Measuring the Structural Similarity of Semistructured Documents Using Entropy

Sven Helmer
Birkbeck, University of London
Malet Street
London WC1E 7HX, United Kingdom
sven@dcs.bbk.ac.uk

ABSTRACT

We propose a technique for measuring the structural similarity of semistructured documents based on entropy. After extracting the structural information from two documents we use either Ziv-Lempel encoding or Ziv-Merhav crossparsing to determine the entropy and consequently the similarity between the documents. To the best of our knowledge, this is the first true linear-time approach for evaluating structural similarity. In an experimental evaluation we demonstrate that the results of our algorithm in terms of clustering quality are on a par with or even better than existing approaches.

1. INTRODUCTION

More and more information is stored in semistructured documents, be it in HTML, XHTML, or XML. Ever larger collections of these documents can be searched online. In traditional Information Retrieval (IR), detecting similarities within a group of unstructured text documents is used widely, e.g. in order to cluster them according to different topics or to use documents that have been identified as relevant by a user during the search (for an overview see [2]). Consequently, similarity measures for text documents have been a research topic since the 1970s. With the advent of semistructured documents new opportunities and challenges have arisen. Mark-up languages such as XML add structural information to documents and they allow a lot of freedom in doing so, which means we have collections of documents showing great structural diversity. In addition to measuring the similarity of the textual content (which can be done with the traditional IR techniques), we have to be able to cope with the structural similarity.

Identifying structural similarity can help in many ways. First, there are several proposals for extracting schema or document type information from a document collection [1, 17, 27]. The more homogeneous a collection is, i.e. the fewer out-of-place documents are contained in it, the better these extraction algorithms work. So it is a good idea to first cluster the documents and then apply the extraction algorithms. In addition to this, a similarity measure could also be used to compare schema information itself [4, 6] Second, being able to determine the structural similarity of documents while integrating heterogeneous data sources helps in identifying sources that supply similar types of information. Currently, the process of finding relevant web pages is mainly done manually and it is quite labor-intensive [3, 15, 22]. Automatically clustering web pages according to their structure will speed up this task considerably. Third, a similarity measure can also be used to merge and fuse data during entity resolution while cleaning data [9, 18]. As more and more data is exchanged or stored in XML format, there is a need for a similarity measure that is able to handle semi-structured data. Last but not least, inexperienced users of (web-)retrieval system often find it difficult to formulate complicated queries in, say, XQuery, to search for semistructured documents. Using a similarity measure could enable a system to allow its users to provide relevant example documents.

Many of the approaches proposed for detecting structural similarity (e.g. [28]) are based on tree-editing [36], computing the minimal editing costs for transforming one document's structure into the other. The main drawback of these approaches is their complexity of at least $O(N^2)$, where N is the number of tags in both documents.

Flesca et al. [16] take a completely different approach by quantifying the structures of two documents and interpreting the results as time series. These two time series are then analyzed and compared using the Discrete Fourier Transformation (DFT), resulting in an algorithm with a complexity of $O(N_1 \log N_1)$ for comparing two documents (where N_1, w.l.o.g., is the size of the larger document).

Buttler [8] uses a much simpler approach based on path shingles. This 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. Although this approach is more efficient than the previous two, in the worst case it is still not linear.

Our approach works in yet another way. We extract the structural information from a document by stripping away the content and then using the entropy between the structural data as a measure of similarity. As we will see, the quality of this approach in terms of avoiding misclusterings is on a par with or even better than the previously described approaches, while having a time complexity of $O(N)$ (where N is the number of tags in both documents).

The remainder of this paper is structured as follows. In

Section 2 we give a brief a brief introduction into the notion of information distance on which our approach is based. We describe the details (and different variants) of our similarity measure in Section 3, while the tree-editing algorithm by Nierman and Jagadish [28], the DFT approach by Flesca et al. [16], and the path shingle approach by Buttler [8] are explained in Section 4. Section 5 contains the results of our experimental evaluation, including a description of the experimental environment and an interpretation of the results. Related work will be covered in Section 6. Finally, we conclude this paper with a short summary and an outlook.

2. INFORMATION DISTANCE

Bennet et al. introduced the concept of using a universal information metric to measure similarity between data objects in [5]. Their work is based on Kolmogorov complexity: given a data object x (e.g. a binary string in $\{0,1\}^*$) the Kolmogorov complexity $K(x)$ is the length of the shortest (binary) program that outputs x. We assume here that this program is written for a universal computer such as a universal Turing machine. A more generalized form of the Kolmogorov complexity is the conditional Kolmogorov complexity $K(x|y)$, which is the length of the shortest program with input y that outputs x. Bennet et al. defined an information distance $E(x,y) = \max\{K(x|y), K(y|x)\}$ and its normalized version

$$NID(x,y) = E(x,y)/\max\{K(x), K(y)\} \tag{1}$$

to measure the similarity of two data objects. They also showed 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; for details see [5]). The only catch is that the Kolmogorov complexity generally is not computable.

However, the Kolmogorov complexity of a data object x can be seen as the length of the ultimate compressed version of x. Obviously, the ultimate compression is not computable either, but we can approximate it by using standard compression algorithms. In [12], Cilibrasi and Vitányi 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 $C(x)$, which is the length of the compressed x. The numerator, containing conditional Kolmogorov complexity expressions, needs to be rewritten before approximating it by compression. Exploiting the additive property of Kolmogorov complexity [24], the numerator of (1) can be rewritten to $\max\{K(x,y) - K(x), K(x,y) - K(y)\}$ within logarithmic additive precision. $K(x,y)$ is the length of the shortest program needed to produce the pair (x,y) of data objects. As it is easier to handle concatenation with compression, this can be estimated by $\min\{C(xy), C(yx)\} - \min\{C(x), C(y)\}$. Due to the symmetrical behavior of many compression algorithms this can be further simplified to $C(xy) - \min\{C(x), C(y)\}$, arriving at the definition of NCD found in [12]:

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

Cilibrasi and Vitányi applied their similarity measure quite successfully to clustering data objects in different domains such as genomics, literature, music, and astronomy.

```
<a>
  <b x="foo">
    <c>
      some text
    </c>
    <d y="bar">
      some more text
    </d>
  </b>
</a>
```

Figure 1: Example XML document

3. MEASURING STRUCTURAL SIMILARITY

We want to apply this technique to measuring the structural similarity between XML documents. While for Kolmogorov complexity the exact format of the data is irrelevant, for compression algorithms in the real world it is not. Depending on the data objects that are to be compared, preprocessing the data may have a significant impact on the quality of the results. For comparing structural similarity this means that we want to extract the structural information from an XML document before doing the actual comparison. In addition to applying the NCD technique in a straightforward fashion (just compressing the XML documents and comparing them[1]), we use four different methods to extract structural information from the XML documents. In the following we are going to have a look at them.

3.1 Tags

In order to illustrate the different methods we show examples using the XML document in Figure 1. We call the simplest method *Tags*, in which we strip XML documents of all content (e.g. text nodes and attribute values) and extract all other nodes in document order by outputting their tag names (both opening and closing) and possible attribute names. For the example document this would result in the sequence <a><b x><c></c><d y></d></b></a>.

3.2 Pairwise

Similar to *Tags*, *Pairwise* also removes all content from the XML document and keeps the remaining structural nodes. However, for each node the name of the parent node is prepended to its name. Again the document order is maintained but we refrain from outputting the closing tags in this case. So applying *Pairwise* to the example document yields: a abx bc bdy.

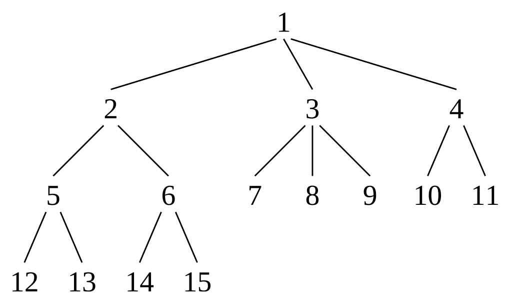
3.3 Full Path

Full Path adds even more information than *Pairwise*, as all node names are prepended with the full path from the root node to the current node (after removing the content). So, for the example document this produces the sequence: a abx abc abdy.

3.4 Family Order

An important concept when designing DTDs is the grouping of elements as siblings. Thus one could argue that describing the structural information of an XML document in

[1]This will lead to a low-quality result as we will see later.



Family order traversal:

```
12 13  14 15  5 6  7 8 9  10 11  2 3 4  1
```

Figure 2: Family order traversal

a breadth-first manner is as appropriate as traversing it in document order (e.g. Levene and Wood propose a breadth-first traversal for compressing and uncompressing XML documents [23]). However, breadth-first traversal is problematic due to its memory consumption: given an XML document of height h, breadth-first traversal may need space that is exponential in h, whereas depth-first traversal uses up space that is linear in h.

Family Order traversal represents a compromise between breadth-first and depth-first traversal [20]. All the children of a node are output en bloc, i.e. all siblings will end up next to each other. However, before doing so, all their descendants have to be processed. Figure 2 shows an example of a family order traversal. In this way we still manage to keep the sibling information intact without having to store whole levels of the tree during the traversal.

3.5 Document Similarity

Measuring the structural similarity of two XML documents works as follows. First, the structural information of both documents is extracted using one of the methods described in the previous sections. The resulting files are compressed (separately) with GNU zip (gzip version 1.3.5) and their size in bytes is noted. These two file sizes are the values we need in formula (2) for $C(x)$ and $C(y)$ to calculate the normalized compression distance. Then the files containing the extracted structural information are concatenated and this concatenation is compressed again. The size of this compressed file yields the value for $C(xy)$. All that is left to do now is to compute the actual value for the NCD.

Compression of files using the Ziv-Lempel approach (as in gzip) can be done in linear time using suffix trees as an index structure for finding the location of the longest matching substring [21]. This means that, except for full path extraction, the similarity of two XML documents can also be computed in linear time.

As a variant we used crossparsing as applied by Ziv and Merhav to measure relative entropy between sequences [38] (for more details see Section 3.7). Instead of actually compressing the documents, we parse them with respect to each other, i.e. we step through one document trying to find the longest prefix that will match a sequence in the other document. Using suffix trees this can also be done in linear time [25].

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

Figure 3: Example for Ziv-Lempel

3.6 Ziv-Lempel Encoding

In order to understand the results of the experimental evaluation better, it helps to know the basic functionality of the Ziv-Lempel family of encoders (of which gzip is a member). 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 form `<distance, length, symbol>` and is generated on the fly during encoding (and decoding). See Figure 3 for a brief example. For our similarity measure 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. For more details on this compression technique see [35, 37].

3.7 Ziv-Merhav Crossparsing

(Cross-)parsing document x with respect to document y, both of which we assume to be represented as strings, works as follows. First we find the longest prefix of x that appears as a string in 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 x with respect to y (if $x_1 \notin y$, then we set $m = 1$)[2]. After determining the first value for m, we find the longest prefix of x starting with x_{m+1} with respect to y. We continue doing so until we have parsed the whole document x. Let $c(x|y)$ be equal to the number of phrases when parsing x with respect to y. For example, let $x = abbbbaaabba$ and $y = baababaabba$. Then

[2]We count the number of times we have to (re-)start the parsing.

$c(x|y) = 3$, the three phrases being *abb*, *bba*, and *aabba*. As a similarity measure between documents x and y we use $\frac{c(x|y)-1+c(y|x)-1}{2}$, i.e. we crossparse both ways (taking the average) and we count the number of times that we have to reapply finding the longest prefix.

Another important difference between the gzip compression and how we apply crossparsing is the different granularity. The compression algorithm identifies characters as the smallest entities in a document, while for the crossparsing algorithm we chose tag names as smallest entities. This should reflect the actual structure of the document better, as we are interested in matching sequences of tags and not sequences of substrings of tag names.

4. THE COMPETITORS

We compare our method to three other techniques: one based on tree-editing distances by Nierman and Jagadish [28], another one based on Discrete Fourier Transformation (DFT) by Flesca et al. [16], and path shingles by Buttler [8]. We give a brief description of all three in the remainder of this section.

4.1 Tree-Editing Distance

The origins of algorithms computing tree-editing distances go back to the dynamic programming algorithms for finding minimum-editing (or Levenshtein) distances between strings. There are several variants of tree-editing distance algorithms depending on the edit operations they allow. Nierman and Jagadish adapted the edit operations to reflect the special requirements of XML documents. Due to optional and repeating elements, XML documents generated from the same DTD can show a considerable difference in their sizes (i.e. the number of nodes they contain). Consequently, a metric that permits changing only one node at a time may overestimate the distance. Therefore, Nierman and Jagadish utilize edit operations that can change more than one node per step. The provide five different operations:

- Relabel: give a node in a tree a new label
- Insert: insert a new node (without children) into a tree
- Delete: deletes a single node from a tree
- Insert Tree: inserts a new subtree into a tree
- Delete Tree: delete a node and all its descendants from a tree

Each of these operations has a non-negative cost associated with it. While the algorithm proposed by Nierman and Jagadish works with general costs, they restrict themselves to a constant unit cost for all operations.

There are two constraints on the Insert Tree and Delete Tree operations. Otherwise, every tree could be transformed into any other tree in just two steps (delete complete original tree, insert complete new tree). The two restrictions are: (1) a subtree may only be inserted if it already occurs in a tree and it may only be deleted if it still occurs in a tree. This models optional and repeated elements. (2) A subtree that has been inserted may not have additional nodes inserted afterwards, while a deleted subtree may not have had some its nodes deleted previously. This makes the computation of the editing costs more efficient.

The complexity of the algorithm is $O(|d_1||d_2|)$, where d_1 and d_2 are two (document) trees that are to be compared and $|d_1|$ and $|d_2|$ are their sizes (in number of nodes), respectively. For more details on the actual algorithm, see [28].

4.2 Discrete Fourier Transformation

The DFT technique takes a totally different approach. The features of an XML document relevant to its structure are described via a numerical encoding and these values are interpreted as a time series. (This approach can be visualized by rotating an XML document by 90 degrees and looking at the element indentations as a time series.)

4.2.1 Tag Encoding

The numerical encoding of the XML documents happens on two different levels: the encoding of the elements or tags and the encoding of a document's substructures (an attribute with name X is treated as a subelement named $ATTRIB@X$). For the tag encoding a function γ maps the tags of a set of documents to real numbers (excluding 0). Flesca et al. distinguish direct, pairwise, and nested encoding. The difference between these encodings is the extent of context that is considered. In the direct encoding, each tag is mapped directly to a real number, in pairwise encoding each tag/parent tag combination is mapped to a real number, while in the nested encoding a path from the root to an element is mapped to a real number. This corresponds to the first three methods used in our approach in Section 3.

4.2.2 Document Encoding

While a tag encoding maps individual tags to real numbers, a document encoding associates an XML document with a time series. A document encoding is a function *enc* that maps a document to a sequence of real numbers in such a way that no structural loss occurs. A function *enc* is without structural loss (WSL), if for each pair of documents d_1 and d_2, $enc(d_1) = enc(d_2)$ implies that d_1 and d_2 have the same sequence[3] of tag instances. Flesca et al. distinguish a trivial, linear, and multi-level encoding.

In the trivial encoding a document d is defined by its sequence of tags $t_1, t_2, \ldots, t_n$ mapped to its sequence of tag encodings:

$$enc(d) = \gamma(t_1), \gamma(t_2), \ldots, \gamma(t_n).$$

In the linear encoding each item in the time series is a linear combination of all tag encodings from the start of the document to the current element. So,

$$enc(d) = \gamma(t_1), \gamma(t_1) + \gamma(t_2), \ldots, \sum_{k \leq n} \gamma(t_k).$$

The multi-level encoding is the most complicated variant, in which the current tag encoding is combined with the tag encoding of all its ancestors in the document:

$$enc(d) = S_1, S_2, \ldots, S_n$$

where

$$\begin{aligned} S_i &= \gamma(t_i) \cdot B^{maxdepth-depth(t_i)} + \\ & \sum_{t_j \in anc(t_i)} \gamma(t_j) \cdot B^{maxdepth-depth(t_j)}. \end{aligned}$$

[3]in document order

maxdepth is the maximum depth for any document in the document collection, B is a fixed value (usually greater or equal to the number of distinct symbols encoded by γ, to avoid mixing tag information with substructure information), and $anc(t_i)$ is the set of tags on the path from the root of the document to t_i.

Let us present a small example. Assume that the direct tag encoding for our example document in Figure 1 (disregarding attributes for a moment) is: <a> = 1, </a> = −1, <b> = 2, </b> = −2, <c> = 3, </c> = −3, <d> = 4, and </d> = −4. Then the time series for the whole document using linear encoding would be: 1 3 6 3 7 3 1 0.

4.2.3 Document Similarity

Flesca et al. use an interpolated $\tilde{\text{DFT}}$ transform to calculate the distance between two documents d_1 and d_2:

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

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 ($M = |tags(d_1)| + |tags(d_2)|$). The motivation for using DFT was to abstract from the length of the documents and to determine whether a given subsequence of tags appears with a certain regularity (independent of the location of the subsequence).

The complexity of the algorithms is $O(N \log N)$ with $N = \max(|d_1|, |d_2|)$, where $|d_1|$ and $|d_2|$ represent the size of the documents in number of nodes. For more details on the DFT technique, see [16].

4.3 Path Shingles

In a first step structural information is extracted from the documents using the Full Path variant described in Section 3.3. The next step consists of computing a hash value h_j ($1 \leq j \leq |d_i|$, where $|d_i|$ is the number of nodes in d_i) for each of the paths in d_i. A shingle of width w is the combination of w consecutive hash values $h_j, \ldots, h_{j+w}$ (w is also called the *window size*). The set of all shingles of width w for document d_i is denoted by $S(d_i, w)$. The similarity of two documents can now be determined with the help of the Dice coefficient:

$$sim(d_i, d_k) = \frac{S(d_i, w) \cap S(d_k, w)}{S(d_i, w) \cup S(d_i, w)}$$

For a distance metric we can use:

$$dist(d_i, d_k) = 1 - sim(d_i, d_k)$$

5. EXPERIMENTAL EVALUATION

In this section we present the results of our experimental evaluation and interpret them. Before doing so, however, we briefly explain how we measured the quality of the similarity measures by using them to cluster XML documents associated with different DTDs and we describe the document collections we used in the experiments.

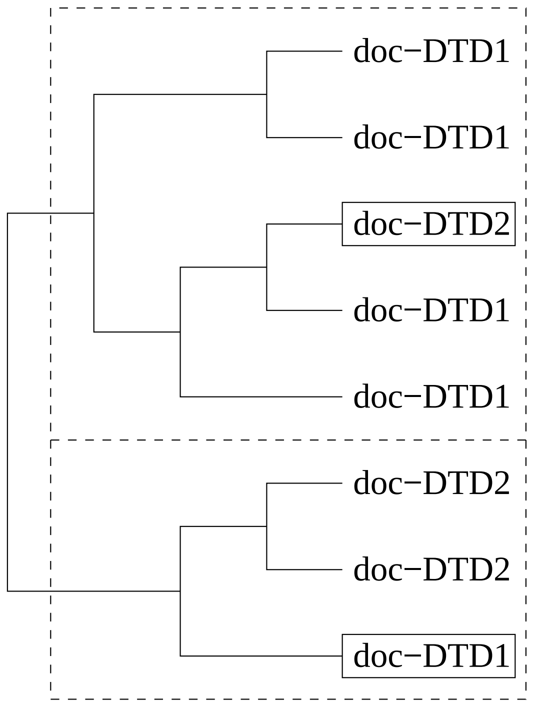


Figure 4: Example dendrogram

5.1 Measuring Clustering Quality

As it is difficult to determine the quality of a similarity measure in a direct way, we used the different measures to cluster collections of XML documents adhering to different DTDs. After clustering the documents we counted the number of documents that were misclustered, i.e., documents that were put into the wrong cluster. The lower this number, the better the similarity measure.

As clustering technique we used the well-known hierarchical agglomerative clustering [19] to make our results comparable to those of previous experiments. In this technique we start with each document representing its own cluster. In each subsequent step the two clusters that have the closest similarity are merged together until we end up with one large cluster. The steps taken by this algorithm can be represented visually in a so-called dendrogram (see Figure 4 for an example). The dendrogram shows, from right to left, in which order the document clusters were merged. To calculate cluster distances we applied the Unweighted Pair Group Averaging Method (UPGMA) [31] that was also used by Nierman and Jagadish in [28]:

$$dist(C_1, C_2) = \frac{1}{|C_1| \cdot |C_2|} \cdot \sum_{i=1}^{|C_1|} \sum_{j=1}^{|C_2|} dist(d_i^{C_1}, d_j^{C_2})$$

where C_1 and C_2 are two document clusters, $|C_i|$ denotes the number of documents contained in cluster C_i, and $d_j^{C_i}$ is the j-th document in cluster C_i.

In our experiments we first determined the similarity for each pair of documents in the collection and then built a dendrogram using UPGMA. The algorithms computing the similarity were not given any hint which DTD a document

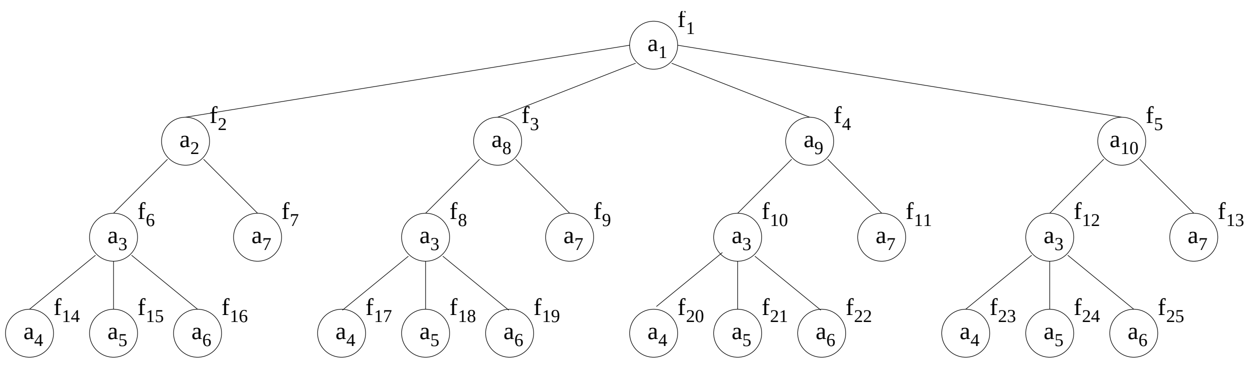


Figure 5: Basic document used for experiments

was associated with (i.e., we removed all information pertaining to DTDs). Afterwards we checked which DTD a document belonged to and identified documents that ended up in the wrong cluster. In Figure 4 these are the two documents marked by a box. So in this case we have two misclustered documents.

5.2 Document Collections

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[4], 60 documents from the heterogeneous track at INEX 2005[5], and 34 music sheets encoded in XML[6].

For the synthetically generated documents (generated with ToXgene[7]) we started out with the DTDs in the DFT paper [16]. As we wanted to have more control over certain 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.

At first glance the synthetically generated document collections might seem quite extreme, as there are often only nuances between different clusters. We (as well as Flesca et al. [16]) did this on purpose to test out the limits of the different approaches. The DTDs used by Flesca et al. can be found in the appendix, our collection is described in the following.

5.2.1 Element Data Set

As first parameter we varied the tag names, calling this data set *Element* because the only difference between the documents is the name of the elements. Figure 5 shows the basic structure of the documents we used for the synthetically generated data sets. For the Element data set we generated eight different clusters by replacing the tag names $a_1, a_2, \ldots, a_{10}$ with different names for each cluster. The frequencies of appearances for each element, represented by f_i, were set to 1-4 for each frequency except f_1 which was set to 1. The numbers mentioned in connection with frequencies denote the number of occurrences of an element.

[4]available at http://www.sigmod.org/record/xml/

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

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

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

5.2.2 Frequency Data Set

The next parameter we varied was the frequency of appearance of the different elements. All documents had the same element names and the basic structure as depicted in Figure 5, but each cluster had different values for the f_i (except for f_1, which was always set to 1). In the first cluster all frequencies were set to 1, in the second cluster all f_is (except f_1) were set to 3-4, in the third cluster f_2 to f_{13} were set to 1 and f_{14} to f_{25} were set to 3-4, and in the fourth cluster we set f_2 to f_5 to 3-4 and f_6 to f_{25} to 1. In the fifth to eighth clusters we varied the frequencies in a vertical manner. Cluster 5: f_2, f_4, and all the frequencies of the elements a_3, a_4, and a_5 set to 1, all others set to 3-4. (for cluster 6 the frequencies 1 and 3-4 were switched). In cluster 7 the following frequencies were set to 1: f_2, f_4, f_7, f_8, f_{11}, f_{12}, f_{14}, f_{15}, f_{19}, f_{20}, f_{21}, f_{25}. The other frequencies were set to 3-4. For cluster 8 this was reversed.

5.2.3 Position Data Set

In this data set we varied the position of the elements within the documents (using the same element names in each cluster and the same basic document structure as shown in Figure 5). The element a_1 was kept as root element for all documents (with $f_1 = 1$, while all other frequencies were set to 1-4). For cluster 1 and 2 we shifted[8] the elements within each level of the document by one and two positions, respectively. In clusters 3 and 4 we shifted[9] down 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 f_2 to f_{25} to be one big sequence with a wraparound from f_{25} to f_2. For clusters 7 and 8 we swapped around element names in an arbitrary manner.

5.2.4 Depth Data Set

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 was very difficult to vary the depth of the document in Figure 5 and by doing so keeping the frequencies of the different elements roughly the same. For the depth data set we used the document structure shown in

[8]Wrapping around on each level.

[9]Wrapping around vertically this time.

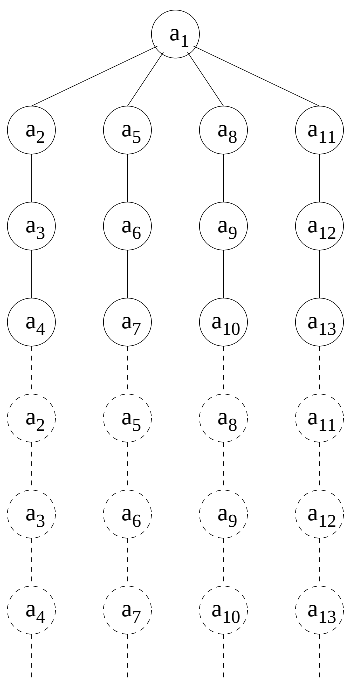


Figure 6: Document used for Depth Data Set

Figure 6. The depth of the documents was increased from 4 (for cluster 1) to 11 (for cluster 8). The frequency for the elements a_2, a_5, a_8, and a_{11} was set to 4 for cluster 1, all other frequencies were set to 1. For the subsequent clusters the frequencies of the elements were modified so as to have roughly the same number of each element in the document.

5.3 Results

Figure 7 shows the results of our experimental evaluation. The columns stand for the nine 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 itself give the number of misclusterings for each case.

For the DFT data sets we used two different sets of parameters: as in [16] choices (|) and ? were modeled using a uniform distribution for both, DFT1 and DFT2, 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 ran the direct multilevel and pairwise multilevel encoding (which outperformed the other encodings [16]). For our compression technique 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. We also added a hybrid version multiplying the distances supplied by a DFT algorithm with those of a crossparsing algorithm.

The results we present for the Path Shingle algorithm were run with a window size of 1 (according to [8] and our own experience, the window size has almost no effect on the accuracy; the differences lie within a tenth of a percent). For the sake of completeness, we also ran the path shingle algorithm with the tag and pairwise structure extraction techniques described in Section 3.

5.4 Interpretation of Results

5.4.1 Tree-Editing Distance

When looking at Figure 7, we notice several things. The tree-editing algorithm showed quite a strong clustering performance for all data sets except Music, DFT1, and DFT2. The main problem with the Music XML files seems to be the large variation in document sizes within the clusters, resulting in a larger number of edit operations than usual. For example, in one cluster (belonging to the same DTD) the smallest document consisted of 315 tags, while the largest consisted of 32680 tags. The clusters in the DFT collections are not all disjoint, e.g. the document <XML> <n> </n> </XML> could belong to DTD5 as well as DTD6. This alone, however, does not explain the rather large number of misclusterings. The tree-editing algorithm had the most trouble clustering documents belonging to DTD4 correctly (they showed up in lots of other clusters), confirming results found in [16] (where the tree-editing algorithm was also run against the DFT algorithm). We believe that 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-editing algorithms can only handle single subtrees in an edit operation, it is not able to detect this pattern correctly.

5.4.2 Discrete Fourier Transformation

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, when it comes to detecting elements appearing with roughly the same frequency at different locations within a document, they seem to fail. 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 a tag is too low, then the DFT algorithms are not able to detect small differences very well.

5.4.3 Path Shingles

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

5.4.4 Compression-Based Entropy

For the compression-based techniques it becomes quite clear that just compressing the original XML documents (*simple* method) does not get the job done. 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 occur

	SIGMOD	INEX	Music	DFT1	DFT2	Element	Frequency	Position	Depth	Overall
No of docs	57	60	34	140	140	80	80	80	80	
tree-edit	1	0	10	56	43	0	0	5	0	15.3%
DFT										
direct ML	0	4	5	55	1	29	9	33	32	22.4%
pairwise ML	0	3	9	53	1	26	0	37	19	19.7%
path shingles										
tags	0	0	1	50	40	0	0	14	48	20.4%
pairwise	0	0	1	46	42	0	0	6	39	17.8%
full path	0	0	1	41	44	0	0	0	29	15.3%
gzip										
simple	1	0	1	39	57	3	30	21	44	26.1%
tags	0	0	1	51	58	0	17	0	6	17.7%
pairwise	0	0	1	45	59	0	25	0	26	20.8%
full path	0	0	1	41	58	0	24	0	3	16.9%
family order	0	0	1	57	62	0	13	9	0	18.9%
Ziv-Merhav										
tags	5	2	1	45	34	0	1	0	0	11.7%
pairwise	0	2	1	43	34	0	8	0	16	13.8%
full path	0	2	1	43	32	0	7	0	0	11.3%
family order	0	0	1	38	39	0	0	2	0	10.6%
Hybrid (DFT/Ziv-Merhav)										
pairw. ML/tags	0	3	9	37	14	0	0	0	3	8.8%
pairw. ML/pairw.	0	3	9	36	16	0	0	0	29	12.4%
pairw. ML/path	0	3	9	35	15	0	0	0	11	9.7%
pairw. ML/family	0	1	8	64	56	0	2	21	9	21.4%

Figure 7: Number of misclusterings for different methods

of a certain type, the better they can be compressed. As soon as a certain pattern appears sufficiently often in a document, it gets an 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-editing algorithms, meaning that the compression-based algorithms have problems clustering documents of the DTD4 type. An oddity for the compression-based techniques was the good performance of the *tags* method and the quite bad performance of the *pairwise* method. We have not been able to find the reasons for this yet. The *family-order* traversal did not bring with it a major improvement 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-order* traversal performed more poorly.

5.4.5 Crossparsing

By crossparsing the documents instead of compressing them and by increasing the granularity from characters to tags, we were able to improve the clustering performance of the entropy-based approaches considerably. Overall, these algorithms are now able to outperform the clustering capabilities of the tree-editing and the DFT algorithms. Three of the variants (*tags*, *pairwise*, and *family-order*) even have linear running time. Combining crossparsing with *family-order* traversal, however, yielded only minimal improvements.

5.4.6 Hybrid Approach

As the crossparsing and DFT algorithms are complementary in their behavior (having strengths and weaknesses in different areas), we decided to combine them in a hybrid approach by multiplying the distances they return. For our own document collections (Element, Frequency, Position, and Depth), the effect of the crossparsing algorithm dominates, while for the real data sets the DFT algorithm seems to dominate. For the DFT data sets the presence of the DFT algorithm kicks in (for DFT2 the result is better than the average of the two original algorithms, while for DFT1 the result is even better than the individual results of the original algorithms). The combination pairwise ML DFT/family-order Ziv-Merhav disappointed somewhat (they are not as complementary as other DFT/Ziv-Merhav combinations, e.g. both have problems clustering the Position data set correctly), while the overall clustering quality for pairwise ML DFT/tags Ziv-Merhav is almost twice as good as that for the tree-editing algorithm, while having a better asymptotic complexity.

5.4.7 Run Time

Measuring the run-time of the different algorithms experimentally is quite difficult, because they were implemented by different people in different programming languages and are not directly comparable, e.g. the actual 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. So we decided to 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 at least the asymptotic run-time complexity of the algorithms can be determined experimentally. We ran the algorithms[10] on a set of shallow documents (ranging in size from 300 up to 102,300 nodes) and on a set

[10]In order not to clutter the graphs, only a selection is shown here. The results of the compression-based algorithms (gzip) are very similar to the crossparsing algorithms, while those for the simple tag variants are very similar to the pairwise variants.

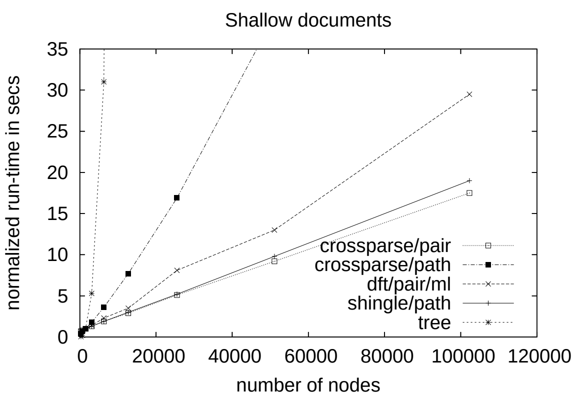


Figure 8: Run-time for shallow documents

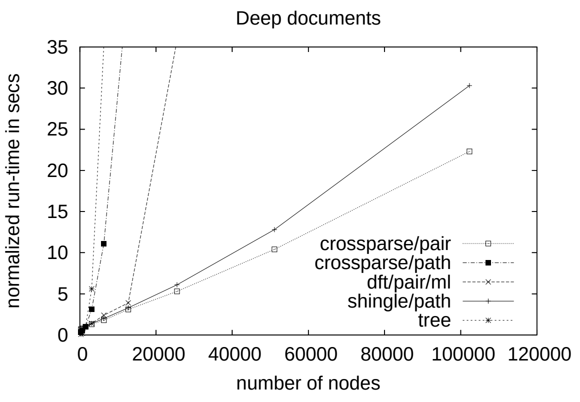


Figure 9: Run-time for deep documents

of deep documents (also ranging in size from 300 to 102,300 nodes).[11] The results for the shallow documents can be seen in Figure 8, while the results for the deep documents are depicted in Figure 9. For the shallow documents the crossparsing/pairwise algorithm and the shingle path algorithm are the clear winners. All other tested algorithms break away from a linear growth sooner or later. For the deep documents the shingle path algorithm also breaks away, since the full path extraction is not linear in the worst case. The crossparsing/pairwise variant, however, is able to hold the linear run time.

6. RELATED WORK

The earliest attempts of detecting structural similarity go back to computing tree-editing distances [29, 30, 32, 34, 36]. These approaches have two major disadvantages: they have at least a quadratic complexity and operate on a node basis, i.e. the basic unit of editing is one node. While Chawathe et al. [10, 11] have enhanced their tree-editing algorithms by subtree editing operations (shedding the latter disadvantage) they have not been able to improve on the quadratic complexity, use heuristics, and in one case work on unordered trees [10]. Nierman and Jagadish developed an algorithm that reflects the properties of XML documents much better [28], but still does not improve the running time. Dalamagas et al. [14] try to improve the performance of the tree-editing approach by using tree structural summaries (eliminating element repetition and nesting) rather than comparing full tree representations. However, from [14] it it not quite clear how the algorithm works for special cases (if there is more than one ancestor node with the same label as a given node) and how costly the preprocessing of the documents to obtain the structural summaries is. By taking a completely different approach based on the analysis of time series via Discrete Fourier Transformation, Flesca et al. were able to improve on the complexity of tree-editing algorithms [16]. In addition to presenting a new shingle-based algorithm, Buttler also gives a brief overview of existing algorithms for detecting document structure similarity [8].

Clustering semistructured documents using a similarity measure forms the basis for algorithms such as [17, 27] extracting schema or document type information. Determining structural similarity is also important for the extraction of information from semistructured data sources [1, 3, 13].

There is also ongoing work on detecting similarity not between XML documents, but between documents and DTDs [7] and comparing unordered XML documents (i.e. datacentric documents for which the order among elements is not relevant) to each other [26]. Finally, there is also a search engine, XXL, employing an ontology similarity measure for retrieving semistructured data semantically [33].

7. 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 a method that has true linear complexity, thus outperforming existing approaches, while being able to keep up with the quality of these approaches. Our technique is based on measuring entropy, which distinguishes it from previously employed methods. In an extensive experimental evaluation we were able to reveal the strengths and weaknesses of the different approaches, demonstrating that the Discrete Fourier Transformation method and our method show quite complementary behavior. Combining these two approaches will yield a very good similarity measure quality.

Future work in this area involves finding more sophisticated ways of describing the structural information of a document before computing the entropy. We would also like to try several other measures for entropy to maybe find one that is better suited for structural information or develop a new entropy measure specifically for structural information.

Acknowledgments

We want to give special thanks to Andy Nierman and Giuseppe Manco for providing us with executables for the tree-editing and the DFT algorithm, respectively. We also want to thank Peter Wood for pointing out the family-order traversal algorithm. Thanks also goes out to the anonymous referees for their helpful hints on improving this paper.

[11]The DTDs of the documents can be found in the appendix.

8. REFERENCES

[1] A. Arasu and H. Garcia-Molina. Extracting structured data from web pages. In *ACM SIGMOD Int. Conf. on Management of Data*, pages 337–348, 2003.

[2] R.A. Baeza-Yates and B.A. Ribeiro-Neto. *Modern Information Retrieval.* Addison Wesley, 1999.

[3] 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.

[4] M. Benedikt, C.-Y. Chan, W. Fan, J. Freire, and R. Rastogi. Capturing both types and constrints in data integration. In *ACM SIGMOD Int. Conf. on Management of Data*, pages 277–288, 2003.

[5] 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.

[6] P.A. Bernstein, J. Madhavan, and E. Rahm. Generic schema matching with Cupid. In *Int. Conf. on Very Large Databases (VLDB'01)*, pages 49–58, 2001.

[7] 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.

[8] D. Buttler. A short survey of document structure similarity algorithms. In *5th Int. Conf. on Internet Computing*, Las Vegas, Nevada, 2004.

[9] 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.

[10] 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.

[11] 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.

[12] R. Cilibrasi and P.M.B Vitányi. Clustering by compression. *IEEE Transactions on Information Theory*, 51(4):1523–1545, April 2005.

[13] 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.

[14] 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.

[15] 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.

[16] 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.

[17] 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.

[18] 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.

[19] N. Jardine and R. Sibson. *Mathematical Taxonomy.* John Wiley and Sons, New York, 1971.

[20] D. Knuth. *The Art of Computer Programming, Volume I: Fundamental Algorithms.* Addison Wesley, 1973.

[21] N.J. Larsson. Extended application of suffix trees to data compression. In *Proceedings Data Compression Conference*, pages 190–199, 1996.

[22] 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.

[23] M. Levene and P.T. Wood. XML structure compression. In *2nd Int. Workshop on Web Dynamics*, Honolulu, Hawaii, 2002.

[24] M. Li and P.M.B Vitányi. *An Introduction to Kolmogorov Complexity.* Springer, New York, 1997.

[25] A. Martins. String kernels and similarity measures for information retrieval. Technical report, Priberam, Lisbon, Portugal, 2006.

[26] 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.

[27] 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.

[28] 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.

[29] S. Selkow. The tree-to-tree editing problem. *Information Processing Letters*, 6(6):184–186, 1977.

[30] D. Shasha and K. Zhang. *Pattern Matching in Strings, Trees, and Arrays*, chapter Approximate Tree Pattern Matching. Oxford University Press, 1995.

[31] P.H.A. Sneath and R.R. Sokal. *Numerical Taxonomy.* Freeman, San Francisco, 1973.

[32] K.C. Tai. The tree-to-tree correction problem. *Journal of the ACM*, 26(3):422–433, 1979.

[33] A. Theobald and Weikum G. 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.

[34] 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.

[35] I.H. Witten, A. Moffat, and T.C. Bell. *Managing Gigabytes.* Morgan Kaufmann, San Francisco, 1999.

[36] 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.

[37] J. Ziv and A. Lempel. A universal algorithm for sequential data compression. *IEEE Transactions on Information Theory*, 23(3):337–343, 1977.

[38] 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.

APPENDIX

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>
]>
```

B. DTDS FOR RUN-TIME EXPERIMENTS

The following two DTDs were used in the run-time experiments from Section 5.4.7. The DTD for the shallow documents was:

```
<!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 SHALLOW [
  <!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.