = \mathsf{v}_1$, i.e., $\mathsf{v}_1$ is the ancestor of all other nodes. $\mathsf{v}_2$, $\mathsf{v}_5$, $\mathsf{v}_6$ and $\mathsf{v}_4$ are leaf nodes. $\mathsf{v}_1$ has 3 children, where $\mathsf{v}_2$ is the first, $\mathsf{v}_3$ the second, and

[1] This is relevant for the *pq*-gram structure extraction technique, more details are provided in Section 4.6.

$v_4$ the third child. More formally, $\mathrm{ch}(v_1) = (v_2, v_3, v_4)$ and $\mathrm{ch}(v_1, 1) = v_2$, $\mathrm{ch}(v_1, 2) = v_3$, $\mathrm{ch}(v_1, 3) = v_4$. The nodes $v_2, v_3$, and $v_4$ are siblings, so $\mathrm{presib}(1, v_4) = v_3$ and $\mathrm{presib}(2, v_4) = v_2$. The node labels of our example tree are $\mathrm{l}(v_1) = \mathtt{a}$, $\mathrm{l}(v_2) = \mathtt{c}$, $\mathrm{l}(v_3) = \mathtt{b}$, $\mathrm{l}(v_4) = \mathtt{c}$, $\mathrm{l}(v_5) = \mathtt{e}$, and $\mathrm{l}(v_6) = \mathtt{f}$.



(a) Tree $\mathbf{T}_1$

```
<a>
  <c> some text </c>
  <b>
    <e> some more text </e>
    <f> even more text </f>
  </b>
  <c/>
</a>
```

(b) XML document

**Fig. 1** Example Tree and Corresponding XML Document.

A *subtree* $\mathbf{S}$ of $\mathbf{T}$ is a tree with $N(\mathbf{S}) \subseteq N(\mathbf{T})$ and $E(\mathbf{S}) \subseteq E(\mathbf{T})$ that retains the node order. A *preorder traversal* of a tree visits the root node first, and then recursively traverses all the subtrees rooted in its children in preorder, preserving the children's order. For example, the preorder traversal of $\mathbf{T}_1$ in Figure 1 is $v_1, v_2, v_3, v_5, v_6, v_4$. The *postorder traversal* recursively traverses all the children of a node before visiting the node itself. The postorder traversal of $\mathbf{T}_1$ in Figure 1 is $v_2, v_5, v_6, v_3, v_4, v_1$.

## 4 Extracting and Linearizing Structural Information

This section defines structure vectors and covers six different approaches for computing structure vectors given a semistructured document: Doc, Doc+, Pair, Path, Family, and *pq*-grams. We illustrate the different techniques by applying them to the example tree from Figure 1.

The *structure vector*, $V(\mathbf{T})$, of a tree $\mathbf{T}$ captures its structural information. Each component of the structure vector is a sequence of node labels (possibly including dummy labels) and represents a subtree or a contiguous sequence of siblings in $\mathbf{T}$. The nodes of a subtree are sorted in preorder, and the labels of the nodes form a component of the structure vector. The subtrees stored in a structure vector may overlap.

*Example 2* Let us assume that we represent tree $\mathbf{T}_1$ (Figure 1) by two subtrees rooted at $v_1$ and $v_3$, respectively, and their children. The corresponding structure vector then is $V(\mathbf{T}_1) = ((\mathtt{a}, \mathtt{c}, \mathtt{b}, \mathtt{c}), (\mathtt{b}, \mathtt{e}, \mathtt{f}))$.

### 4.1 Doc

In *Doc* order, each node of the tree is a sequence in the structure vector, and the order of the nodes is defined by the preorder in the tree, which is the canonical way of ordering elements in an XML document. For example, the Doc structure vector of $\mathbf{T}_1$ is $\mathrm{Doc}(\mathbf{T}_1) = ((\mathtt{a}), (\mathtt{c}), (\mathtt{b}), (\mathtt{e}), (\mathtt{f}), (\mathtt{c}))$.

**Definition 1 (Doc)** Let $\mathbf{T}$ be a tree, $n = |\mathbf{T}|$, $v_{i_j}$ the $j$-th node of $\mathbf{T}$ in preorder. The *Doc* structure vector is defined as $\mathrm{Doc}(\mathbf{T}) = ((\mathrm{l}(v_{i_1})), (\mathrm{l}(v_{i_2})), \ldots, (\mathrm{l}(v_{i_n})))$.

### 4.2 Doc+

In *Doc+* order, each node of a tree appears twice: once when first encountered, the second time when leaving the subtree rooted at that node. The first occurrence of a node in the structure vector uses the start tag as its label, the second occurrence uses the end tag. This outputs all start and end tags of an XML file in document order. For example, the Doc+ structure vector of $\mathbf{T}_1$ is $\mathrm{Doc+}(\mathbf{T}_1) = ((\mathtt{a}), (\mathtt{c}), (\mathtt{/c}), (\mathtt{b}), (\mathtt{e}), (\mathtt{/e}), (\mathtt{f}), (\mathtt{/f}), (\mathtt{/b}), (\mathtt{c}), (\mathtt{/c}), (\mathtt{/a}))$.

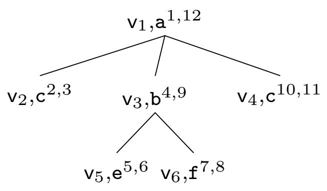


**Fig. 2** Nested-set traversal indexing.

**Definition 2 (Doc+)** Let $\mathbf{T}$ be a tree, $n = |\mathbf{T}|$, $v_{i_j}$ the $j$-th node of $\mathbf{T}$ in nested-set traversal. In nested-set traversal all nodes are numbered using natural numbers from the interval $[1, 2n]$. Every node receives two numbers: one when first visited, another when its subtree is left, for an example see Figure 2. These numbers are used as an index for the position of a node in the structure vector. The *Doc+* structure vector is defined as $\mathrm{Doc}(\mathbf{T}) = ((\mathrm{l}(v_{i_1})), (\mathrm{l}(v_{i_2})), \ldots, (\mathrm{l}(v_{i_{2n}})))$. If a node is encountered the second time, then instead of its original label a correspoding end tag is output.

Note that *Doc* and *Doc+* describe relationships between nodes only implicitly. The following two extraction techniques, *Pair* and *Path*, make these relationships more explicit.

## 4.3 Pair

Sequences in the *Pair* structure vector consist of parent-child pairs and they are ordered by the preorder position of the child. The root node does not have a parent and is a sequence on its own. For example, the Pair structure vector of $\mathbf{T}_1$ is $\text{Pair}(\mathbf{T}_1) = ((\mathtt{a}), (\mathtt{a}, \mathtt{c}), (\mathtt{a}, \mathtt{b}), (\mathtt{b}, \mathtt{e}), (\mathtt{b}, \mathtt{f}), (\mathtt{a}, \mathtt{c}))$.

**Definition 3 (Pair)** Let $\mathbf{T}$ be a tree, $n = |\mathbf{T}|$, $\mathsf{v}_{i_j}$ the $j$-th node of $\mathbf{T}$ in preorder. The *Pair* structure vector is defined as $\text{Pair}(\mathbf{T}) = (\text{l}((\mathsf{v}_{i_1})), \text{l}((\text{par}(\mathsf{v}_{i_2}), \mathsf{v}_{i_2})), \ldots, \text{l}((\text{par}(\mathsf{v}_{i_n}), \mathsf{v}_{i_n})))$.

## 4.4 Path

Each sequence in the *Path* structure vector consists of a node and all its ancestors. The paths are ordered by the preorder number of their leaf nodes. For example, the Path structure vector of $\mathbf{T}_1$ is $\text{Path}(\mathbf{T}_1) = ((\mathtt{a}), (\mathtt{a}, \mathtt{c}), (\mathtt{a}, \mathtt{b}), (\mathtt{a}, \mathtt{b}, \mathtt{e}), (\mathtt{a}, \mathtt{b}, \mathtt{f}), (\mathtt{a}, \mathtt{c}))$.

**Definition 4 (Path)** Let $\mathbf{T}$ be a tree, $n = |\mathbf{T}|$, and $\mathsf{v}_{i_j}$ the $j$-th node of $\mathbf{T}$ in preorder. The root path of $\mathsf{v}_{i_j}$ is the sequence of nodes on the path from the root to $\mathsf{v}_{i_j}$ (including $\mathsf{v}_{i_j}$ itself), $\text{rp}(\mathsf{v}_{i_j}) = (\text{anc}(\text{level}(\mathsf{v}_{i_j}), \mathsf{v}_{i_j}), \text{anc}(\text{level}(\mathsf{v}_{i_j}) - 1, \mathsf{v}_{i_j}), \ldots, \text{anc}(1, \mathsf{v}_{i_j}), \mathsf{v}_{i_j})$. Then the *Path* structure vector is $\text{Path}(\mathbf{T}) = (\text{l}(\text{rp}(\mathsf{v}_{i_1})), \text{l}(\text{rp}(\mathsf{v}_{i_2})), \ldots, \text{l}(\text{rp}(\mathsf{v}_{i_n})))$.

## 4.5 Family

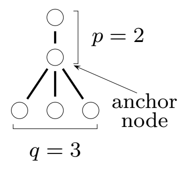
An important concept of XML schemas is the grouping of subelements such as siblings. This grouping is not captured explicitly by the structure vectors discussed so far, which mainly look at relationships between nodes on vertical axes. The *Family* structure vector is composed of the sequences of all children of each node (and the single-node sequence with the root node), which means that relationships along horizontal axes are considered. Within the structure vector the child sequences are ordered by the postorder position of the last node in each sequence. For example, the Family structure vector of $\mathbf{T}_1$ is $\text{Family}(\mathbf{T}_1) = ((\mathtt{e}, \mathtt{f}), (\mathtt{c}, \mathtt{b}, \mathtt{c}), (\mathtt{a}))$.

**Definition 5 (Family)** Let $\mathbf{T}$ be a tree, $n = |\mathbf{T}|$, $\mathsf{v}_{i_j}$ the $j$-th node of $\mathbf{T}$ in *postorder*, and $\text{ch}(\mathsf{v}_{i_j})$ the sequence of all children of $\mathsf{v}_{i_j}$. The *Family* structure vector is $\text{Family}(\mathbf{T}) = (\text{l}(\text{ch}(\mathsf{v}_{i_1})), \text{l}(\text{ch}(\mathsf{v}_{i_2})), \ldots, \text{l}(\text{ch}(\mathsf{v}_{i_n})), \text{l}((\mathsf{v}_{i_n})))$, where empty child sequences are omitted.

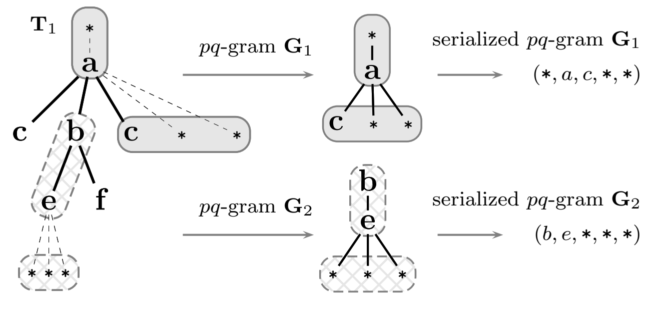
Computing the Family structure vector involves traversing the tree in the so-called family order [32], a combination of breadth-first and depth-first traversal. In family order, a node is visited before its right siblings, after its own children, and after the children of its siblings. The breadth-first traversal of the tree could also be used to group siblings together, but the family order traversal requires less memory to compute. In family order, only the left siblings of all ancestors of a node must be stored, whereas breadth-first stores all nodes of a tree level. This is relevant for XML trees, which are typically flat and bushy [7].

## 4.6 *pq*-Gram

A *pq*-gram is a small subtree consisting of an anchor node, $p-1$ ancestors, and $q$ consecutive children. Intuitively, the *pq*-grams are formed by shifting a broom-shaped pattern over the tree (see Figure 3). The nodes covered by the pattern form a *pq*-gram. The pattern is shifted in such a way that each node appears in the anchor position and each non-root node also appears in each leaf position of the pattern. The parts of the pattern that extend beyond the tree border are filled with dummy nodes $\mathsf{o}_i$ that all have the same, special label $\text{l}(\mathsf{o}_i) = *$.



(a) *pq*-Gram Pattern.



(b) Example Tree $\mathbf{T}_1$ and Two 2, 3-Grams of $\mathbf{T}_1$.

**Fig. 3** Computing the *pq*-Grams of a Tree ($p = 2, q = 3$).

**Definition 6 (*pq*-Gram)** Let $\mathbf{T}$ be a tree, $p > 0$, $q > 0$, and let $\mathbf{T}^{p,q}$ be a version of $\mathbf{T}$ extended with dummy nodes as follows: $p-1$ ancestors above the root node, $q-1$ children before the first and after the last child of each non-leaf node, and $q$ children below each leaf. A *pq-gram* of $\mathbf{T}$ with *anchor node* $\mathtt{a}$, $\mathtt{a} \in V(\mathbf{T})$, is a

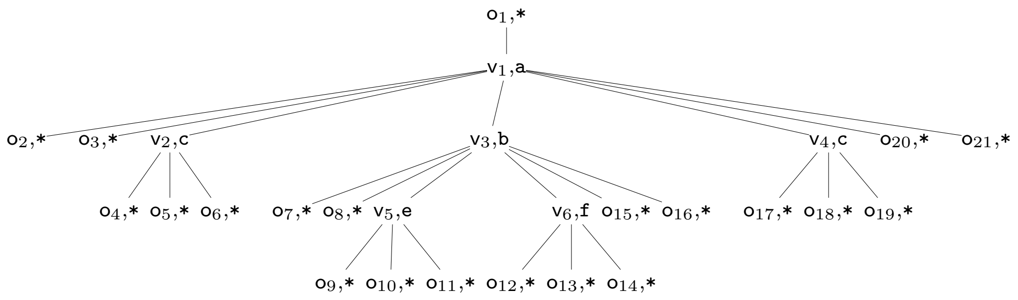


**Fig. 4** Extended Tree $\mathbf{T}_1^{p,q}$ for Example Tree $\mathbf{T}_1$ in Figure 1 ($p = 2, q = 3$).

subtree of $\mathbf{T}^{p,q}$ that is composed of the following nodes: $p-1$ ancestor nodes $\text{anc}(p-1, \mathtt{a}), \ldots, \text{anc}(1, \mathtt{a})$, the anchor node $\mathtt{a}$, and $q$ contiguous children $\mathtt{c}_k, \ldots, \mathtt{c}_{k+q-1}$ of $\mathtt{a}$. We serialize these subtrees by representing them as an ordered sequence of node labels, so the *pq*-gram created for the anchor node $\mathtt{a}$ is equal to $\text{l}(\text{anc}(p-1, \mathtt{a})), \ldots, \text{l}(\text{anc}(1, \mathtt{a})), \text{l}(\mathtt{a}), \text{l}(\mathtt{c}_k), \ldots, \text{l}(\mathtt{c}_{k+q-1})$.

Altogether there are $2l + qi - 1$ *pq*-grams in a tree $\mathbf{T}$ where $l$ and $i$ are the number of leaf and inner nodes of $\mathbf{T}$, respectively [4]. As we will see in Section 6.1 the *pq*-grams of a tree can be computed in linear time by a single preorder scan of the tree (which is equivalent to a single scan of an XML document [4]).

**Definition 7 (*pq*-Gram Structure Vector)** Let $\mathbf{T}$ be a tree and $\mathbf{T}^{p,q}$ its extended version as described above. The *pq-gram* structure vector, pqGram($\mathbf{T}$), is defined as the sequence of all serialized *pq*-grams of $\mathbf{T}$, where the *pq*-grams are ordered by the preorder numbers in $\mathbf{T}^{pq}$ of their rightmost leaf node.

Figure 4 shows the example tree $\mathbf{T}_1$ extended with dummy nodes. The *pq*-gram vector for the example tree $\mathbf{T}_1$ is pqGram($\mathbf{T}_1$) = ((*,a,*,*,c), (a,c,*,*,*), (*,a,*,c,b), (a,b,*,*,e), (b,e,*,*,*), (a,b,*,e,f), (b,f,*,*,*), (a,b,e,f,*), (a,b,f,*,*), (*,a,c,b,c), (a,c,*,*,*), (*,a,b,c,*), (*,a,c,*,*)).

## 4.7 Summary

Ideally, we would like to have a structure vector that is able to capture vertical as well as horizontal relationships between nodes and does so in an explicit manner. Pair and Path describe vertical relationships among nodes in a tree along the parent-child axis, while Family is mainly concerned with horizontal relationships along the sibling axis. *pq*-Grams capture both the vertical and the horizontal tree structure. Another important aspect is that each component of a structure vector gives a local view of a document's structure. If these components are completely disconnected from each other, then we may lose knowledge about the global structure. For example, the Doc vector ((a), (b), (b)) could describe the structure of a document in which the b elements are all children of a or the second b element is a child of the first one. Having overlapping components is a feature allowing us to distinguish (some of) these cases. Clearly a balance has to be struck, as adding too much overlapping information would make a scheme inefficient. Doc and Family yield structure vectors that do not overlap, while Path, pq-grams, and Pair yield structure vectors that overlap.

## 5 Information-Theoretic Approaches

The previous section discussed different approaches to extract and linearize the structure from semistructured documents. This section proposes two solutions that use this information to measure the similarity between documents. We use techniques that are based on two information-theoretic concepts: Kolmogorov complexity and Shannon entropy. In Section 5.1 we introduce Kolmogorov complexity and illustrate how a similarity measure can be derived from it, while in Section 5.2 we do the same for Shannon entropy. This is followed by a discussion on what these concepts have in common.

### 5.1 Kolmogorov Complexity and Similarity

Kolmogorov complexity (also called algorithmic complexity) was developed independently by Kolmogorov [33], Solomonoff [48,49], and Chaitin [14], and it measures the information content of an object. An important feature is its universality since it makes very few assumptions about how a data object was generated.

**Definition 8 (Kolmogorov Complexity)** Given a finite binary string $x$ in $\{0,1\}^*$, the Kolmogorov complexity $K(x)$ is the length of the shortest (binary) program for a universal computer (such as a Turing machine) that outputs $x$.

Definition 8 assumes that the program generates the object $x$ from scratch, i.e., it starts with an empty input. A generalized version of the Kolmogorov complexity defines the complexity of an object given another object.

**Definition 9 (Conditional Kolmogorov Complexity)** The conditional Kolmogorov complexity $K(x|y)$ is the length of the shortest program with input $y$ that outputs $x$.

### *5.1.1 Normalized Information Distance*

Bennet et al. define a similarity function based on Kolmogorov complexity [9]:

**Definition 10 (Normalized Information Distance)** Given two data objects $x$ and $y$, the normalized information distance (or $NID$) is defined as follows:

$$dist_{NID}(x, y) = \frac{\max\{K(x|y), K(y|x)\}}{\max\{K(x), K(y)\}} \tag{1}$$

They show that this information distance exhibits several desirable properties (e.g., it is a metric, up to negligible violations of the metric inequalities, and it is a lower bound for admissible distances[2]). The only catch is that the Kolmogorov complexity is not computable in the general case.

### *5.1.2 Normalized Compression Distance*

The Kolmogorov complexity of a data object $x$ can be interpreted as the length of the ultimately compressed version of $x$. Obviously, the ultimate compression is not computable either, but we can approximate it by compressing $x$ with a standard algorithm. Let $C(x)$ be the length of the compressed $x$ using a standard compression technique. Cilibrasi and Vitányi [19] defined a normalized compression distance ($NCD$) derived from the $NID$. Rewriting the denominator of the $NID$ poses no problem, we just replace $K(x)$ by $C(x)$. The numerator, containing conditional Kolmogorov complexity expressions, needs to be rewritten before approximating it by compression. The numerator of Equation (1) can be rewritten to $\max\{K(x, y) - K(x), K(x, y) - K(y)\}$ within logarithmic additive precision [36], where $K(x, y)$ is the length of the shortest program needed to produce the pair $(x, y)$ of data objects.[3] As it is easier to handle concatenation with compression, this can be estimated by $\min\{C(x \circ y), C(y \circ x)\} - \min\{C(x), C(y)\}$. Cilibrasi and Vitányi assume that most compression algorithms exhibit symmetrical behavior, so they simplify the numerator further to $C(x \circ y) - \min\{C(x), C(y)\}$, arriving at the following definition of $NCD$ [19]:

**Definition 11 (Normalized Compression Distance)** Given two data objects $x$ and $y$, the normalized compression distance is defined as follows:

$$dist_{NCD}(x, y) = \frac{C(x \circ y) - \min\{C(x), C(y)\}}{\max\{C(x), C(y)\}} \tag{2}$$

*Example 3* We apply the normalized compression distance by computing the distance between tree $\mathbf{T}_1$ from Figure 1 and trees $\mathbf{T}_2$ and $\mathbf{T}_3$ shown in Figure 5.

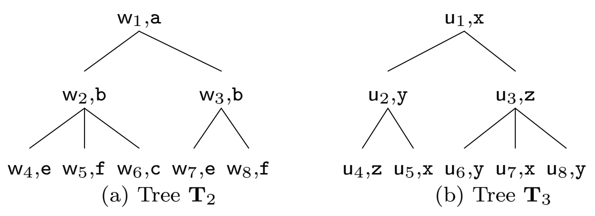


**Fig. 5** Example Trees for Comparison.

In this example we employ the Pair structure extraction technique, resulting in the following structure vectors:

- $\text{Pair}(\mathbf{T}_1) = ((\mathtt{a}), (\mathtt{a}, \mathtt{c}), (\mathtt{a}, \mathtt{b}), (\mathtt{b}, \mathtt{e}), (\mathtt{b}, \mathtt{f}), (\mathtt{a}, \mathtt{c}))$
- $\text{Pair}(\mathbf{T}_2) = ((\mathtt{a}), (\mathtt{a}, \mathtt{b}), (\mathtt{b}, \mathtt{e}), (\mathtt{b}, \mathtt{f}), (\mathtt{b}, \mathtt{c}), (\mathtt{a}, \mathtt{b}), (\mathtt{b}, \mathtt{e}), (\mathtt{b}, \mathtt{f}))$
- $\text{Pair}(\mathbf{T}_3) = ((\mathtt{x}), (\mathtt{x}, \mathtt{y}), (\mathtt{y}, \mathtt{z}), (\mathtt{y}, \mathtt{x}), (\mathtt{x}, \mathtt{z}), (\mathtt{z}, \mathtt{y}), (\mathtt{z}, \mathtt{x}), (\mathtt{z}, \mathtt{y}))$

We compressed the structure vectors using gzip after mapping each component of a vector onto a one-byte token and concatenating the tokens. As these are very small sequences, we only considered the compressed blocks in the gzip files (gzip added 28 bytes of metadata consisting of IDs, flags, the original file name, a CRC32 value, etc.)[4] The lengths of the compressed sequences and selected concatenated sequences (in bytes) are

- $C(\text{Pair}(\mathbf{T}_1)) = 8$
- $C(\text{Pair}(\mathbf{T}_2)) = 8$
- $C(\text{Pair}(\mathbf{T}_3)) = 10$
- $C(\text{Pair}(\mathbf{T}_1) \circ \text{Pair}(\mathbf{T}_1)) = 10$
- $C(\text{Pair}(\mathbf{T}_1) \circ \text{Pair}(\mathbf{T}_2)) = 13$
- $C(\text{Pair}(\mathbf{T}_1) \circ \text{Pair}(\mathbf{T}_3)) = 16$

[2] Given an arbitrary object $x$ and distance $d$, limiting the number of objects $y$ that are at a distance $d$ from $x$ is an important property of an admissible distance.

[3] More precisely, Li and Vitányi show that $|K(x, y) - K(x) - K(y|x)| = \Omega(\log(K(x)))$.

[4] For more details, see the gzip file format specification: `http://www.gzip.org/zlib/rfc-gzip.html`.

Inserting this into Equation (2) yields

– $dist_{NCD}(\text{Pair}(\mathbf{T}_1), \text{Pair}(\mathbf{T}_1)) = 0.25$
– $dist_{NCD}(\text{Pair}(\mathbf{T}_1), \text{Pair}(\mathbf{T}_2)) = 0.625$
– $dist_{NCD}(\text{Pair}(\mathbf{T}_1), \text{Pair}(\mathbf{T}_3)) = 0.8$

## 5.2 Shannon Entropy and Similarity

In his seminal paper [45] Shannon proposed to measure information content based on probability distributions, i.e., how much information is gained by witnessing a certain event (e.g., receiving a certain message). In our approach we interpret a semistructured document as a data source, emitting one component of its structure vector after another. When comparing two documents, we use the concept of relative entropy [21] to measure their distance.

**Definition 12 (Shannon Entropy)** The uncertainty of a discrete random variable $X$ can be measured using the concept of entropy. The *entropy* $H(X)$ of $X$ is defined as

$$H(X) = -\sum_{x \in \mathcal{X}} p(x) \log p(x) \tag{3}$$

where $p(x)$ is the probability mass function of $X$ and $\mathcal{X}$ is the domain of $X$. The entropy of a random variable can be interpreted as the average amount of information (in bits) needed to describe the variable.

### *5.2.1 Relative Entropy*

Entropy can be used to measure the difference between two probability distributions, using the concept of relative entropy.

**Definition 13 (Kullback-Leibler Distance)** Given two probability distributions $p$ and $q$, $D(p||q)$ is called the *Kullback-Leibler distance* (or *relative entropy*) between $p$ and $q$:[5]

$$D(p||q) = \sum_{x \in \mathcal{X}} p(x) \log \frac{p(x)}{q(x)} \tag{4}$$

The relative entropy measures the inefficiency introduced by assuming that a random variable follows the distribution $q$, while in reality it is distributed according to $p$. The inefficiency is the overhead for encoding the information generated by $X$ with a non-optimal code. If we knew the true distribution, we could create a code that uses $H(p)$ bits per symbol $x \in \mathcal{X}$ on average. Using distribution $q$ instead, we would need $H(p) + D(p||q)$ bits. $D(p||q)$ is not a distance in the metric sense, as it is not symmetric and does not satisfy the triangle inequality.

Unfortunately, computing the relative entropy between two data sources is computationally prohibitive or even impossible, depending on the distributions that the two data sources are modeled on and our knowledge of the distributions. Furthermore, if there is an $x \in \mathcal{X}$ for which $p(x) > 0$ and $q(x) = 0$ the Kullback-Leibler distance would be equal to $\infty$ [34] (even though $p$ and $q$ might be very similar for all other values of $x$). This makes sense for encoding symbols, as in this case the non-optimal code would not be able to encode the information generated by $X$ at all. For comparing documents the direct application of relative entropy is not appropriate since a document lacking a certain element found in another document could lead to a distance of $\infty$ between these documents (despite being very similar otherwise).

Ziv and Merhav developed an efficient method to discriminate between two Markov processes [58] that avoids the above issues. They assume that two realizations of $p$ and $q$ are given in the form of the emitted sequences $\mathbf{p} = (p_1, p_2, \ldots, p_n)$ and $\mathbf{q} = (q_1, q_2, \ldots, q_n)$. They show that the relative entropy $D(p||q)$ can be estimated with the help of a function that uses crossparsing.

### *5.2.2 Crossparsing*

Crossparsing a word $\mathbf{x}$ with respect to word $\mathbf{y}$ works as follows. First we find the longest prefix of $\mathbf{x}$ that appears as a string in $\mathbf{y}$, i.e., find the largest integer $m$ such that $x_1, x_2, \ldots, x_m = y_i, y_{i+1}, \ldots, y_{i+m-1}$ for some $i$. $x_1, x_2, \ldots, x_m$ is the first phrase of $\mathbf{x}$ with respect to $\mathbf{y}$ (if $x_1 \notin \mathbf{y}$, then we set $m = 1$). After determining the first value for $m$, we find the longest prefix of $\mathbf{x}$ starting with $x_{m+1}$ with respect to $\mathbf{y}$. We continue doing so until we have parsed the whole document $\mathbf{x}$. Effectively, we count the number of times we have to (re-)start the parsing.

Let $c(\mathbf{x}|\mathbf{y})$ be equal to the number of phrases we get when parsing $\mathbf{x}$ with respect to $\mathbf{y}$. For example, let $\mathbf{x} =$ *abbbbcaabba* and $\mathbf{y} =$ *baababaabba*. Then $c(\mathbf{x}|\mathbf{y}) = 4$, the four phrases being *abb*, *bb*, *c*, and *aabba*.

**Definition 14 (Crossparsing)** Let $\mathbf{x} = x_1, x_2, \ldots, x_j$ and $\mathbf{y} = y_1, y_2, \ldots, y_k$ be two (binary) words. Furthermore, let $s(\mathbf{x}|\mathbf{y})$ denote the set of subsequences of $\mathbf{x}$ obtained by crossparsing $\mathbf{x}$ with respect to $\mathbf{y}$. We describe a subsequence of $\mathbf{x}$ starting at position $i$ having length $l$ by $x_{i,l}$ and use the symbol $\preceq_{seq}$ for the relationship

[5] We follow the conventions that $0 \log 0 = 0$, $0 \log \frac{0}{0} = 0$, and $p \log \frac{p}{0} = \infty$.

"is a subsequence of (or equal to)", e.g., $x_{i,l} \preceq_{seq} \mathbf{x}$. For all $x_{i,l} \in s(\mathbf{x}|\mathbf{y})$ the following holds:

$$x_{i,l} \preceq_{seq} \mathbf{y} \vee l = 1$$

and

$$x_{i,l+1} \not\preceq_{seq} \mathbf{y}$$

For any two subsequences $x_{i_u,l_u}$ and $x_{i_v,l_v}$ (with $x_{i_u,l_u} \neq x_{i_v,l_v}$) the following holds: $i_u < i_v \vee i_u \geq i_v + l_v$. In addition to this, the sum of the lengths of all $x_{i_w,l_w} \in s(\mathbf{x}|\mathbf{y})$ is equal to $j$:

$$\sum_w l_w = j$$

We denote the cardinality of $s(\mathbf{x}|\mathbf{y})$ by $c(\mathbf{x}|\mathbf{y})$, i.e., $c(\mathbf{x}|\mathbf{y}) = |s(\mathbf{x}|\mathbf{y})|$.

#### *5.2.3 Crossparsing Distance (CPD)*

The original crossparsing function used by Ziv and Merhav is only defined if both sequences have the same length ($j = k$), which is not the case in our scenario. Moreover, the resulting distance measure is neither symmetric nor normalized, meaning that the order in which documents are compared plays a role and the size of the documents has a significant impact.

In Definition 14 we have generalized the definition of crossparsing [58] by allowing crossparsed sequences that have different lengths. We now develop solutions for dealing with the issues of symmetry and normalization. Let us turn to normalization first: when looking at the result of $c(\mathbf{x}|\mathbf{y})$, we realize that $1 \leq c(\mathbf{x}|\mathbf{y}) \leq |\mathbf{x}|$ holds. We get at least one phrase (if $\mathbf{x}$ is found as a substring of $\mathbf{y}$) and at most $|\mathbf{x}|$ phrases (if the largest substring of $\mathbf{x}$ ever found in $\mathbf{y}$ has at most length 1). Therefore, we normalize $c(\mathbf{x}|\mathbf{y})$ to $\frac{c(\mathbf{x}|\mathbf{y})-1}{|\mathbf{x}|}$. Next, in order to get a symmetric distance, we add the normalized terms for $c(\mathbf{x}|\mathbf{y})$ and $c(\mathbf{y}|\mathbf{x})$.[6]

**Definition 15 (Crossparsing Distance)** Given two words $\mathbf{x}$ and $\mathbf{y}$, the crossparsing distance $dist_{CPD}(\mathbf{x},\mathbf{y})$ between $\mathbf{x}$ and $\mathbf{y}$ is

$$dist_{CPD}(\mathbf{x},\mathbf{y}) = \frac{\frac{c(\mathbf{x}|\mathbf{y})-1}{|\mathbf{x}|} + \frac{c(\mathbf{y}|\mathbf{x})-1}{|\mathbf{y}|}}{2} \tag{5}$$

Note that we divide the sum by two in order to keep the distance normalized.

[6] This is a well-known trick used, for example, for obtaining a symmetric variant for conditional entropies: $H(X|Y) + H(Y|X)$ [21].

**Theorem 1 (Crossparsing is a semimetric)** *The crossparsing distance satisfies the following conditions of a metric:*

*1.* non-negativity*:* $dist_{CPD}(\mathbf{x},\mathbf{y}) \geq 0$
*2.* reflexivity*:* $\mathbf{x} = \mathbf{y} \Leftrightarrow dist_{CPD}(\mathbf{x},\mathbf{y}) = 0$
*3.* symmetry*:* $dist_{CPD}(\mathbf{x},\mathbf{y}) = dist_{CPD}(\mathbf{y},\mathbf{x})$

*The only condition not satisfied is the*

*4.* triangle inequality*:* $dist_{CPD}(\mathbf{x},\mathbf{y}) + dist_{CPD}(\mathbf{y},\mathbf{z}) \geq dist_{CPD}(\mathbf{x},\mathbf{z})$

*which makes* $dist_{CPD}(\mathbf{x},\mathbf{y})$ *a semimetric.*

*Proof*

1. Immediately follows from the fact that $c(\mathbf{x}|\mathbf{y}) \geq 1$ for all $\mathbf{x}$ and $\mathbf{y}$.
2. $\mathbf{x} = \mathbf{y}$
   $\Rightarrow c(\mathbf{x}|\mathbf{y}) = c(\mathbf{y}|\mathbf{x}) = 1$
   $\Rightarrow dist_{CPD}(\mathbf{x},\mathbf{y}) = 0$

   Since $c(\mathbf{x}|\mathbf{y}) \geq 1$ and $c(\mathbf{y}|\mathbf{x}) \geq 1$:
   $dist_{CPD}(\mathbf{x},\mathbf{y}) = 0$
   $\Rightarrow c(\mathbf{x}|\mathbf{y}) = c(\mathbf{y}|\mathbf{x}) = 1$
   $\Rightarrow \mathbf{x} \preceq_{seq} \mathbf{y} \wedge \mathbf{y} \preceq_{seq} \mathbf{x}$
   $\Rightarrow \mathbf{x} = \mathbf{y}$
3. Immediately follows from the commutative property of addition.
4. A counterexample for which $dist_{CPD}(\mathbf{x},\mathbf{y}) + dist_{CPD}(\mathbf{y},\mathbf{z}) \not\geq dist_{CPD}(\mathbf{x},\mathbf{z})$ is the following: $\mathbf{x} = 000000$, $\mathbf{y} = 000111$, and $\mathbf{z} = 111111$.

*Example 4* We apply the crossparsing distance to the trees $\mathbf{T}_1$, $\mathbf{T}_2$, and $\mathbf{T}_3$ (depicted in Figures 1 and 5, respectively). Again we use the Pair structure extraction technique, so we have the same structure vectors as in Example 3. Each component of a structure vector is considered to be a single symbol during the crossparsing process, thus we get the following results:

– $c(\text{Pair}(\mathbf{T}_1)|\,\text{Pair}(\mathbf{T}_1)) = 1$
– $c(\text{Pair}(\mathbf{T}_1)|\,\text{Pair}(\mathbf{T}_2)) = 4$
– $c(\text{Pair}(\mathbf{T}_2)|\,\text{Pair}(\mathbf{T}_1)) = 4$
– $c(\text{Pair}(\mathbf{T}_1)|\,\text{Pair}(\mathbf{T}_3)) = 6$
– $c(\text{Pair}(\mathbf{T}_3)|\,\text{Pair}(\mathbf{T}_1)) = 8$

Applying Equation (5) yields:

– $dist_{CPD}(\text{Pair}(\mathbf{T}_1), \text{Pair}(\mathbf{T}_1)) = 0$
– $dist_{CPD}(\text{Pair}(\mathbf{T}_1), \text{Pair}(\mathbf{T}_2)) = 0.4375$
– $dist_{CPD}(\text{Pair}(\mathbf{T}_1), \text{Pair}(\mathbf{T}_3)) \approx 0.8542$

Compared to the results obtained for the compression distance in Example 3, we notice that distance between identical trees is zero. Further, the values for

CPD are more spread out. This indicates that the crossparsing distance is better at distinguishing documents. We give a more detailed explanation in Section 8.3.8 after presenting the results of our experimental evaluation.

## 5.3 Kolmogorov Complexity vs. Shannon Entropy

At first glance it seems as if Kolmogorov complexity and Shannon entropy measure the information content of an object differently. The latter is about describing the random process that outputs data objects, but does not consider the concrete shape of these objects. The former, on the other hand, concerns itself with the shapes of individual objects, but treats them in isolation.

A closer look, however, reveals that the two concepts are closely related to each other [27]. Assume that we have a source $X$ emitting binary words of length $n$ according to a probability mass function $p(x)$. Then the entropy of this source is asymptotically equal to the expected Kolmogorov complexity of these words. Our aim is to find out which of the similarity measures derived from these information-theoretic concepts is more suitable for measuring structural similarity.

## 6 Algorithms

In this section we describe the implementation of the algorithms for computing the different structure vectors and the distance between them using compression and crossparsing.

### 6.1 Computing the Structure Vector

In our algorithms we use the following notation: Let $x = (x_1, x_2, \ldots, x_n)$ and $y = (y_1, y_2, \ldots, y_m)$ be two sequences, then $x \circ y = (x_1, x_2, \ldots, x_n, y_1, y_2, \ldots, y_m)$ is the concatenation of $x$ and $y$. Furthermore, assuming that $()$ is the empty sequence, $x \circ () = () \circ x = x$. Nesting the sequence $x$, $(x)$, results in a sequence that contains the sequence $x$ as its only element. The shift operation, $\text{shift}(x, x_i) = (x_2, x_3, \ldots, x_n, x_i)$, removes the first element of $x$ and adds $x_i$ as a new element at the end.

#### *6.1.1 Doc, Doc+, Pair, Path*

The Doc, Doc+, Pair, and Path structure vectors are computed in a single preorder traversal of the tree, thus the implementation in a standard XML parser is straightforward. Algorithm 6.1, which is called with the parameters $\mathsf{r} = \text{root}(\mathbf{T})$, $V = ()$, and $path = ()$, computes the Path structure vector $V$ of tree $\mathbf{T}$. The algorithm can easily be adapted to compute the Doc or Pair structure vectors by adding only the last or the last two elements of the path to the structure vector in Line 2, respectively.

**Algorithm 6.1**: `Path`($\mathsf{r}$, $V$, $path$)

```
1 $path \leftarrow path \circ (\text{l}(\mathsf{r}))$;
2 $V \leftarrow V \circ (path)$;
3 foreach child $\mathsf{c}$ (from left to right) of $\mathsf{r}$ do
4     $V \leftarrow$ Path($\mathsf{c}$, $V$, path);
5 return $V$;
```

In case of a Doc+ structure vector we also have to add end tags. So immediately before returning the structure vector $V$ in line 5, we append the closing tag of $r$ to $V$.

#### *6.1.2 Family*

Algorithm 6.2, called with $\mathsf{r} = \text{root}(\mathbf{T})$ and $V = ()$, recursively computes the Family structure vector of tree $\mathbf{T}$. The nodes are traversed in postorder, and for each non-leaf node all its children are appended to the structure vector ($\text{ch}(\mathsf{v}) = ()$ if $\mathsf{v}$ is a leaf); finally the root node is appended.

**Algorithm 6.2**: `Family`($\mathsf{r}$, $V$)

```
1 foreach child $\mathsf{c}$ (from left to right) of $\mathsf{r}$ do
2     $V \leftarrow$ Family($\mathsf{c}$, $V$);
3 $V \leftarrow V \circ (\text{l}(\text{ch}(\mathsf{r})))$;
4 if $\mathsf{r}$ is the root node then
5     $V \leftarrow V \circ ((\text{l}(\mathsf{r})))$;
6 return $V$;
```

#### *6.1.3 pq-Gram*

The `pqGram` algorithm (see Algorithm 6.3) moves the *pq*-gram pattern vertically and horizontally over the tree. After each move, the nodes covered by the pattern form a *pq*-gram.

The two sequences *anc* of length $p$ and *sib* of length $q$ represent the labels of the ancestor and the leaf nodes that are covered by the *pq*-gram pattern, respectively. They are implemented as shift registers to efficiently support the shift operation. The concatenation of the two registers, $anc \circ sib$, forms one component of the *pq*-gram structure vector. Algorithm 6.3, initially called with the parameters $\mathsf{r} = \text{root}(\mathbf{T})$, $V = ()$, and the shift

**Algorithm 6.3**: $\texttt{pqGram}(\mathsf{r}, V, anc)$

```
sib: shift register of size q (filled with *);
anc ← shift(anc, l(r));
if r is a leaf then
    $V \leftarrow V \circ (anc \circ sib)$;
else
    foreach child c (from left to right) of r do
        sib ← shift(sib, l(c));
        $V \leftarrow V \circ (anc \circ sib)$;
        $V \leftarrow$ pqGram(c, V, anc);
    for $k \leftarrow 1$ to $q - 1$ do
        sib ← shift(sib, *);
        $V \leftarrow V \circ (anc \circ sib)$;
return V;
```

register *anc* filled with $p$ dummy labels (*), computes the *pq*-gram structure vector of tree $\mathbf{T}$.

### *6.1.4 Complexity*

The algorithms for Doc, Doc+, Pair, Family, and *pq*-grams run in $O(n)$ time and space for a tree $\mathbf{T}$ with $n = |\mathbf{T}|$ nodes: each recursive call processes one node, and each node is processed exactly once. The size of the output (the structure vector $V$) is linear in $n$. `Path` also processes each node exactly once, but in the worst case in each step a path of size $O(n)$ is added to the structure vector, resulting in an $O(n^2)$ complexity.

## 6.2 Computing the Similarity

Instead of using individual characters of node labels for comparison, we map them to 32-bit codes using a hash table. For example, the Path structure vector ((a),(a, c),(a, b), (a, b, e),(a, b, f), (a, c)) results in the following sequence of code words: $c_1$ $c_2$ $c_3$ $c_4$ $c_5$ $c_2$. This sequence is processed by our similarity measure algorithms as a string. Therefore, we refer to these sequences as strings in the remainder of this section.

### *6.2.1 Compression Distance*

Selfparsing a string is an important part of the Ziv-Lempel family of encoders [55, 57] (we were using gzip in our experiments). These algorithms build a dictionary for encoding (and decoding) by utilizing substrings that have already appeared in the previously encoded part of the document. For that purpose a Ziv-Lempel encoder stores *distances* and *lengths* in the dictionary. *Distance* refers to the starting position of the referenced substring as measured by going backwards from the current symbol (the first symbol of the rest of the document that is yet to be compressed), while *length* indicates the length of the substring. The algorithm always tries to find the longest possible substring that matches a string starting at the current symbol. Added to the *distance-length* information is the first symbol that does not match the previously encoded substring. So the dictionary consists of triplets of the form <`distance, length, symbol`> and is generated on the fly during encoding (and decoding). See Figure 6 for an example. For the compression-based distance this means that when concatenating two documents, the second document can fall back on substrings of the first document. The more overlap there is between two documents, the better the compression rate will be.

String to encode:
aababbccccd

Triplets:

| | |
|---|---|
| <0,0,a> | first appearance of *a*, no previous substring available |
| <1,1,b> | *ab* is encoded with a reference to *a* followed by the symbol *b* |
| <2,2,b> | *abb* is encoded with a reference to the previous *ab* followed by *b* |
| <0,0,c> | first appearance of *c* |
| <1,3,d> | the longest matching substring previously encoded and the string to be compressed may overlap |

**Fig. 6** Example for Ziv-Lempel.

### *6.2.2 A Linear Crossparsing Algorithm*

Crucial for the efficiency of the crossparsing algorithm is a *suffix tree* [54] that stores all the different suffixes of a string making it possible to look up substrings of a string in linear time. Most important for our purposes is the algorithm by Ukkonen [52], which constructs a suffix tree for a given string in linear time (i.e., $O(|s|)$, where $|s|$ is the length of the string). In addition to the linear run-time, this is an online algorithm, which means that the suffix tree is constructed on the fly while parsing the string. When going from symbol $\sigma_i$ to $\sigma_{i+1}$ in a string $s$ during parsing, all the suffixes for the string from $\sigma_1$ to $\sigma_i$ already stored in the tree are extended by $\sigma_{i+1}$.

Algorithm 6.4 illustrates how we can use a suffix tree to crossparse strings in linear time. This technique has been applied by Martins [38], but no proof for the complexity of the algorithm has been provided.

**Theorem 2** *Crossparsing a string* $\mathbf{x}$ *with respect to* $\mathbf{y}$ *is linear in the sum of the lengths of the strings* $\mathbf{x}$ *and* $\mathbf{y}$*:* $O(|\mathbf{x}| + |\mathbf{y}|)$.

**Algorithm 6.4**: `crossparse`($\mathbf{x}$, $\mathbf{y}$)

```
1 $ST_{\mathbf{y}} \leftarrow$ buildSuffixTree($\mathbf{y}$);
2 $i \leftarrow 1$;
3 $c \leftarrow 0$;
4 repeat
5     find largest $j$ for which $\mathbf{x}[i..i+j]$ can be found in $ST_{\mathbf{y}}$;
6     $c \leftarrow c + 1$;
7     $i \leftarrow i + j + 1$;
8 until $i > |\mathbf{x}|$ ;
9 return $c$;
```

*Proof* Building $ST_y$ is in the order of $O(|\mathbf{y}|)$ [52]. Determining the largest $j$ is done by traversing the suffix tree symbol by symbol until we hit a dead end. Therefore the complexity of a lookup is $O(|j+1|)$. We do this $c(\mathbf{x}|\mathbf{y})$ times (remember that $c(\mathbf{x}|\mathbf{y})$ is the number of subsequences of $\mathbf{x}$ obtained by crossparsing $\mathbf{x}$ with respect to $\mathbf{y}$). Thus the total time spent for looking up subsequences of $\mathbf{x}$ in $ST_y$ is $\sum_{k=1}^{c(\mathbf{x}|\mathbf{y})}(j_k+1) = O(|\mathbf{x}|)$.

## 7 The Competitors

In this section we compare our information-theoretic approaches to five other techniques: the tree edit distance (TED) by Nierman and Jagadish [41], the Discrete Fourier Transformation (DFT) by Flesca et al. [25], path shingles (PS) by Buttler [13], propagation vectors for trees (PVT) by Cherukuri and Candan [18], and the Dice coefficient applied to the structure vectors.

### 7.1 Tree Edit Distance (TED)

The tree edit distance between two trees is the minimum number of edit operations that transform one tree into the other. The standard edit operations are node deletion, node insertion, and node relabeling. Due to optional and repeating elements, XML documents generated from the same DTD or schema can exhibit a considerable difference in their sizes (i.e., the number of nodes they contain). Merely allowing edit operations that change one node at a time would lead to very large distances in these cases. Thus, Nierman and Jagadish [41] develop a set of edit operations that reflect the specific needs of XML documents and allow subtree deletion and insertion:

– *Relabel:* change the label of a node
– *Insert:* insert a new leaf node
– *Delete:* delete a leaf node
– *Insert Subtree:* attach a new subtree to a leaf node
– *Delete Subtree:* delete a node and all its descendants

There are two constraints on subtree insertion and deletion: (1) a subtree may only be inserted if it already occurs in the tree, and it may only be deleted if it still occurs in the tree; (2) no new nodes can be inserted into inserted subtrees, and no nodes can be deleted from subtrees before deleting them. Without these restrictions, any tree could be transformed into any other tree in just two steps by deleting the original tree and inserting the new tree. The complexity of the algorithm is $O(|\mathbf{T}_1||\mathbf{T}_2|)$.

### 7.2 Discrete Fourier Transformation (DFT)

Flesca et al. [25] represent an XML document with $n$ nodes as a sequence of $2n$ opening and closing tags in the order of appearance in the document, $d = (t_1, t_2, \ldots, t_{2n})$. Intuitively, if we assume that the tags in a document are indented and we rotate the document by 90 degrees, then we can interpret this as a time series by mapping the indentations to numerical values.

The indentation value for a tag is computed in two steps. In the first step, called the *tag encoding*, each tag $t$ is mapped to a integer value, $\gamma(t)$. Flesca et al. distinguish direct, pairwise, and nested encoding. The direct encoding maps each tag to a value that depends only on the tag name and ignores the context; the pairwise encoding considers also the name of the parent element; the nested encoding takes into account all ancestors.

In the second step, the *document encoding*, the tag numbers of a document $d$ are composed into a sequence of $2n$ real numbers, $enc(d)$, that represents the document. Flesca et al. distinguish trivial, linear, and multi-level encodings. The trivial encoding is the sequence of all tag encodings, $enc(d) = (\gamma(t_1), \gamma(t_2), \ldots, \gamma(t_{2n}))$; in the linear encoding each element of the sequence is a linear combination of all the preceding elements, $enc(d) = (\gamma(t_1), \gamma(t_1) + \gamma(t_2), \ldots, \sum_{i=1}^{2n}\gamma(t_i))$; in the multi-level encoding a tag encoding is combined with the tag encodings of all its ancestors in the document, $enc(d) = (S_1, S_2, \ldots, S_{2n})$,

$$S_i = \gamma(t_i) \cdot B^{maxlevel-level(t_i)} + \sum_{j=1}^{level(t_i)} \gamma(anc(j,t_i)) \cdot B^{maxlevel-level(anc(j,t_i))},$$

where $anc(j,t_i) = anc(j,\mathsf{v})$ and $level(t_i) = level(\mathsf{v})$ for a tag $t_i$ of an element node $\mathsf{v}$, $maxlevel$ is the deepest level of a tag in the document collection, and $B$ is a constant (at least the maximum of $|\gamma(t_i)|+1$ for any tag $t_i$, thus distinguishing tag and document encoding).

*Example 5* Consider the following direct tag encoding: $\gamma($`<a>`$) = 1$, $\gamma($`</a>`$) = -1$, $\gamma($`<b>`$) = 2$, $\gamma($`</b>`$) = -2$,

$\gamma($`<c>`$) = 3$, $\gamma($`</c>`$) = -3$, $\gamma($`<e>`$) = 4$, $\gamma($`</e>`$) = -4$, $\gamma($`<f>`$) = 5$, and $\gamma($`</f>`$) = -5$. Then the linear encoding of the example document $d =$ (`<a>`, `<c>`, `</c>`, `<b>`, `<e>`, `</e>`, `<f>`, `</f>`, `</b>`, `<c>`, `</c>`, `</a>`) in Figure 1 is $enc(d) = (1, 4, 1, 3, 7, 3, 8, 3, 1, 4, 1, 0)$.

Flesca et al. use the Discrete Fourier Transformation (DFT) to compare the encodings of two documents. DFT is used to abstract from the length of the documents and determine whether a given subsequence of tags appears with a certain regularity (independent of the location of the subsequence). The distance between two documents, $d_1$ and $d_2$, is

$$dist_{DFT}(d_1, d_2) = \left( \sum_{k=1}^{\frac{M}{2}} \left( |[\tilde{\text{DFT}}(enc(d_1))](k)| - |[\tilde{\text{DFT}}(enc(d_2))](k)| \right)^2 \right)^{\frac{1}{2}}$$

where $\tilde{\text{DFT}}$ is an interpolation of DFT to the frequencies appearing in both $d_1$ and $d_2$, and $M$ is the total number of points appearing in this interpolation, i.e., $M = |\{t_i \mid t_i \in d_1 \vee t_i \in d_2\}|$. The complexity of the algorithm is $O(N \log N)$ with $N = \max\{|d_1|, |d_2|\}$.

## 7.3 Path Shingles (PS)

Path shingles by Buttler [13] use path vectors as defined in Section 4.4. The $j$-th element of the path vector is hashed to an integer $h_j$. A shingle is a sequence of $w$ consecutive hash values, $(h_j, h_{j+1}, \ldots, h_{j+w})$, of the path vector, where $w \geq 1$ is the window size. The set of all shingles of width $w$ for a document tree $\mathbf{T}_i$ is denoted by $S(\mathbf{T}_i, w)$. The similarity of two document trees is the normalized intersection between their path shingle sets (Dice coefficient):

$$dist_{PS}(\mathbf{T}_i, \mathbf{T}_k) = 1 - \frac{S(\mathbf{T}_i, w) \cap S(\mathbf{T}_k, w)}{S(\mathbf{T}_i, w) \cup S(\mathbf{T}_k, w)}$$

## 7.4 Propagation Vectors for Trees (PVT)

Cherukuri and Candan [18] use concept vectors and the propagation of their component values to other vectors as a basis for measuring structural similarity. Originally, this technique was developed to measure the similarities between concept nodes in hierarchical taxonomies [31].

Each node of a tree $\mathbf{T}$ has a concept vector. The concept vector of the $i$-th node of $\mathbf{T}$ in preorder is $C_i = (\omega_{i,1}, \omega_{i,2}, \ldots, \omega_{i,n})$, $n = |\mathbf{T}|$, where $\omega_{i,j}$ is the similarity between node $i$ and node $j$ and is initialized as follows: $\omega_{i,j} = 1$ if $i = j$ and $\omega_{i,j} = 0$ otherwise.

During the propagation phase the concept vectors are iteratively updated (Kim and Candan [31] suggest $h$ iterations for a tree of height $h$). In each iteration the values of the concept vectors are propagated from a parent $p$ (concept vector $C_p$) to its children ($c_1, c_2, \ldots, c_k$, concept vectors $C_{c_i}$) and vice versa. The similarities $w_{p,j}$ and $w_{c_i,j}$ of the concept vectors $C_p$ and $C_{c_i}$ are updated as follows:

$$w'_{p,j} = w_{p,j} + \sum_{i=1}^{k} (\alpha_{c_i \to p} \times w_{c_i,j})$$
$$w'_{c_i,j} = w_{c_i,j} + \alpha_{p \to c_i} \times w_{p,j}$$

The propagation degrees $\alpha_{c_i \to p}$ and $\alpha_{p \to c_i}$ are constants that control the impact of the nodes on each other.

The concept vector of a node expresses its similarity to the concepts represented by the other nodes of the same tree. In order to compute the distance between two different trees, the concept vectors of their root nodes are compared. The concept vectors of the two trees, $\mathbf{T}_1$ and $\mathbf{T}_2$, which may have different dimensions, are mapped to $m$-dimensional vectors, the *concept vector summaries* $V_1$ and $V_2$, where $m = |\{\lambda \mid \lambda = \mathrm{l}(\mathsf{v}), \mathsf{v} \in V(\mathbf{T}_1) \cup V(\mathbf{T}_2)\}|$ is the number of labels in both trees. Each dimension $l$ of the vectors $V_i$ stands for a different label $\lambda_l$ that (possibly repeating) appears in $\mathbf{T}_1$ and/or $\mathbf{T}_2$. Let set $S_{i,l}$ be the elements of the root concept vector of tree $\mathbf{T}_i$ that represent a node with label $\lambda_l$. The elements of the concept vector summaries $V_i$ are computed as $v_{i,l} = \sqrt{\sum_{\omega \in S_{i,l}} \omega^2}$.

*Example 6* Consider tree $\mathbf{T}_1$ in Figure 1. The concept vector of $\mathsf{v}_1$ is initialized as $C_1 = (1, 0, 0, 0, 0, 0)$. The parent-child propagation (all propagation degrees are assumed to be $\frac{1}{3}$) results in the concept vectors $C_2 = (\frac{1}{3}, 1, 0, 0, 0, 0)$ for $\mathsf{v}_2$, $C_3 = (\frac{1}{3}, 0, 1, 0, 0, 0)$ for $\mathsf{v}_3$, and $C_6 = (\frac{1}{3}, 0, 0, 0, 0, 1)$ for $\mathsf{v}_4$. The child-parent propagation gives $C_1 = (1\frac{1}{3}, \frac{1}{3}, \frac{1}{3}, 0, 0, \frac{1}{3})$. Let $\lambda_1 = \mathtt{a}$, $\lambda_2 = \mathtt{c}$, $\lambda_3 = \mathtt{b}$, $\lambda_4 = \mathtt{e}$, $\lambda_5 = \mathtt{f}$, then $V_1 = (1\frac{1}{3}, \frac{\sqrt{2}}{3}, \frac{1}{3}, 0, 0)$.

Cherukuri and Candan propose to use one of the following distances between two concept vector summaries, $V_1$ and $V_2$ (with $|V_1| = |V_2| = m$): the cosine distance, $\text{dist}_{cos}(V_1, V_2) = \sum_{l=1}^{m} (v_{1,l} v_{2,l})/m^2$; the intersection distance, $\text{dist}_{is}(V_1, V_2) = \sum_{l=1}^{m} \min\{v_{1,l} v_{2,l}\}/\sum_{l=1}^{m} \max\{v_{1,l} v_{2,l}\}$; an adaption of the Kullback-Leibler distance (4) to numeric vectors, which interprets the vectors $V_1$ and $V_2$ as probability distributions:

$$\mathrm{dist}_{KL}(V_1, V_2) = \frac{D(V_1||V_2) + D(V_2||V_1)}{2} = \frac{1}{2}\sum_{l=1}^{m}\left(v_{1,l}\,\log\frac{v_{1,l}}{v_{2,l}} + v_{2,l}\,\log\frac{v_{2,l}}{v_{1,l}}\right)$$

As mentioned in Section 5.2 there is a problematic case if either $v_{1_i}$ or $v_{2_i}$ is equal to 0, and $\mathrm{dist}_{KL}(V_1, V_2)$ is undefined. It appears that this will never be the case for summary vectors, but this is not addressed explicitly [18]. Also note that the concept vector summaries are $m$-dimensional vectors of real numbers, whereas the elements of our structure vectors are categorical values that represent node sequences and cannot be interpreted as probabilities. Our crossparsing distance in Section 5.2 adapts the Kullback-Leibler distance to structure vectors.

In order to create a summary vector of a document $O(dn)$ exchanges of $m$-dimensional vectors between parent and child node pairs are necessary, where $d$ is the depth of the document. The actual comparison of the summary vectors can be done in linear time, so the total complexity is $O(m \times \max\{\mathrm{depth}(\mathbf{T}_1)|\mathbf{T}_1|, \mathrm{depth}(\mathbf{T}_2)|\mathbf{T}_2|\})$.

### 7.5 Dice Coefficient

The dice coefficient is the normalized intersection between two sets and measures their similarity. We define the Dice distance between two structure vectors $V(\mathbf{T}_1)$ and $V(\mathbf{T}_2)$ as $\mathrm{dist}_{Dice}(V(\mathbf{T}_1), V(\mathbf{T}_2)) = 1 - 2|S_1 \Cap S_2|/|S_1 \uplus S_2|$, where $S_1$ and $S_2$ are the multisets of all elements of $V(\mathbf{T}_1)$ and $V(\mathbf{T}_2)$, respectively. Combining the dice distance with *pq*-gram vectors represents the *pq*-gram distance as originally defined by Augsten et al. [5].

## 8 Experimental Evaluation

In our experiments we demonstrate the effectiveness of combining the *pq*-gram approach with information-theoretic similarity measures. We evaluate all combinations of structure extraction techniques (cf. Section 4) with the information theoretic approaches (cf. Section 5) and compare them to five competing approaches found in the literature (cf. Section 7). The remainder of this section is organized as follows: first we discuss the experimental setup, then we present the data sets used in the experiments, and finally we discuss our effectiveness and efficiency results for each of the methods.

### 8.1 Experimental Setup

We use our structure-extraction/information-theoretic approaches and five competing distance measures to cluster collections of XML documents adhering to different DTDs or schemas. The algorithms are not given any hint which DTD or schema a document is associated with (i.e., we removed all respective information). After clustering the documents we count the number of documents that are misclustered, i.e., documents that were put into the wrong DTD/schema cluster.

We use the well-known hierarchical agglomerative clustering [30] to cluster a set of XML documents. Initially, each document represents its own cluster. In each subsequent step the two clusters that are closest are merged until there is only a single cluster left. The distance between two clusters, $c_i$ and $c_j$, is computed using the Unweighted Pair Group Method with Arithmetic mean (UPGMA) [47]: $dist(c_i, c_j) = \frac{1}{|c_i||c_j|}\sum_{d_i \in c_i}\sum_{d_j \in c_j} dist(d_i, d_j)$, where $d_i$ and $d_j$ are documents. The same clustering approach was taken by Nierman and Jagadish [41], which makes our results directly comparable.

The output of the agglomerative clustering is a dendrogram (see Figure 7 for an example). Intuitively, a dendrogram is a binary tree that shows in which order the document clusters were merged. The leaf nodes of a dendrogram are clusters that consist of a single document, the inner nodes represent clusters that consist of all the documents in all their leaf descendants.

**Definition 16 (Dendrogram)** A *dendrogram* over a document set $A = \{d_1, d_2, \ldots, d_n\}$ is a full binary tree with $|A|$ leaf nodes and $|A| - 1$ inner nodes that are labeled with mutually different subsets of $A$ as follows: (a) all leaf nodes $\mathsf{v}_i$ are labeled with a single-document subset, $\mathsf{l}(\mathsf{v}_i) = \{d_k\}, d_k \in A$; (b) all inner nodes are labeled with the union of the labels of their leaf descendants.

A $k$-clustering $C = \{c_1, c_2, \ldots, c_k\}$ of a document set $A$ partitions $A$ into $k$ non-overlapping subsets, i.e., $\bigcup_{i=1}^{k} c_i = A$, and for all pairs $(c_i, c_j) \in C \times C, c_i \neq c_j : c_i \cap c_j = \emptyset$. Given a correct $k$-clustering[7] and any other $k$-clustering of the same dataset, the cluster error measures the minimum number of misclustered documents between the two clusterings.

**Definition 17 (Cluster Error)** Let $P = \{p_1, p_2, \ldots, p_k\}$ be the correct $k$-clustering of a dataset $A$, $C = \{c_1, c_2, \ldots, c_k\}$ be any $k$-clustering of $A$, $\pi^k$ the set of all

[7] In a correct (or perfect) clustering all elements of each subset $c_i$ belong to the same category and every $c_i$ represents a different category.

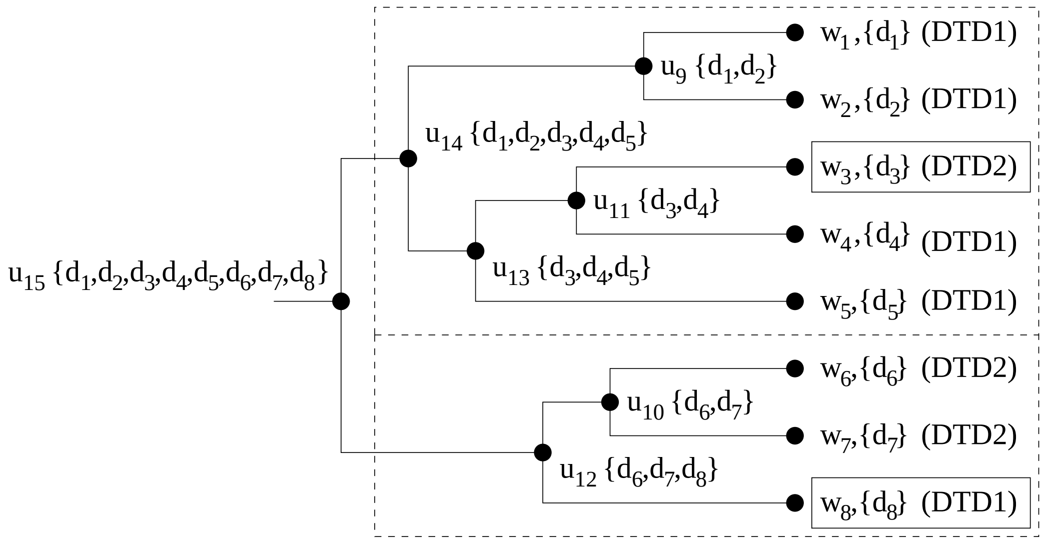


**Fig. 7** Example Dendrogram.

permutations over $(1, 2, \dots, k)$, $\pi_i$ the $i$-th element of $\pi \in \pi^k$. The *cluster error* of $C$ with respect to $P$ is

$$\epsilon(C, P) = \min_{\pi \in \pi^k} \{e \,|e = \sum_{i=1}^{k} |p_i \setminus c_{\pi_i}|\}$$

In our experiments, the correct clustering $P$ is such that two documents are in a cluster iff they were produced from the same DTD or schema. The $k$-clustering $C$ is derived from the dendrogram computed with one of the algorithms (either using our approach or one of the competitors). The labels of a dendrogram over $A$ uniquely define a $k$-clustering for each value of $k$, $1 \leq k \leq |A|$. For example, $C = \{\mathrm{l}(u_{12}), \mathrm{l}(u_{14})\}$ is the 2-clustering defined by the dendrogram in Figure 7. A small cluster error means that the distance effectively distinguishes between documents that where produced from a different DTD or schema.

## 8.2 The Benchmark Data Sets

In our experiments we used different data sets consisting of real and synthetically generated documents. For the real data sets we used 57 documents from the XML version of the SIGMOD record[8], 60 documents from the heterogeneous track at INEX 2005[9], and 34 music sheets encoded in XML[10]. These three data sets are called SIGMOD, INEX, and Music in the following.

[8] available at http://www.sigmod.org/publications/sigmod-record/xml-edition

[9] available at http://inex.is.informatik.uni-duisburg.de/2005/

[10] available at http://xml.coverpages.org/xmlMusic.html

For the synthetically generated documents (generated with ToXgene[11]) we started out with the DTDs in the DFT paper [25] (we call these data sets DFT1 and DFT2). As we wanted to further investigate individual parameters, we also created our own collection, in which we varied one parameter per document collection and kept all other parameters the same throughout one collection. Each document collection contained eight different clusters and for each cluster we generated ten documents.

The synthetically generated document collections are quite extreme, as there are often only nuances between different clusters. We (as well as Flesca et al. [25]) chose this setting to test out the limits of the different approaches. The DTDs used by Flesca et al. can be found in Appendix A, our collection is described in the following.

### *8.2.1 Element Data Set (Elem)*

In our first data set, all documents have exactly the same structure and vary only in the element names. Each cluster has a unique set of element names, i.e., no two clusters share any names. This data set tests the algorithms on their ability to distinguish between different tags. Ideally, we expect that documents with different tags end up in different clusters, despite documents having exactly the same tree structure. Figure 8 shows the basic structure of the documents we used for this data set (see Appendix B for a more formal definition in XML Schema).

[11] available at http://www.cs.toronto.edu/tox/toxgene/

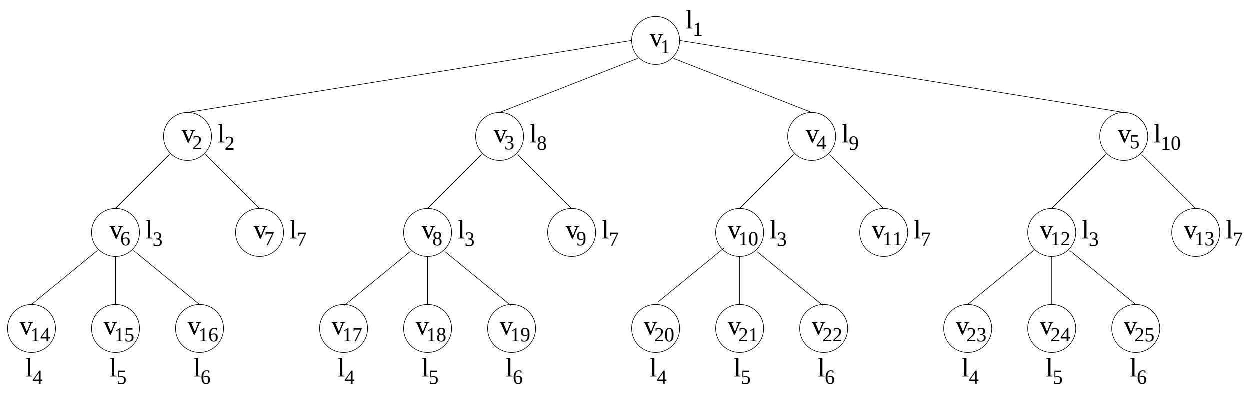


**Fig. 8** Basic Document Used for Experiments.

Figure 8 also shows the basic labeling scheme we used for this document collection: within each document, the following sets of nodes have the same element names: $\{v_6, v_8, v_{10}, v_{12}\}$, $\{v_7, v_9, v_{11}, v_{13}\}$, $\{v_{14}, v_{17}, v_{20}, v_{23}\}$, $\{v_{15}, v_{18}, v_{21}, v_{24}\}$, and $\{v_{16}, v_{19}, v_{22}, v_{25}\}$. Let $f_i$ denote the frequency of appearance for a node $v_i$ (and its subtree). We set the frequency for each node to 1-4 (i.e., each node appears at least once and up to four times), except for the root node, whose frequency $f_1$ is set to 1.

### *8.2.2 Frequency Data Set (Freq)*

With the next dataset we evaluate how well the different approaches can cope with varying frequencies. Again we used the same basic structure as before (see Figure 8). However, this time all documents used the same set of element names (with certain node sets sharing the same element names, as above for the Element data set). The only difference between documents belonging to different clusters was the number of occurrences of each element. Ideally, we expect an algorithm to distinguish between different element frequencies in documents and assign them to different clusters. The frequency $f_1$ was always set to 1; for the other frequency values, see Table 1.

| cluster | $f_i$s | freq |
|---|---|---|
| 1 | $f_2$ - $f_{25}$ | 1 |
| 2 | $f_2$ - $f_{25}$ | 3-4 |
| 3 | $f_2$ - $f_{13}$ | 1 |
| | $f_{14}$ - $f_{25}$ | 3-4 |
| 4 | $f_2$ - $f_5$ | 3-4 |
| | $f_6$ - $f_{25}$ | 1 |
| 5 | $f_3, f_5, f_7, f_9, f_{11}, f_{13}, f_{16}, f_{19}, f_{22}, f_{25}$ | 3-4 |
| | all others | 1 |
| 6 | $f_3, f_5, f_7, f_9, f_{11}, f_{13}, f_{16}, f_{19}, f_{22}, f_{25}$ | 1 |
| | all others | 3-4 |
| 7 | $f_2, f_4, f_7, f_8, f_{11}, f_{12}, f_{14}, f_{15}, f_{19}, f_{20} f_{21}, f_{25}$ | 1 |
| | all others | 3-4 |
| 8 | $f_2, f_4, f_7, f_8, f_{11}, f_{12}, f_{14}, f_{15}, f_{19}, f_{20} f_{21}, f_{25}$ | 3-4 |
| | all others | 1 |

**Table 1** Element Frequencies for the Frequency Data Set.

### *8.2.3 Position Data Set (Pos)*

Our third document collection evaluates the ability to handle different sibling and parent/child relationships. Thus, we used the same element names in each cluster and the same basic document structure as shown in Figure 8. The root node, $v_1$, has the same element name and frequency ($f_1$=1) for all documents. All other frequencies, ($f_2$ - $f_{25}$), are set to 1-4.

For cluster 1 and 2 we shifted the elements within each level of the document by one and two positions, respectively. So for level 1 in cluster 1 we have the following labeling: $l(v_2) = l_{10}, l(v_3) = l_2, l(v_4) = l_8, l(v_5) = l_9$; in cluster 2: $l(v_2) = l_9, l(v_3) = l_{10}, l(v_4) = l_2, l(v_5) = l_8$. This is done similarly for the other levels as well. In clusters 3 and 4 we shifted down[12] all elements by one and two levels, respectively. This changed the number of occurrences of the elements, as there is a different number of elements on each level. For the clusters 5 and 6 we shifted all elements to the left and right by two positions, respectively. For this purpose, we interpreted the elements associated with $v_2$ to $v_{25}$ to be one big sequence with a wraparound from $v_{25}$ to $v_2$. For clusters 7 and 8 we swapped around element names in an arbitrary manner. For a complete overview of the labeling, see Appendix C.

### *8.2.4 Depth Data Set (Depth)*

In this data set we varied the depth of the documents from cluster to cluster. For this benchmark we modified the document structure, as it is difficult to vary the depth of the document in Figure 8, while keeping the

[12] Wrapping around vertically

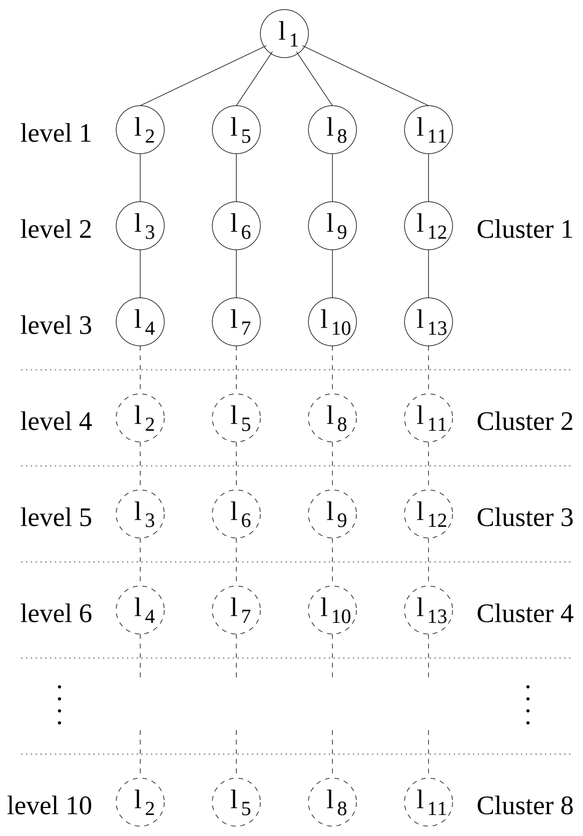


**Fig. 9** Document Used for Depth Data Set.

frequencies of the different elements roughly the same. For the Depth data set we used the document structure shown in Figure 9. The depth of the documents was increased from 4 (for cluster 1) to 11 (for cluster 8). The frequency for the element names $l_2$, $l_5$, $l_8$, and $l_{11}$ is 4 for cluster 1, all other frequencies are 1. For the subsequent clusters the frequencies of the elements were modified, such that each element appears roughly the same number of times in the document. Table 2 shows the exact frequency values for the clusters 1 to 8. All nodes on the same level of a document in the Depth data set have the same frequency.

| cluster | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 |
|---|---|---|---|---|---|---|---|---|
| level 1 | 4 | 2 | 2 | 2 | 1 | 1 | 1 | 1 |
| level 2 | 1 | 2 | 1 | 1 | 2 | 1 | 1 | 1 |
| level 3 | 1 | 1 | 2 | 1 | 1 | 2 | 1 | 1 |
| level 4 | - | 0-1 | 0-1 | 1 | 1 | 1 | 2 | 1 |
| level 5 | - | - | 1 | 1 | 1 | 1 | 1 | 2 |
| level 6 | - | - | - | 1 | 1 | 1 | 1 | 1 |
| level 7 | - | - | - | - | 0-1 | 0-1 | 0-1 | 0-1 |
| level 8 | - | - | - | - | - | 1 | 1 | 1 |
| level 9 | - | - | - | - | - | - | 1 | 1 |
| level 10 | - | - | - | - | - | - | - | 1 |

**Table 2** Frequencies for the Depth Data Set.

### *8.2.5 Horizontal/Vertical Data Set (HV)*

Finally, we investigated the effect of slowly shifting a horizontal arrangement of elements into a vertical one, checking how well the different techniques can distinguish between horizontal and vertical relationships of elements. The base document we used for the first cluster was a document containing a root element labeled $l_1$, which had six child elements labeled $l_2$ to $l_7$ from left to right. The frequency of appearance for each child ranged from one to four (using a uniformly random distribution).

For clusters two to six we moved the labels $l_3$ to $l_7$ in succession beneath the (final) node with label $l_2$, so cluster six would have $l_7$ beneath $l_6$ beneath $l_5$ and so on up to $l_2$. For cluster seven we took the base document and moved $l_3$ beneath $l_2$, $l_5$ beneath $l_4$, and $l_7$ beneath $l_6$, while for cluster eight $l_4$ was moved beneath $l_3$, which was moved beneath $l_2$, and $l_7$ was moved beneath $l_6$, which was moved beneath $l_5$. For a more detailed description, see Appendix D.

## 8.3 Effectiveness Results

Figure 10 shows the results of our experimental evaluation. The columns stand for the ten different data sets used in the experiments and the error rate over all data sets, while the rows show the results for the different similarity measures. The numbers in the table are the misclusterings for each combination of dataset and similarity measure.

For the DFT data sets, DFT1 and DFT2, we used two different sets of parameters: as in [25] choices (|) and ? were modeled using a uniform distribution for both, while * and + were modeled using a log-normal distribution (with $\mu = .75$, $\sigma^2 = .75$ for DFT1 and $\mu = 4$, $\sigma^2 = 1$ for DFT2).

For the DFT algorithm we show the results for the direct multilevel and pairwise multilevel encodings, which outperformed the other encodings [25]. For our compression technique NCD we have one additional variant, called *simple*, in which no structural information was extracted from the XML documents, i.e., they were compressed as they were, which resembles the original NCD idea.

The results for the Path Shingle algorithm were obtained by running it with a window size of 1 (according to [13] and our own experience, the window size has almost no effect on the accuracy; the differences lie within a tenth of a percent). The original Path Shingle algorithm hashes the components of a *Path* structure vector. In order to make it more efficient, we also ran it using *Doc* and *Pair* structure vectors instead, turning

| | SIGMOD | INEX | Music | DFT1 | DFT2 | Elem | Freq | Pos | Depth | HV | Overall |
|---|---|---|---|---|---|---|---|---|---|---|---|
| No of docs | 57 | 60 | 34 | 140 | 140 | 80 | 80 | 80 | 80 | 80 | 831 |
| TED | 1 | 0 | 13 | 26 | 0 | 0 | 0 | 5 | 0 | 3 | 5.8% |
| DFT | | | | | | | | | | | |
| direct ML | 0 | 3 | 9 | 52 | 1 | 27 | 9 | 32 | 33 | 24 | 22.9% |
| pairwise ML | 0 | 4 | 7 | 39 | 3 | 19 | 0 | 42 | 22 | 19 | 18.7% |
| PS | | | | | | | | | | | |
| Doc | 0 | 0 | 0 | 42 | 0 | 0 | 0 | 14 | 48 | 55 | 19.1% |
| Pair | 0 | 0 | 1 | 6 | 0 | 0 | 0 | 6 | 39 | 4 | 6.7% |
| Path | 0 | 0 | 0 | 6 | 0 | 0 | 0 | 0 | 30 | 4 | 4.8% |
| PVT | | | | | | | | | | | |
| cosine | 0 | 0 | 0 | 10 | 0 | 0 | 12 | 5 | 4 | - | 4.1% |
| intersection | 0 | 3 | 0 | 6 | 0 | 0 | 12 | 11 | 2 | - | 4.5% |
| KL | 0 | 0 | 0 | 6 | 0 | 0 | 0 | 0 | 3 | - | 1.2% |
| Dice | | | | | | | | | | | |
| Doc | 1 | 0 | 0 | 49 | 0 | 0 | 0 | 14 | 52 | 57 | 20.8% |
| Doc+ | 1 | 0 | 0 | 31 | 0 | 0 | 0 | 13 | 52 | 56 | 18.4% |
| Pair | 0 | 0 | 0 | 8 | 0 | 0 | 0 | 3 | 37 | 4 | 6.3% |
| Path | 0 | 0 | 0 | 6 | 0 | 0 | 0 | 0 | 2 | 4 | 1.4% |
| Family | 0 | 0 | 0 | 42 | 0 | 0 | 0 | 14 | 52 | 54 | 19.5% |
| *pq*-gram | 0 | 2 | 0 | 1 | 0 | 0 | 0 | 0 | 3 | 0 | 0.7% |
| NCD | | | | | | | | | | | |
| simple | 1 | 2 | 0 | 15 | 19 | 3 | 29 | 15 | 44 | 31 | 19.1% |
| Doc | 0 | 0 | 0 | 20 | 31 | 0 | 16 | 0 | 41 | 53 | 19.4% |
| Doc+ | 0 | 0 | 0 | 7 | 20 | 0 | 16 | 0 | 6 | 3 | 6.3% |
| Pair | 0 | 0 | 0 | 7 | 38 | 0 | 20 | 0 | 26 | 12 | 12.4% |
| Path | 0 | 0 | 1 | 2 | 38 | 0 | 24 | 0 | 3 | 8 | 9.1% |
| Family | 0 | 0 | 0 | 23 | 34 | 0 | 13 | 8 | 0 | 16 | 11.3% |
| *pq*-gram | 0 | 0 | 0 | 0 | 17 | 0 | 9 | 0 | 0 | 0 | 3.1% |
| CPD | | | | | | | | | | | |
| Doc | 0 | 2 | 0 | 6 | 0 | 0 | 7 | 0 | 38 | 60 | 13.6% |
| Doc+ | 0 | 2 | 0 | 1 | 0 | 0 | 0 | 0 | 0 | 0 | 0.4% |
| Pair | 0 | 2 | 0 | 6 | 0 | 0 | 7 | 0 | 19 | 3 | 4.5% |
| Path | 0 | 2 | 0 | 6 | 0 | 0 | 7 | 0 | 0 | 3 | 2.2% |
| Family | 0 | 0 | 0 | 1 | 0 | 0 | 0 | 2 | 0 | 16 | 2.3% |
| *pq*-gram | 0 | 2 | 0 | 1 | 0 | 0 | 0 | 0 | 0 | 0 | 0.4% |

**Fig. 10** Number of Misclusterings for Different Methods.

it into an algorithm with linear complexity. We investigate how well the more efficient variants perform in terms of clustering quality.

Cherukuri and Candan used the data sets described in Section 8.2 to run their benchmarks. The results shown in Figure 10 for PVT are taken directly from [18].

### *8.3.1 Tree Edit Distance (TED)*

The tree edit distance performed well for all data sets except Music, DFT1, and DFT2. The main problem with the Music XML files is the large variation in document sizes within the clusters, resulting in a large number of edit operations. For example, in one cluster (belonging to the same DTD) the smallest document consisted of 315 elements, while the largest consisted of 32680 elements.

The tree edit distance had the most trouble clustering documents belonging to DTD4 correctly, confirming results found in [25] (where the tree edit distance was also run against the DFT algorithm). The reason for this is that DTD4 is the only DTD that contains a repetition of a pair of nodes: (x,y)*, which means that we have a pattern that is not restricted to a single subtree but spans two subtrees. As the tree edit distance can only handle a single subtree in each edit operation, it is not able to detect this pattern correctly. Summarizing, the tree edit distance performs well for DTDs without repetition, but breaks otherwise.

### *8.3.2 Discrete Fourier Transformation (DFT)*

The DFT algorithms, although having a strong showing for the real data sets, DFT2, and the Frequency data set, perform quite poorly for the other data sets. It does not come as a surprise that these algorithms are very good at detecting differences in frequencies of appearances, as they were originally developed for analyzing frequencies. However, they fail to distinguish elements that appear with roughly the same frequency at different locations within a document. We did not expect the high number of misclusterings for the Element data set, as different tag names are encoded differently. It seems that the tag encoding only plays a minor role in the DFT method for detecting similarity. The gap between the results for DFT1 and DFT2 shows that, if the frequency of tags is low, as in the case of DFT1, this leads to a higher number of misclusterings by the DFT algorithms.

### 8.3.3 Path Shingles (PS)

Except for one case, the Depth data set, the path shingle algorithms exhibits similar strengths and weaknesses as the tree edit distance. Overall, the full path shingle variant is able to outperform the DFT algorithms and to match the tree edit distances in terms of accuracy. Running the path shingles with the *Doc* and *Pair* extraction technique results in an algorithm having linear run-time, but the accuracy of these variants drops considerably.

### 8.3.4 Propagation Vectors for Trees (PVT)

Looking at the three different variants used by Cherukuri and Candan to compare concept vectors, the Kullback-Leibler (KL) inspired distance emerged as a clear winner. However, this approach is not able to outperform crossparsing with *pq*-grams for structure extraction: PVT with KL distance has three times as many misclustered documents as *pq*-gram with crossparsing.[13] In addition to that, the main memory implementation described in [18] was not able to handle the largest document in the Music collection: during the propagation process (in which each node is represented by a concept vector) the PVT algorithm ran out of memory.

### 8.3.5 Dice Coefficient (Dice)

In this approach we use a simple Dice coefficient on the output of the structure extraction algorithms. As expected for the extraction techniques *Doc*, *Doc+*, *Pair*, and *Path*, the more information we extract, the better the Dice coefficient performs as a similarity measure. The only approach that breaks ranks is *Family*, which performs weaker than all other techniques. Overall, the performance of the Dice coefficient is best when using *pq*-grams. The combination of *pq*-grams with the Dice coefficient is the *pq*-gram distance as defined by Augsten et al. [5].

### 8.3.6 Compression (NCD)

For the compression-based techniques it becomes quite clear that just compressing the original XML documents without any structure extraction (denoted by *simple*) does not achieve the goal. Indeed, the main motivation for adding this naive method was to illustrate the benefits of extracting structural information first.

The compression-based techniques have the most problems with the DFT and the Frequency data sets. The more tags of a certain type occur, the better they can be compressed. As soon as a certain pattern appears sufficiently often in a document, it gets its own entry in the gzip dictionary, resulting in a very good compression rate for this pattern regardless of how often it appears in the other document. This makes it difficult to detect differences in frequencies by only looking at the size of the compressed files. For the DFT data sets we have the same picture as for the tree edit distance, meaning that compression-based algorithms have problems clustering documents of the DTD4 type.

In contrast to the Dice coefficient (where we form multisets consisting of vector components), the order of the components in a structure vector plays a role for the compression distance. If the same sequence of components appears in two structure vectors, then this results in a better compression rate compared to the same components appearing out of sequence. *Doc+* performs much better than *Doc*, as an ordered sequence of start and end tags defines the structure of an XML document unambiguously, while this may not be the case for an ordered sequence of start tags only.

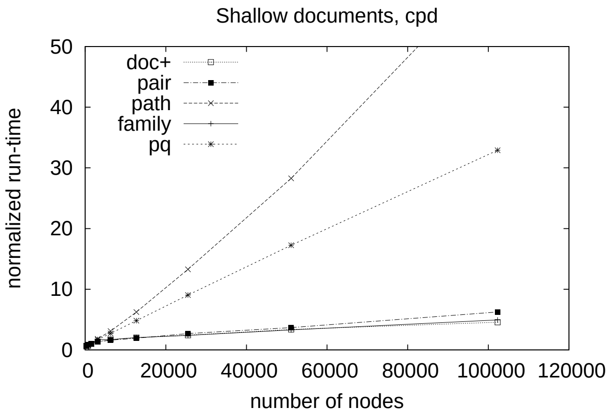
The *Family* traversal did not lead to significant improvements in terms of overall clustering performance. Although there were some improvements for the Frequency and the Depth data sets, for other data sets (DFT1, DFT2, and Position) the *Family* traversal performed worse. Again, the best results were achieved with *pq-grams*.

### 8.3.7 Crossparsing (CPD)

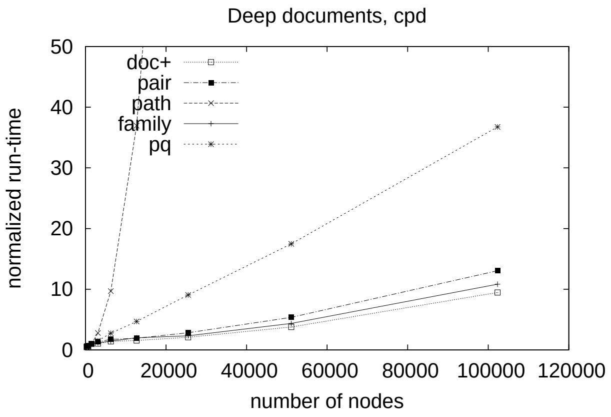
Overall, crossparsing outperformed all the other approaches in terms of clustering quality. There is only one case, the *Path* extraction technique combined with the Dice coefficient, for which it shows slightly worse behavior than one of its competitors. Crossparsing combined with the *Doc+* and *pq*-gram extraction techniques are the clear winners, misclustering only three documents and clustering all other documents correctly.

Compared to the other similarity measures, CPD puts the most emphasis on the order of the elements within a structure vector. This explains why two documents in the INEX data set were consistently misclustered, as two of the elements (<year> and <pages>) were in a different order in these two documents compared to the other documents in the same cluster.

[13] This does not include the HV data set, as the code for PVT was not available to us.



(a) Shallow documents



(b) Deep documents

**Fig. 11** Run-time for CPD.

### *8.3.8 Summary of Results*

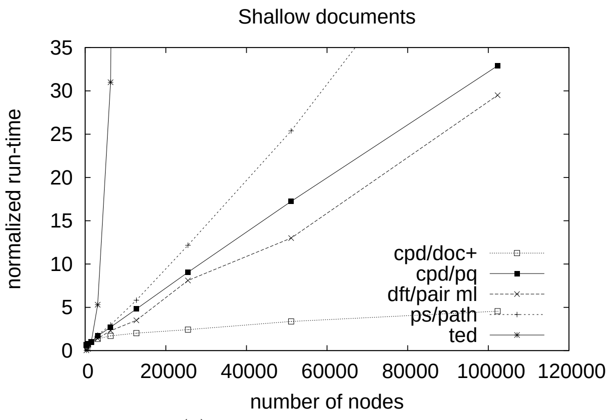
We conclude with a summary of the two-phase approaches that combine structure extraction and similarity measures. In Section 4 we have already identified the main characteristics of the structure extraction techniques: overlap of local patterns and the explicit representation of vertical and horizontal relationships. The overlap of patterns, which is part of *Pair*, *Path*, and *pq-grams*, preserves some of the (global) order. This is much more important for Dice, as it does not consider order when comparing multisets of structure vector components. By contrast, the compression distance and crossparsing consider the order of structure vectors components, thus reducing the need for extraction methods to explicitly represent the order within local patterns. This is reflected in the results shown in the last three blocks (Dice, NCD, CPD) of Figure 10. For the Dice coefficient, *Pair*, *Path*, and *pq-grams* perform much better than the other extraction techniques, while for NCD and CPD this is not necessarily the case. The only technique that explicitly represents vertical and horizontal relationships between nodes are *pq-grams*, which consistently comes out on top of the other extraction techniques for Dice, NCD, and CPD. Thus, preserving relationships seems to be an important feature when extracting the structure from documents.

When comparing the two information-theoretic measures, crossparsing performs much better than the compression distance. This confirms results in [20,42] showing that the compression distance has a crucial drawback: there is only a certain window in which we measure the difference between two data objects. When we start compressing two concatenated files, $X$ and $Y$, the gzip algorithm first builds a dictionary for $X$. We actually start measuring the difference between $X$ and $Y$ when we begin to parse $Y$, since at that point we try to encode $Y$ using a code created for $X$. As we continue parsing $Y$, more of the encoding will rely on the parts of $Y$ that we have already scanned and less will come from the dictionary for $X$. Puglisi et al. interpret this as a process in which the compression algorithm learns how to encode $Y$ and unlearns how to do so for $X$ [42]. Crossparsing, on the other hand, does not display such an effect, as we use $X$ to encode $Y$ and vice versa. Thus we do not consider the self-similarity of data objects when comparing them to others.

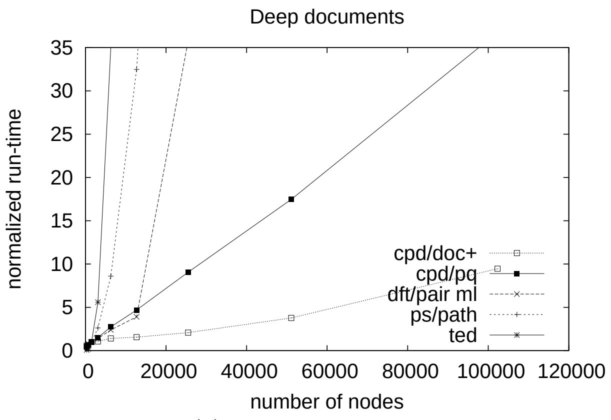
## 8.4 Scalability Results

In order to measure the run-time of the different algorithms experimentally we had to consider that they were implemented in different programming languages by different authors. For example, the comparison after extracting the structural information for the compression-based algorithms was done with a part UNIX shell script/C implementation solution, while the tree-editing algorithm was written in Java. Therefore, we present results for normalized run-time, i.e., we assume that all algorithms have a run-time of 1 second for a document containing 1500 nodes. The run-times for all other document sizes were divided by the actual run-time of an algorithm for a document size of 1500 nodes. In this way the asymptotic run-time complexity of the algorithms can be determined experimentally. We ran the algorithms on a set of shallow documents (ranging in size from 300 up to 102,300 nodes) and on a set of deep documents (also ranging in size from 300 to 102,300 nodes).[14]

[14] The DTDs of the documents can be found in Appendix E.



(a) Shallow documents



(b) Deep documents

**Fig. 12** Run-time for CPD and Competitors.

In Figure 11 we show the results for crossparsing combined with the various structure extraction techniques (Figure 11(a) depicts the normalized run times for shallow documents, while those in Figure 11(b) are for deep documents). The results for the compression-based algorithms (NCD) and the dice coefficient (Dice) are very similar to those for CPD, so we restrict ourselves to showing only CPD here. As can be seen in Figure 11, the Doc and Family variants are the quickest, followed closely by Pair (Pair, although a simpler algorithm than Family, is slower since it needs to handle twice as much output: every node together with its parent, rather than each node once.) *pq*-Gram is found in between the three quickest variants and Path. The linear complexity of CPD combined with *pq*-grams can be clearly seen, while Path is obviously not linear in complexity, as demonstrated by the results for the deep documents (the deep documents only have a minor impact on Doc, Pair, Family, and *pq*-Gram, while affecting the performance of Path significantly.

In Figure 12 we compare our three best variants in terms of clustering quality (CPD combined with *pq*-Gram, Doc, and Family) with competing approaches. Again, Figure 12(a) shows the results for shallow documents, while those in Figure 12(b) are for deep documents. Although the DFT and PS algorithms are still able to keep up with the CPD ones for shallow documents, for deep documents all algorithms apart from the CPD ones break away from a linear growth sooner or later.

## 9 Conclusion and Outlook

Determining the similarity of documents is an important aspect during retrieval or when clustering document collections. Measuring the similarity of semistructured data adds another dimension, as we have to be able to measure structural similarity as well. Due to the number of documents involved and their size, it is crucial to utilize an algorithm that shows good performance in terms of efficiency as well as in terms of output quality.

We proposed several different variants of a technique for measuring the structural similarity of semistructured documents based on information-theoretic concepts. In an extensive experimental evaluation that includes five competing methods we were able to illustrate the strengths and weaknesses of the different approaches. Looking at extraction techniques, *pq*-grams were the most robust method, meaning that they yielded the best overall results regardless of the similarity measure (Dice, NCD, or CPD) they were combined with. Only *Doc+* was able to keep up with *pq*-grams and only for CPD (it performed considerably worse for Dice and NCD).

Crossparsing had the best overall results for measuring the similarity of structure vectors, misclustering only a very small proportion of documents. Consequently, crossparsed *pq*-grams is one of the top techniques for measuring structural similarity.

An interesting direction for future work is to confirm our experimental results by analyzing the approach theoretically, e.g., by showing that our approach can be used as a bound for the (non-computable) information distance or Kullback-Leibler distance.

## References

1. A. Arasu and H. Garcia-Molina. Extracting structured data from web pages. In *ACM SIGMOD Int. Conf. on Manage-*

*ment of Data*, pages 337–348, 2003.

2. N. Augsten, D. Barbosa, M. Böhlen, and T. Palpanas. TASM: Top-k approximate subtree matching. In *Proceedings of the International Conference on Data Engineering (ICDE)*, pages 353–364, Long Beach, California, USA, March 2010. IEEE Computer Society.
3. N. Augsten, M. Böhlen, C. Dyreson, and J. Gamper. Approximate joins for data-centric XML. In *Proceedings of the International Conference on Data Engineering (ICDE)*, pages 814–823, Cancún, Mexico, April 2008. IEEE Computer Society.
4. N. Augsten, M. Böhlen, and J. Gamper. The pq-gram distance between ordered labeled trees. *ACM Transactions on Database Systems (TODS)*, 35(1), February 2010.
5. N. Augsten, M.H. Böhlen, and J. Gamper. Approximate matching of hierarchical data using pq-grams. In *Proc. of the 31st Int. Conf. on Very Large Data Bases (VLDB'05)*, pages 301–312, Trondheim, 2005.
6. R.A. Baeza-Yates and B.A. Ribeiro-Neto. *Modern Information Retrieval*. Addison Wesley, 1999.
7. Denilson Barbosa, Laurent Mignet, and Pierangelo Veltri. Studying the XML Web: Gathering statistics from an XML sample. *World Wide Web J.*, 8(4):413–438, 2005.
8. R. Baumgartner, S. Flesca, and G. Gottlob. Visual web information extraction with Lixto. In *Int. Conf. on Very Large Databases (VLDB'01)*, pages 119–128, 2001.
9. C.H. Bennet, P. Gács, M. Li, P.M.B Vitányi, and W.H Zurek. Information distance. *IEEE Transactions on Information Theory*, 44(4):1407–1423, July 1998.
10. E. Bertino, G. Guerrini, and M. Mesiti. A matching algorithm for measuring the structural similarity between an XML document and a DTD and its applications. *Inf. Syst.*, 29(1):23–46, 2004.
11. P. Bohannon, W. Fan, F. Geerts, X. Jia, and A. Kementsietsidis. Conditional functional dependencies for data cleaning. In *Proc. 23rd Int. Conf. on Data Engineering (ICDE)*, pages 746–755, Istanbul, 2007.
12. K. Bohrer, X. Liu, S. McLaughlin, E. Schonberg, and M. Singh. Object oriented XML query by example. In *ER (Workshops)*, pages 323–329, 2003.
13. D. Buttler. A short survey of document structure similarity algorithms. In *5th Int. Conf. on Internet Computing*, Las Vegas, Nevada, 2004.
14. G.J. Chaitin. On the length of programs for computing finite binary sequences. *Journal of the ACM*, 13:547–569, 1966.
15. S. Chaudhuri, V. Ganti, and R. Motwani. Robust identification of fuzzy duplicates. In *Proc. 21st Int. Conf. on Data Engineering (ICDE)*, pages 865–876, Tokyo, 2005.
16. S. Chawathe and H. Garcia-Molina. Meaningful change detection in structured data. In *ACM SIGMOD Int. Conf. on Management of Data*, pages 26–37, 1997.
17. S. Chawathe, A. Rajaraman, H. Garcia-Molina, and J. Widom. Change detection in hierarchically structured information. In *ACM SIGMOD Int. Conf. on Management of Data*, pages 493–504, 1996.
18. V.S. Cherukuri and K. Selçuk Candan. Propagation-vectors for trees (PVT): concise yet effective summaries for hierarchical data and trees. In *2008 ACM Workshop on Large-Scale Distributed Systems for Information Retrieval (LSDS-IR)*, Napa Valley, California, 2008.
19. R. Cilibrasi and P.M.B Vitányi. Clustering by compression. *IEEE Transactions on Information Theory*, 51(4):1523–1545, April 2005.
20. D.P. Coutinho and M.A.T. Figueiredo. Information theoretic text classification using the Ziv-Merhav method. In *Proc. 2nd Iberian Conf. on Pattern Recognition and Image Analysis (IbPRIA)*, pages 355–362, Estoril, Portugal, 2005.
21. T.M. Cover and J.A. Thomas. *Elements of Information Theory*. Wiley, 2006.
22. V. Crescenzi, G. Mecca, and Paolo Merialdo. RoadRunner: Towards automatic data extraction from large web sites. In *Int. Conf. on Very Large Databases (VLDB'01)*, pages 109–118, 2001.
23. T. Dalamagas, T. Cheng, K.-J. Winkel, and T. Sellis. A methodology for clustering XML documents by structure. *Inf. Syst.*, 31(3):187–228, 2006.
24. D. de Castro Reis, P.B. Golgher, A.S. da Silva, and A.H.F. Laender. Automatic web news extraction using tree edit distance. In *13th Int. World Wide Web Conf. (WWW'04)*, Manhattan, New York, 2004.
25. S. Flesca, G. Manco, E. Masciari, L. Pontieri, and A. Pugliese. Fast detection of XML structural similarity. *IEEE Transactions on Knowledge and Data Engineering*, 17(2):160–175, February 2005.
26. M.N. Garofalakis, A. Gionis, R. Rastogi, S. Seshadri, and K. Shim. XTRACT: A system for extracting document type descriptors from XML documents. In *ACM SIGMOD Int. Conf. on Management of Data*, pages 165–176, 2000.
27. P. Grünwald and P.M.B. Vitányi. Shannon information and Kolmogorov complexity. *The Computing Research Repository (CoRR)*, cs.IT/0410002, 2004.
28. S. Helmer. Measuring the structural similarity of semistructured documents using entropy. In *Proc. of the 33rd Int. Conf. on Very Large Data Bases (VLDB'07)*, pages 1022–1032, Vienna, 2007.
29. M.A. Hernandez and S.J. Stolfo. The merge/purge problem for large databases. In *ACM SIGMOD Int. Conf. on Management of Data*, pages 127–138, 1995.
30. N. Jardine and R. Sibson. *Mathematical Taxonomy*. John Wiley and Sons, New York, 1971.
31. J.W. Kim and K. Selçuk Candan. Cp/cv: concept similarity mining without frequency information from domain describing taxonomies. In *2006 ACM Int. Conf. on Information and Knowledge Management (CIKM)*, pages 483–492, Arlington, Virginia, 2006.
32. D. Knuth. *The Art of Computer Programming, Volume I: Fundamental Algorithms*. Addison Wesley, 1973.
33. A.N. Kolmogorov. Three approaches to the quantitative definition of information. *Problems of Information Transmission*, 1:1–7, 1965.
34. S. Kullback. *Information Theory and Statistics*. Dover Publications, New York, 1968.
35. M.L. Lee, L.H. Yang, W. Hsu, and X. Yang. XClust: clustering XML schemas for effective integration. In *11th Int. Conf. on Information and Knowledge Management (CIKM'02)*, McLean, Virginia, 2002.
36. M. Li and P.M.B Vitányi. *An Introduction to Kolmogorov Complexity*. Springer, New York, 1997.
37. Wang Lian, David Wai-Lok Cheung, Nikos Mamoulis, and Siu-Ming Yiu. An efficient and scalable algorithm for clustering xml documents by structure. *IEEE Transactions on Knowledge and Data Engineering (TKDE)*, 16(1):82–96, January 2004.
38. A. Martins. String kernels and similarity measures for information retrieval. Technical report, Priberam, Lisbon, Portugal, 2006.
39. M. Mesiti, E. Bertino, and G. Guerrini. An abstraction-based approach to measuring the structural similarity between two unordered XML documents. In *ISICT '03: Proceedings of the 1st international symposium on Information and communication technologies*, pages 316–321, 2003.
40. S. Nestorov, S. Abiteboul, and R. Motwani. Extracting schema from semistructured data. In *ACM SIGMOD Int. Conf. on Management of Data*, pages 295–306, 1998.

41. A. Nierman and H.V. Jagadish. Evaluating structural similarity in XML documents. In *Proc. of the 5th International Workshop on the Web and Databases (WebDB)*, pages 61–66, Madison, Wisconsin, 2002.
42. A. Puglisi, D. Benedetto, E. Caglioti, V. Loreto, and A. Vulpiani. Data compression and learning in time sequences analysis. *Physica D*, 189:92–107, 2003.
43. S. Santini and R. Jain. Similarity measures. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 21(9), September 1999.
44. S. Selkow. The tree-to-tree editing problem. *Information Processing Letters*, 6(6):184–186, 1977.
45. C.E. Shannon. The mathematical theory of communication. *Bell System Technical Journal*, 27:379–423, 623–656, 1948.
46. D. Shasha and K. Zhang. *Pattern Matching in Strings, Trees, and Arrays*, chapter Approximate Tree Pattern Matching. Oxford University Press, 1995.
47. P.H.A. Sneath and R.R. Sokal. *Numerical Taxonomy.* Freeman, San Francisco, 1973.
48. R. Solomonoff. A formal theory of inductive inference, part I. *Information and Control*, 7(1):1–22, 1964.
49. R. Solomonoff. A formal theory of inductive inference, part II. *Information and Control*, 7(2):224–254, 1964.
50. K.C. Tai. The tree-to-tree correction problem. *Journal of the ACM*, 26(3):422–433, 1979.
51. A. Theobald and G. Weikum. The XXL search engine: ranked retrieval of XML data using indexes and ontologies. In *ACM SIGMOD Int. Conf. on Management of Data*, page 615, 2002.
52. E. Ukkonen. On-line construction of suffix trees. *Algorithmica*, 14(3):249–260, 1995.
53. J. Wang, K. Zhang, K. Jeong, and D. Shasha. A system for approximate tree matching. *IEEE Trans. on Knowledge and Data Eng.*, 6(4):559–571, 1994.
54. P. Weiner. Linear pattern matching algorithms. In *14th Annual Symp. on Found. of Computer Science (FOCS)*, pages 1–11, Iowa City, Iowa, 1973.
55. I.H. Witten, A. Moffat, and T.C. Bell. *Managing Gigabytes.* Morgan Kaufmann, San Francisco, 1999.
56. K. Zhang and D. Shasha. Simple fast algorithms for the editing distance between trees and related problems. *SIAM J. Comput.*, 18(6):1245–1262, 1989.
57. J. Ziv and A. Lempel. A universal algorithm for sequential data compression. *IEEE Transactions on Information Theory*, 23(3):337–343, 1977.
58. J. Ziv and N. Merhav. A measure of relative entropy between individual sequences with application to universal classification. *IEEE Transactions on Information Theory*, 39(4):1270–1279, July 1993.

## A DTDs Used in the DFT Paper

```
<!DOCTYPE DTD1 [
  <!ELEMENT XML (a*)>
  <!ELEMENT a (b,c,d,e*)>
  <!ELEMENT b (f?)>
  <!ELEMENT c (g|h)>
  <!ELEMENT d EMPTY>
  <!ELEMENT e EMPTY>
  <!ELEMENT f EMPTY >
  <!ELEMENT g EMPTY>
  <!ELEMENT h EMPTY>
]>
```

```
<!DOCTYPE DTD2 [
  <!ELEMENT XML (a1*)>
  <!ELEMENT a1 (b1,c1,d1,e1*)>
  <!ELEMENT b1 (f1?)>
  <!ELEMENT c1 (g1|h1)>
  <!ELEMENT d1 EMPTY>
  <!ELEMENT e1 EMPTY>
  <!ELEMENT f1 EMPTY >
  <!ELEMENT g1 EMPTY>
  <!ELEMENT h1 EMPTY>
]>
```

```
<!DOCTYPE DTD3 [
  <!ELEMENT XML (h*)>
  <!ELEMENT h (f,g)>
  <!ELEMENT f (d*)>
  <!ELEMENT g (b|c)>
  <!ELEMENT d (a?)>
  <!ELEMENT b EMPTY>
  <!ELEMENT c EMPTY>
  <!ELEMENT a EMPTY>
]>
```

```
<!DOCTYPE DTD4 [
  <!ELEMENT XML ((x,y)*)>
  <!ELEMENT x ((a,w)|z*)>
  <!ELEMENT a EMPTY>
  <!ELEMENT w (c?)>
  <!ELEMENT c EMPTY>
  <!ELEMENT z (v,c)>
  <!ELEMENT v EMPTY>
  <!ELEMENT y EMPTY>
]>
```

```
<!DOCTYPE DTD5 [
  <!ELEMENT XML (m*,n)>
  <!ELEMENT m (q*)>
  <!ELEMENT q (x,y)>
  <!ELEMENT x ((a,c)|z*)>
  <!ELEMENT a EMPTY>
  <!ELEMENT c EMPTY>
  <!ELEMENT z EMPTY>
  <!ELEMENT n EMPTY>
  <!ELEMENT y EMPTY>
]>
```

```
<!DOCTYPE DTD6 [
  <!ELEMENT XML (m*,n)>
  <!ELEMENT m (x)>
  <!ELEMENT x ((a,c)|z*)>
  <!ELEMENT a EMPTY>
  <!ELEMENT c EMPTY>
  <!ELEMENT z EMPTY>
  <!ELEMENT n EMPTY>
]>
```

```
<!DOCTYPE DTD7 [
  <!ELEMENT XML (m)>
  <!ELEMENT m (x,n)>
  <!ELEMENT x (z*)>
  <!ELEMENT z EMPTY>
  <!ELEMENT n EMPTY>
]>
```

Note that there is an overlap between DTD5 and DTD6: the document <XML> <n> </n> </XML> could belong to both. However, this document was not generated when running the experimental evaluation.

## B XML Schema Used for Our Collections

Figure 13 shows a definition of the XML Schema used for generating the XML documents for our own data set collections.

## C Labeling for Position Data Set

In the following we describe in detail the labeling used for the different clusters in the position data set.

**Cluster 1:** Level 1: $l(v_2) = l_{10}$,$l(v_3) = l_2$,$l(v_4) = l_8$,$l(v_5) = l_9$
Level 2: $l(v_6) = l_7$,$l(v_7) = l_3$,$l(v_8) = l_7$,$l(v_9) = l_3$,$l(v_{10}) = l_7$,$l(v_{11}) = l_3$,$l(v_{12}) = l_7$,$l(v_{13}) = l_3$
Level 3: $l(v_{14}) = l_6$,$l(v_{15}) = l_4$,$l(v_{16}) = l_5$,$l(v_{17}) = l_6$,$l(v_{18}) = l_4$,$l(v_{19}) = l_5$,$l(v_{20}) = l_6$,$l(v_{21}) = l_4$,$l(v_{22}) = l_5$,$l(v_{23}) = l_6$,$l(v_{24}) = l_4$,$l(v_{25}) = l_5$

**Cluster 2:** Level 1: $l(v_2) = l_9$,$l(v_3) = l_{10}$,$l(v_4) = l_2$,$l(v_5) = l_8$
Level 2: $l(v_6) = l_3$,$l(v_7) = l_7$,$l(v_8) = l_3$,$l(v_9) = l_7$,$l(v_{10}) = l_3$,$l(v_{11}) = l_7$,$l(v_{12}) = l_3$,$l(v_{13}) = l_7$
Level 3: $l(v_{14}) = l_5$,$l(v_{15}) = l_6$,$l(v_{16}) = l_4$,$l(v_{17}) = l_5$,$l(v_{18}) = l_6$,$l(v_{19}) = l_4$,$l(v_{20}) = l_5$,$l(v_{21}) = l_6$,$l(v_{22}) = l_4$,$l(v_{23}) = l_5$,$l(v_{24}) = l_6$,$l(v_{25}) = l_4$

**Cluster 3:** Level 1: $l(v_2) = l_4$,$l(v_3) = l_5$,$l(v_4) = l_6$,$l(v_5) = l_4$
Level 2: $l(v_6) = l_2$,$l(v_7) = l_8$,$l(v_8) = l_9$,$l(v_9) = l_{10}$,$l(v_{10}) = l_2$,$l(v_{11}) = l_8$,$l(v_{12}) = l_9$,$l(v_{13}) = l_{10}$
Level 3: $l(v_{14}) = l_3$,$l(v_{15}) = l_7$,$l(v_{16}) = l_3$,$l(v_{17}) = l_7$,$l(v_{18}) = l_3$,$l(v_{19}) = l_7$,$l(v_{20}) = l_3$,$l(v_{21}) = l_7$,$l(v_{22}) = l_3$,$l(v_{23}) = l_7$,$l(v_{24}) = l_3$,$l(v_{25}) = l_7$

**Cluster 4:** Level 1: $l(v_2) = l_3$,$l(v_3) = l_7$,$l(v_4) = l_3$,$l(v_5) = l_7$
Level 2: $l(v_6) = l_4$,$l(v_7) = l_5$,$l(v_8) = l_6$,$l(v_9) = l_4$,$l(v_{10}) = l_5$,$l(v_{11}) = l_6$,$l(v_{12}) = l_4$,$l(v_{13}) = l_5$
Level 3: $l(v_{14}) = l_2$,$l(v_{15}) = l_8$,$l(v_{16}) = l_9$,$l(v_{17}) = l_{10}$,$l(v_{18}) = l_2$,$l(v_{19}) = l_8$,$l(v_{20}) = l_9$,$l(v_{21}) = l_{10}$,$l(v_{22}) = l_2$,$l(v_{23}) = l_8$,$l(v_{24}) = l_9$,$l(v_{25}) = l_{10}$

**Cluster 5:** Level 1: $l(v_2) = l_5$,$l(v_3) = l_6$,$l(v_4) = l_2$,$l(v_5) = l_8$
Level 2: $l(v_6) = l_9$,$l(v_7) = l_{10}$,$l(v_8) = l_3$,$l(v_9) = l_7$,$l(v_{10}) = l_3$,$l(v_{11}) = l_7$,$l(v_{12}) = l_3$,$l(v_{13}) = l_7$
Level 3: $l(v_{14}) = l_3$,$l(v_{15}) = l_7$,$l(v_{16}) = l_4$,$l(v_{17}) = l_5$,$l(v_{18}) = l_6$,$l(v_{19}) = l_4$,$l(v_{20}) = l_5$,$l(v_{21}) = l_6$,$l(v_{22}) = l_4$,$l(v_{23}) = l_5$,$l(v_{24}) = l_6$,$l(v_{25}) = l_4$

**Cluster 6:** Level 1: $l(v_2) = l_9$,$l(v_3) = l_{10}$,$l(v_4) = l_3$,$l(v_5) = l_7$
Level 2: $l(v_6) = l_3$,$l(v_7) = l_7$,$l(v_8) = l_3$,$l(v_9) = l_7$,$l(v_{10}) = l_3$,$l(v_{11}) = l_7$,$l(v_{12}) = l_4$,$l(v_{13}) = l_5$
Level 3: $l(v_{14}) = l_6$,$l(v_{15}) = l_4$,$l(v_{16}) = l_5$,$l(v_{17}) = l_6$,$l(v_{18}) = l_4$,$l(v_{19}) = l_5$,$l(v_{20}) = l_6$,$l(v_{21}) = l_4$,$l(v_{22}) = l_5$,$l(v_{23}) = l_6$,$l(v_{24}) = l_2$,$l(v_{25}) = l_8$

**Cluster 7:** Level 1: $l(v_2) = l_4$,$l(v_3) = l_8$,$l(v_4) = l_3$,$l(v_5) = l_{10}$
Level 2: $l(v_6) = l_9$,$l(v_7) = l_6$,$l(v_8) = l_9$,$l(v_9) = l_6$,$l(v_{10}) = l_9$,$l(v_{11}) = l_6$,$l(v_{12}) = l_9$,$l(v_{13}) = l_6$
Level 3: $l(v_{14}) = l_2$,$l(v_{15}) = l_5$,$l(v_{16}) = l_7$,$l(v_{17}) = l_2$,$l(v_{18}) = l_5$,$l(v_{19}) = l_7$,$l(v_{20}) = l_2$,$l(v_{21}) = l_5$,$l(v_{22}) = l_7$,$l(v_{23}) = l_2$,$l(v_{24}) = l_5$,$l(v_{25}) = l_7$

**Cluster 8:** Level 1: $l(v_2) = l_2$,$l(v_3) = l_7$,$l(v_4) = l_9$,$l(v_5) = l_5$
Level 2: $l(v_6) = l_6$,$l(v_7) = l_8$,$l(v_8) = l_6$,$l(v_9) = l_8$,$l(v_{10}) = l_6$,$l(v_{11}) = l_8$,$l(v_{12}) = l_6$,$l(v_{13}) = l_8$
Level 3: $l(v_{14}) = l_4$,$l(v_{15}) = l_{10}$,$l(v_{16}) = l_3$,$l(v_{17}) = l_4$,$l(v_{18}) = l_{10}$,$l(v_{19}) = l_3$,$l(v_{20}) = l_4$,$l(v_{21}) = l_{10}$,$l(v_{22}) = l_3$,$l(v_{23}) = l_4$,$l(v_{24}) = l_{10}$,$l(v_{25}) = l_3$

## D Documents for HV Data Set

.

Figures 14 to 19 show the structure of the documents used in the Horizontal/Vertical Data Set. The numbers in the top right corner of each node indicate the frequency of that node.

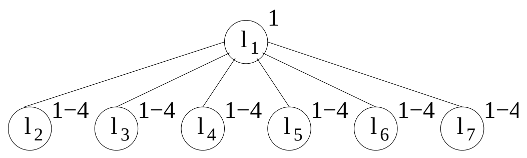


**Fig. 14** HV cluster one.

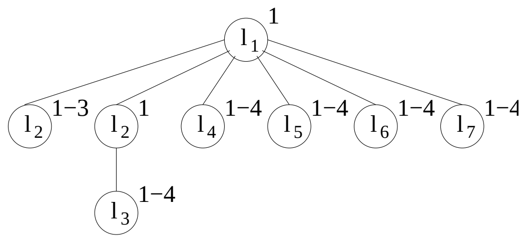


**Fig. 15** HV cluster two.

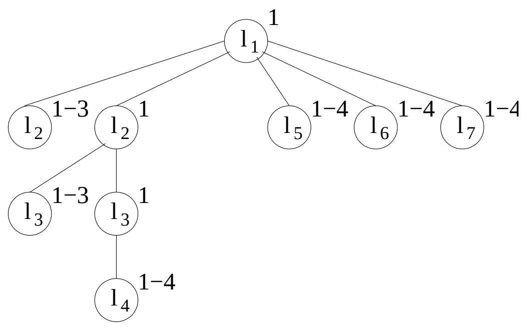


**Fig. 16** HV cluster three.

## E DTDs for Run-Time Experiments

The following two DTDs were used in the run-time experiments from Section 8.4. The DTD for the shallow documents was:

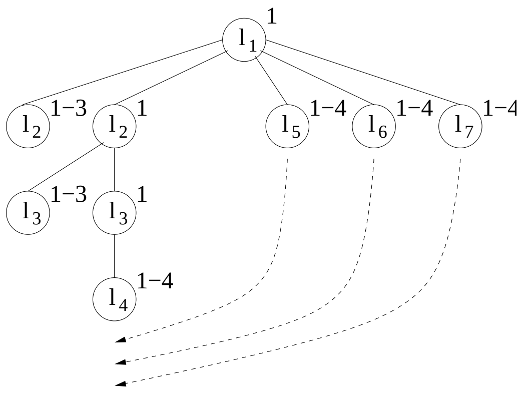


**Fig. 17** HV clusters four to six.

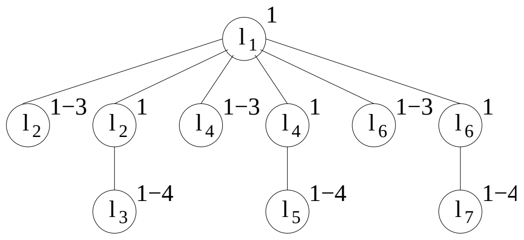


**Fig. 18** HV cluster seven.

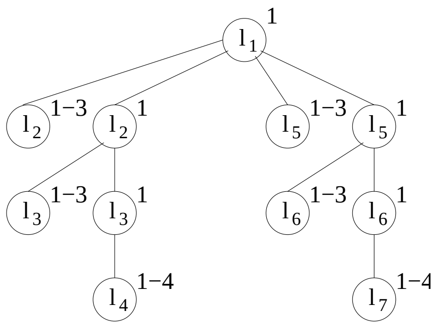


**Fig. 19** HV cluster eight.

```
<!DOCTYPE SHALLOW [
  <!ELEMENT TOPMOST (PERSON)>
  <!ELEMENT PERSON (NAME,AGE,CHILDREN)>
  <!ELEMENT NAME #PCDATA>
  <!ELEMENT AGE #PCDATA>
  <!ELEMENT CHILDREN (PERSON*)>
]>
```

The number of children a person element has was not chosen freely, but a person either had two children or none. We generated documents with $(2^i - 1) * 100$ nodes $(2 \leq i \leq 10)$ having $i$ levels (100 person elements on the top level and $2^{i-1} * 100$ persons on the other levels).

The DTD for the deep documents was:

```
<!DOCTYPE DEEP [
  <!ELEMENT TOPMOST (PERSON)>
  <!ELEMENT PERSON (NAME,AGE,CHILDREN)>
  <!ELEMENT NAME #PCDATA>
  <!ELEMENT AGE #PCDATA>
  <!ELEMENT CHILDREN (PERSON?)>
]>
```

Here we generated documents with $(2^i - 1) * 100$ nodes $(2 \leq i \leq 10)$ having $2^i - 1$ levels (100 person elements on each level).

```
<?xml version="1.0"?>
<xs:schema>

  <xs:element name="v1">
    <xs:complexType>
      <xs:sequence>

        <xs:element name="v2" minOccurs="f2.min" maxOccurs="f2.max">
          <xs:complexType>
            <xs:sequence>
              <xs:element name="v6" minOccurs="f6.min" maxOccurs="f6.max">
                <xs:complexType>
                  <xs:sequence>
                    <xs:element name="v14" type="xs:string" minOccurs="f14.min" maxOccurs="f14.max"/>
                    <xs:element name="v15" type="xs:string" minOccurs="f15.min" maxOccurs="f15.max"/>
                    <xs:element name="v16" type="xs:string" minOccurs="f16.min" maxOccurs="f16.max"/>
                  </xs:sequence>
                </xs:complexType>
              </xs:element>
              <xs:element name="v7" type="xs:string" minOccurs="f7.min" maxOccurs="f7.max"/>
            </xs:sequence>
          </xs:complexType>
        </xs:element>

        <xs:element name="v3" minOccurs="f3.min" maxOccurs="f3.max">
          <xs:complexType>
            <xs:sequence>
              <xs:element name="v8" minOccurs="f8.min" maxOccurs="f8.max">
                <xs:complexType>
                  <xs:sequence>
                    <xs:element name="v17" type="xs:string" minOccurs="f17.min" maxOccurs="f17.max"/>
                    <xs:element name="v18" type="xs:string" minOccurs="f18.min" maxOccurs="f18.max"/>
                    <xs:element name="v19" type="xs:string" minOccurs="f19.min" maxOccurs="f19.max"/>
                  </xs:sequence>
                </xs:complexType>
              </xs:element>
              <xs:element name="v9" type="xs:string" minOccurs="f9.min" maxOccurs="f9.max"/>
            </xs:sequence>
          </xs:complexType>
        </xs:element>

        <xs:element name="v4" minOccurs="f4.min" maxOccurs="f4.max">
          <xs:complexType>
            <xs:sequence>
              <xs:element name="v10" minOccurs="f10.min" maxOccurs="f10.max">
                <xs:complexType>
                  <xs:sequence>
                    <xs:element name="v20" type="xs:string" minOccurs="f20.min" maxOccurs="f20.max"/>
                    <xs:element name="v21" type="xs:string" minOccurs="f21.min" maxOccurs="f21.max"/>
                    <xs:element name="v22" type="xs:string" minOccurs="f22.min" maxOccurs="f22.max"/>
                  </xs:sequence>
                </xs:complexType>
              </xs:element>
              <xs:element name="v11" type="xs:string" minOccurs="f11.min" maxOccurs="f11.max"/>
            </xs:sequence>
          </xs:complexType>
        </xs:element>

        <xs:element name="v5" minOccurs="f5.min" maxOccurs="f5.max">
          <xs:complexType>
            <xs:sequence>
              <xs:element name="v12" minOccurs="f12.min" maxOccurs="f12.max">
                <xs:complexType>
                  <xs:sequence>
                    <xs:element name="v23" type="xs:string" minOccurs="f23.min" maxOccurs="f23.max"/>
                    <xs:element name="v24" type="xs:string" minOccurs="f24.min" maxOccurs="f24.max"/>
                    <xs:element name="v25" type="xs:string" minOccurs="f25.min" maxOccurs="f25.max"/>
                  </xs:sequence>
                </xs:complexType>
              </xs:element>
              <xs:element name="v13" type="xs:string" minOccurs="f13.min" maxOccurs="f13.max"/>
            </xs:sequence>
          </xs:complexType>
        </xs:element>

      </xs:sequence>
    </xs:complexType>
  </xs:element>

</xs:schema>
```

**Fig. 13** XML Schema for our data set collections