Predictive Modeling in XML Compression

Olli Luoma & Jukka Teuhola
Department of Information Technology and Turku Centre for Computer Science
20014 University of Turku, FINLAND
{olli.luoma, jukka.teuhola}@it.utu.fi

Abstract

Since its advent, the Extensible Markup Language (XML) has gained tremendous popularity in many different application areas. However, XML data is generally very verbose and redundant, and thus it requires a lot of disk space to store and bandwidth to transfer. To overcome this problem, many methods for compressing XML documents have been proposed. In general, data compression requires a model which is used to predict the next symbol in the data. In this paper, we compare different models suitable for XML compression. We also present a novel modeling method and measure the information content in a set of XML documents using different modeling methods.

1. Introduction

Data compression can be regarded as a process which transforms a stream of symbols into a stream of codes. In general, codes can be assigned to single symbols or larger coding units. In order to actually compress the data, the stream of codes should be more compact than the stream of symbols, which can obviously be achieved by using fewer bits for frequent symbols and more bits for rare ones. This requires a *model* which is capable of predicting the next symbol in the stream as accurately as possible. The purpose of the *encoder*, then, is to assign codes for the source stream based on the information provided by the model. The former stage of the process is generally referred to as *modeling* and the latter as *coding*. The coding stage has been well studied and methods for producing optimal codes are known [6, 11, 12]. Modeling, on the contrary, depends on the nature of the compressed data, and thus this part of data compression still offers interesting research problems.

A predictive model bases its predictions on a *context*, i.e., a set of previously appeared symbols. For each context, a model provides the probability distribution of the followers. The number of previously appeared symbols taken into account is usually referred to as the *order* of the model. A model of order 1 therefore takes only one previously appearing symbol into account, whereas a model of order 0 considers the global probability distribution of the symbols. One could also define a context of order -1 in which all symbols are equally probable. In general, increasing the order of the model, i.e., taking more previously appeared symbols into account, increases the accuracy of the predictions thus leading to shorter codes and better compression ratio.

In this paper, we focus on the problem of compressing XML data, i.e., data marked up using a meta-language recommended by the World Wide Web Consortium [14]. As mentioned, the coding stage is generally well known, and thus our focus is on predictive modeling. In the case of XML compression, this problem is especially interesting since the hierarchical nature of XML provides us with many different modeling alternatives which cannot be applied on flat text. A model could, for example, take advantage of the observation that certain symbols are more likely to appear between or after certain tags and less likely to appear between or after some others. We also propose a novel modeling method which is suitable for partial decompression and provides accurate predictions.

The rest of this paper is organized as follows. In Section 2, we briefly review the related work and in Section 3, different modeling methods including our own proposal are discussed. We then move on to our experimental results in Section 4 and conclude this paper in Section 5.

2. Related work

Because of the popularity of XML and because of its inherently verbose nature a plethora of methods for compressing XML data have been proposed. Some proposals, such as XGRIND [13], XPRESS [10], XQZip [5], and XSeq [9], support queries, whereas others, such as XMill [8] and XMLPPM [3], do not. The methods in the former category usually aim at supporting some subset of XPath [15], an XML query language which can essentially be used to select parts of XML documents. One might, for example, want to decompress all parts of the documents which ap-

pear between <abstract> and </abstract> tags. In the latter group, partial decompression is not desired, and thus only compression time and *compression ratio*, that is, the size of the compressed document vs. the size of the original document, are the determining factors.

In our view, however, most of the previous work has blurred the distinction between modeling and coding to some extent. It is therefore very hard to say which part of their claimed performance is due to the use of more efficient coding methods, e.g., using arithmetic coding instead of Huffman coding, and which is due to modeling. Nevertheless, there are some papers which concentrate on modeling in the case of hierarchical data compression. Cheney [4], for instance, discussed different modeling methods but his work dismissed the possibility of partial decompression. One should keep in mind, however, that standard text compressors usually perform very well also on XML data, and thus it makes more sense to concentrate on partial decompression than aim at marginally improving the compression ratio.

3. Modeling methods

In what follows, an XML document is treated as an *XML tree*, a labeled, partially ordered tree in which each node corresponds to an element, attribute, or piece of text in the document. We use special ending nodes with empty tags </> in order to preserve the nested structure of the document and consider names of element and attribute nodes as extensions of our alphabet. Characters in text nodes or in attribute values are treated as individual symbols. Fig. 1 illustrates such a tree representation for the following document:

```
<books>
  <book isbn="3540285830">
    <name>XML Technologies and
Applications</name>
    <publisher>Springer</publisher>
  </book>
  <book isbn="0672315149">
    <name>XML Unleashed</name>
    <publisher>Sams</publisher>
  </book>
</books>
```

3.1 Preorder

The most straightforward way of defining the context for XML trees is to use the *preorder* (depth-first order) of the nodes. In preorder traversal, a node is visited before its subtrees are recursively visited from left to right. This is also the order in which the symbols actually appear in the physical document, and thus preorder is often also referred to as *document order*. Since this is also the order in which a SAX parser encounters the symbols the approach is very easy to implement. For the same reason, the preorder model suits well for *adaptive* compression, i.e., a compression scheme in which the model is constructed in the fly. This modeling method considers the XML document in Fig. 1, for example, simply as the following stream of symbols: <books>, <book>, <@isbn>, 3, 5, 4, 0, 2, 8, 5, 8, 3, 0, </>, <name>, X, M, L, ...

Using a model of order 2, for example, we have two equally probable symbols in context <publisher>,S, i.e., symbols p and a. Thus, encoding each of these symbols takes just one bit. Context </>,<publisher>, on the other hand, is always followed by symbol S, and thus no bits at all are needed to encode symbols in this context[1]. Based on the information provided by the model, the decoder always outputs symbol S when context </>,<publisher> is encountered. Initially, the context can be filled with special <null> symbols, and thus the context for symbol <books> is <null>,<null> and for the first <book> symbol <null>,<books>. However, defining the context using the preorder of the nodes is somewhat cumbersome if partial decompression is desired. This is due to the fact that in order to construct a model to transform a code into a symbol, the whole preceding document has to be decompressed. Thus, it is hard to skip subtrees during decompression. Of course, general methods aimed at partial decompression, such as the proposal by Lekatsas and Wolf [7], can still be applied in order to support partial decompression to some extent.

3.2 Level-order

In level-order (breadth-first order) traversal, nodes on level i are visited from left to right before nodes at level $i + 1$ are visited from left to right. The tree presented in Fig. 1, for instance, would be traversed in the following order: <books>, <book>, <book>, </>, <@isbn>, <name>, <publisher>, </>, <@isbn>, ... However, the method is not easy to implement since in order to predict the leftmost node of, say, level 5, we obviously need to know the rightmost node of level 4. A practical approach would therefore be to reorder the symbols from preorder into level-order before actually performing the compression, which limits the use of the approach in an adaptive setting. Conversely, of course, the symbols have to be ordered back into preorder after decompression. For the same reason, it is very tricky to support partial decompression when this modeling method is used.

[1] This, of course, assumes a semi-adaptive compression scheme in which the model is transferred with the compressed data.

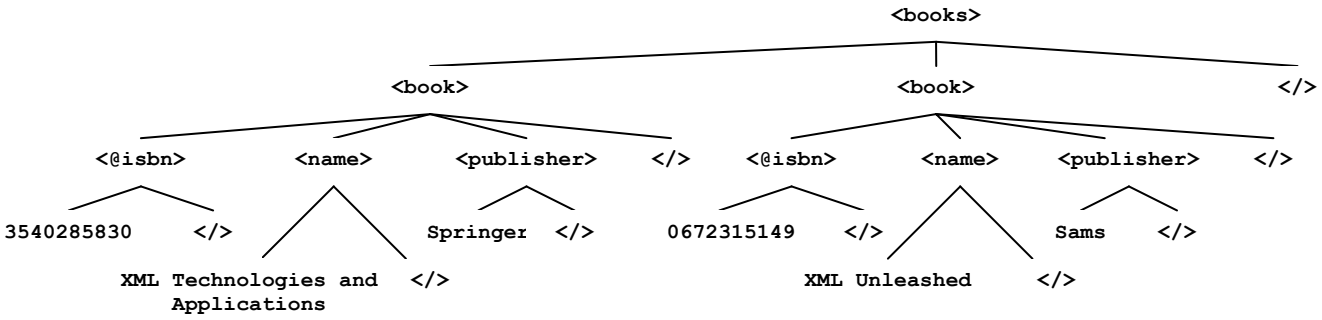

Figure 1. A tree representation of an XML document

3.3 Children-depth-first

In [1], an interesting order called *children-depth-first* for traversing a tree was introduced. In this approach, a node and its siblings are visited from left to right before recursively visiting the children of the node and the children of its siblings from left to right. If applied to the tree presented in Fig. 1 this approach would yield the following symbol sequence: `<books>`, `<book>`, `<book>`, `</>`, `<@isbn>`, `<name>`, `publisher`, `</>`, `3`, `5`, `4`, `0`, ... In other words, this method can be regarded as a hybrid between preorder and level-order. However, the children-depth-first approach suffers from similar drawbacks as the level-order approach since in order to compress the leftmost node of a set of siblings on level $i+1$, its rightmost "uncle" on level i is needed to build the context.

3.4 Ancestor path

One option is to predict a symbol based on its ancestors in the tree. This method can be easily implemented by maintaining a stack of ancestors during compression and decompression. In the context `<book>,<publisher>`, for example, we now have symbols `S` and `</>` each appearing with probability 2/16 and symbols `p`, `r`, `i`, `n`, `g`, `e`, `r`, `a`, `m`, `s` each appearing with probability 1/16. Since only element nodes and attribute nodes can now be part of the context the contextual information available in the text nodes is lost. Moreover, this approach dismisses the order of the nodes, and thus its ability to predict element and attribute nodes is also quite limited. However, the ancestor path approach provides us with the possibility of partial decompression, and thus it has been followed in some XML compressors.

3.5 Left-sibling-parent

Our own context proposal, *left-sibling-parent*, supports partial decompression but can still use the context information available in the text nodes and provide accurate predictions. For any node, the left-sibling-parent predecessor is defined as the left sibling of the node if a left sibling exists and as the parent of the node otherwise. Initially, the context can again be filled using `<null>` symbols. For example, in the tree presented in Fig. 1, the context `<book>,<@isbn>` is followed by symbols `3` and `0` each appearing with probability 1/4 and symbol `<name>` appearing with probability 1/2. By applying Huffman coding, for example, symbols `3` and `0` can therefore be coded with 00 and 01, whereas `<name>` could be coded with 1. Context `<name>,X`, on the other hand, is always followed by symbol `M` and context `X,M` by symbol `L`. As a larger example, the "maximal" left-sibling-parent context for symbol `a` in `Sams` is `<books>`, `<book>`, `<book>`, `<@isbn>`, `<name>`, `<publisher>`, `S`.

In order to support partial decompression, we could encode each subtree as a 3-tuple $(name, length, codes)$ where $name$ is a code for the name of the root of the subtree, $length$ the length of the $codes$ part in bits, and $codes$ simply a (possibly empty) compressed representation of the children of the root, i.e, a set of similar 3-tuples. If $name$ is not the searched tag we simply skip the next $length$ bits in the compressed representation before continuing the decompression. Otherwise, we continue decompressing $codes$. This approach can obviously only support XPath queries generated using name tests and the `child` axis. One should notice, however, that the `descendant` and `descendant-or-self` axes or containment queries could also be supported by adding information about the different tags appearing in the subtree into the representation of subtrees. It is also worth noticing that the context information can be used to predict not only $name$ but also $length$ and - in the case of containment query support - the set of tags appearing in the subtree. A detailed analysis of this alternative, however, is beyond the scope of this paper.

4. Experimental results

This section presents the results of our experiments. For simplicity, we assume a *semi-adaptive* compression scheme, i.e., we assume that the model is built before the actual encoding. Thus, two passes through the document are needed and the model has to be passed on with the compressed data. One should, however, notice that this is the only practical possibility if partial decompression is desired. In order to support partial decompression, we have to be able to skip the compressed representations of some subtrees which obviously requires encoding subtree sizes. However, since the root of a subtree appears before the subtree in an XML document the subtree size cannot be encoded before visiting the subtree. Of course, if partial decompression is not aimed at one can resort to adaptive compression. It is our belief that our results of predictive power of different context models are applicable also in the case of adaptive compression.

In our tests, we used the following set of XML documents:

- Document `allelements.xml` available at [16] is the periodic table of chemical elements marked up in XML. The document is very shallow with some structural variation. The content consists mainly of numbers and the text nodes are small, i.e., they contain a small amount of characters.
- In `sigmodrecord.xml` available at [17], the text nodes are larger than in `allelements.xml`. The structure is again simple and the amount of structural variation is small.
- Document `dream.xml` available at [16] is Shakespeare's *A Midsummer Night's Dream* marked up in XML. In this case, the document consists mostly of textual content and the structure is very simple.

To start with, we measured the information content of these documents using the modeling methods discussed in Section 3. The names of the element and attribute nodes were treated as single symbols and ending tags were treated as discussed earlier. The *information content* for symbol s_i appearing in context $s_{i_1}, \ldots, s_{i_n}$ denoted as $I(s_i|s_{i_1}, \ldots, s_{i_n})$ was defined as usual, i.e.,

$$I(s_i|s_{i_1}, \ldots, s_{i_n}) = log_2 \left(\frac{1}{P(s_i|s_{i_1}, \ldots, s_{i_n})} \right)$$

where n denotes the order of the model and $P(s_i|s_{i_1}, \ldots, s_{i_n})$ denotes the conditional probability of symbol s_i appearing in context $s_{i_1}, \ldots, s_{i_n}$. The average information content per symbol, i.e., the *entropy*, then, was simply defined as the mean of the information contents of the symbols.

The results are presented in Fig. 2, Fig. 3 and Fig. 4. In our results, the preorder method is referred to as PRE, ancestor path as ANC, level-order as LVL, children-depth-first as CDF, and left-sibling-parent as LSP. The entropy was measured as bits per symbol with tags treated as single symbols; *model size* was simply defined as the number of different contexts. It should be emphasized that the results are not actual compression results but rather measured entropies of the documents. One cannot therefore expect to reach actual compression ratios close to these figures. In general, the size of the model was roughly inversely proportional to the accuracy of the predictions, i.e., the average information content of a symbol. There were, however, some interesting details in our results. In most cases, LVL was asymptotically able to produce the most accurate predictions, which suggests that shuffling the parts of a document in level-order before compression should lead to gains in compression ratio. As expected, the performance of ANC was clearly the worst in all cases. Finally, one should notice that our own proposal LSP is competitive especially in the case of short contexts. Furthermore, LSP seems to provide a good tradeoff between the size of the model and accuracy of the predictions, and thus it might be a viable option for adaptive compression in which memory utilization is an important factor [4].

To study the difference between PRE and LVL further, we compared these methods by applying the popular `gzip` and `bzip2` compressors to the documents ordered in preorder and level-order[2]. Since `gzip` and `bzip2` are also used as actual compressors in XMill, an XML-conscious compressor presented in [8], we also included the results obtained using XMill with the same implementations of `gzip` and `bzip2`. The results of these tests are presented in Table 4; all values represent the compression ratio as bits in the compressed data per character in the original document. In the case of `gzip`, XMill was able to outperform both LVL which was able to outperform PRE. This supports our earlier observation of the better prediction ability of LVL. However, when `bzip2` was used, the approaches performed much more evenly and PRE actually marginally outperformed both LVL and XMill. This is very probably due to the fact that `bzip2` performs Burrows-Wheeler transformation [2] prior to compression. Thus, when the document reaches the compressor it is no longer ordered in preorder or level-order. Nevertheless, the results obtained using `bzip2` are very interesting since they show that the practicability of XMill is somewhat questionable.

[2] Strictly speaking, the order was not level-order since attributes had to be maintained in conjunction with the starting tags.

Document	gzip XMill	gzip LVL	gzip PRE	bzip2 XMill	bzip2 LVL	bzip2 PRE
`allelements.xml`	0.4376	0.5736	0.5872	0.4112	0.4256	0.4184
`sigmodrecord.xml`	1.1248	1.2584	1.5072	0.9784	0.9344	0.9480
`dream.xml`	2.2768	2.2816	2.3456	1.9432	1.8344	1.8096
Average	1.2797	1.3712	1.4800	1.1109	1.0648	1.0587

Table 1. The results concerning the tests with `gzip` and `bzip2` in bits per character

5 Concluding remarks

XML data is inherently redundant in nature and therefore very amenable to compression. Although standard text compressors do a rather good job, they are restricted to preorder when it comes to defining the context. However, since the hierarchy of XML documents involves more structural information and correlations than flat text there is a challenge to find better contexts for elements, attributes, and characters. Thus, we experimented with several different models comparing their prediction power and model size. Moreover, we introduced a novel modeling method, a so called left-sibling-parent context, which is capable of supporting partial decompression and yet provides accurate predictions. One interesting detail in our experiments was that the level-order context could outperform the preorder context. The margin was, however, quite narrow, and thus we are more interested in supporting partial decompression of XML documents compressed using the left-sibling-parent method.

References

[1] J. Banerjee, W. Kim, S.-J. Kim, and J. F. Garza. Clustering a DAG for CAD databases. *IEEE Transactions on Software Engineering*, 14(11): 1684-1699, 1988.

[2] M. Burrows and D. J. Wheeler. A block-sorting lossless data compression algorithm. Digital Systems Research Center Research Report 124, 1994.

[3] J. Cheney. Compressing XML with multiplexed hierarchical models. In *Proc. 2001 IEEE Data Compression Conf.*, pages 163-172, 2001.

[4] J. Cheney. Tradeoffs in XML database compression. In *Proc. 2006 IEEE Data Compression Conf.*, pages 392-401, 2006.

[5] J. Cheng and W. Ng. XQZip: Querying compressed XML using structural indexing. In *Proc. 9th Intl Conf. on Extending Database Technology*, pages 219-236, 2004.

[6] D. A. Huffman. A method for the construction of minimum redundancy codes. In *Proc. IRE*, pages 1098-1102, 1952.

[7] H. Lekatsas and W. Wolf. Random access decompression using binary arithmetic coding. In *Proc. 1999 IEEE Data Compression Conf.*, pages 306-315, 1999.

[8] H. Liefke and D. Suciu. XMill: An efficient compressor for XML data. In *Proc. 2000 ACM SIGMOD Intl Conf. on Management of Data*, pages 153-164, 2000.

[9] Y. Lin, Y. Zhang, Q. Li, and J. Yang. Supporting efficient query processing on compressed XML files. In *Proc. 2005 ACM Symposium on Applied Computing*, pages 660-665, 2005.

[10] J.-K. Min, M.-J. Park, and C.-W. Chung. XPRESS: A queriable compression for XML data. In *Proc. 2003 ACM SIGMOD Intl Conf. on Management of Data*, pages 122-133, 2003.

[11] R. Pasco. *Source Coding Algorithms for Fast Data Compression*. PhD thesis, Stanford University, 1976.

[12] J. J. Rissanen and G. G. Langdon. Generalized Kraft inequality and arithmetic coding. *IBM Journal of Research and Development*, 20: 198-203, 1976.

[13] P. M. Tolani and J. R. Haritsa. XGRIND: A query-friendly XML compressor. In *Proc. 18th Intl Conf. on Data Engineering*, pages 225-234, 2001

[14] W3C. Extensible Markup Language (XML) 1.0. `http://www.w3c.org/TR/REC-xml/`

[15] W3C. XML path language (XPath) 2.0. `http://www.w3c.org/TR/xpath20/`

[16] `http://www.ibiblio.org/xml/examples/`

[17] `http://www.cs.washington.edu/research/xmldatasets/www/`

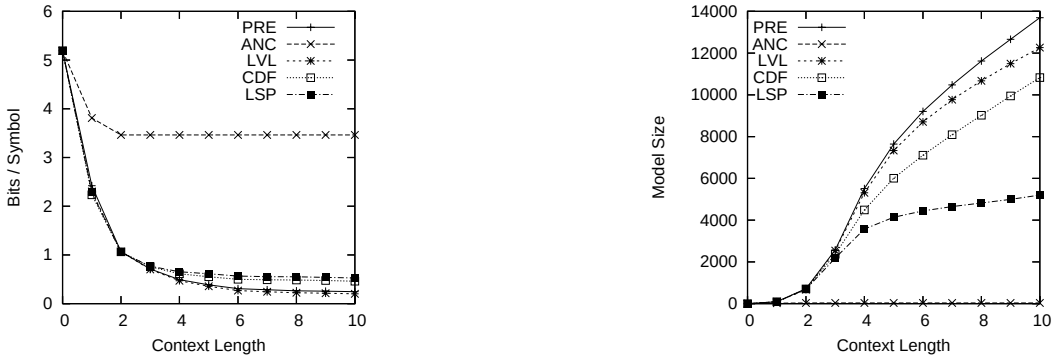

Figure 2. The measured entropies (left) and model sizes (right) for `allelements.xml`

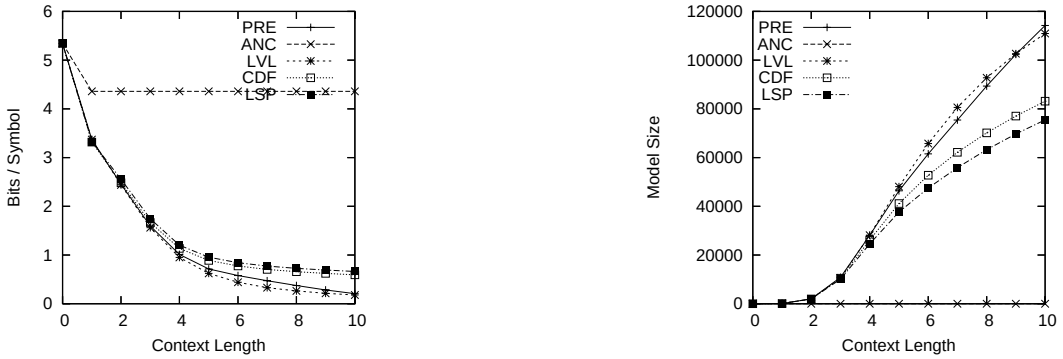

Figure 3. The measured entropies (left) and model sizes (right) for `sigmodrecord.xml`

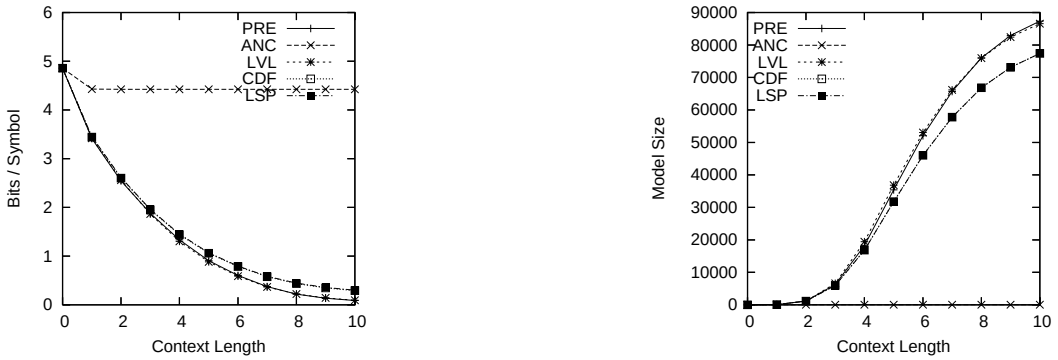

Figure 4. The measured entropies (left) and model sizes (right) for `dream.xml`