XML Structure Compression

Mark Levene and Peter Wood
Birkbeck College, University of London
London WC1E 7HX, U.K.
{m.levene,p.wood}@dcs.bbk.ac.uk

Abstract

XML is becoming the universal language for communicating information on the Web and has gained wide acceptance through its standardisation. As such XML plays an important enabling role for dynamic computation over the Web. Compression of XML documents is crucial in this process as, in its raw form, it often contains a sizable amount of redundancy. Several XML compression algorithms have been proposed but none make use of the DTD when it is available. Here we present a novel compression algorithm for XML documents that conform to a given DTD, that separates the document's structure from its data, taking advantage of the regular structure of XML elements. Our approach seems promising as we are able to show that it minimises the length of encoding under the assumption that document elements are independent of each other. Our presentation is a preliminary investigation; it remains to carry out experiments to validate our approach on real data.

1 Introduction

Extensible Markup Language (XML; www.w3c.org/XML/) [GP01] is the universal format for structured documents and data on the Web. With the vision of the semantic Web [BLHL01] becoming a reality, communication of information on the machine level will ultimately be carried out through XML. As the level of XML traffic grows so will the demand for compression techniques which take into account the XML structure to increase the compression ratio.

The ability to compress XML is useful because XML is a highly verbose language, especially regarding the duplication of meta-data in the form of *elements* and *attributes*. A simple solution would be to use known text compression techniques [BCW90] and pipe XML documents through a standard text compression tool such as `gzip` (www.gzip.org/) or `bzip2` (sourceware.cygnus.com/bzip2/index.html). The problems with this approach are twofold: firstly, compression of elements or attributes may be limited by existing tools due to the long range dependencies between elements and between attributes, i.e. the duplication is not necessary local, and secondly, to enhance compression, it may be useful to use different compression techniques on different components of XML.

A simple idea to improve on just using standard text compression tools is to use a symbol table for XML elements and attributes prior to piping the result through `gzip` or `bzip2`. We are aware of two XML compression systems XMILL [LS00] and XMLPPM [Che01] that attempt to further improve on this idea. The idea behind XMILL is to transform the XML into three components: (1) elements and attributes, (2) text, and (3) document structure,

and then to pipe each of these components through existing text compressors. Another XML compression system, XMLPPM, refines this idea further by using different text compressors with different XML components, i.e. one model for element and attribute compression and another for text compression, and, in addition, it utilises the hierarchical structure of XML documents to further compress documents.

We are aware of one previous compression algorithm for XML that uses the knowledge encapsulated in a *Document Type Definition* (DTD) [GP01]. The proposed method, called differential DTD compression [SM01], claims to encode only the information that is present in an XML document but not in its DTD. In this sense their approach is the same as the one we present here, but as opposed to [SM01] we concentrate on a particular algorithm, independently discovered, and its detailed analysis.

To simplify the presentation, in this paper we consider DTDs which define only elements, rather than allowing the definition of attributes and entities as well. The compression techniques and algorithms could be adapted to cater for these additional components. As a result, all XML documents in this paper comprise only occurrences of elements.

Prior to explaining our compression algorithm we briefly introduce the XML concepts we use via an example.

Example 1.1 Consider the following DTD D which provides a simplistic representation for the contents of books:

```
<!ELEMENT book       (author, title, chapter+) >
<!ELEMENT chapter    (title, (paragraph|figure)+) >
<!ELEMENT author     (#PCDATA) >
<!ELEMENT title      (#PCDATA) >
<!ELEMENT paragraph  (#PCDATA) >
<!ELEMENT figure     (EMPTY) >
```

Each element definition comprises two parts: the left side gives the name of the element, while the right side defines the *content model* for the element. The content model is defined using a regular expression built from other element names.

The six content models defined above are interpreted as follows. A `book` element has an `author` element as first child, a `title` element as second child, and one or more `chapter` elements as further children. A `chapter` element must have a `title` element followed by one or more `paragraph` or `figure` elements. Elements `author`, `title` and `paragraph` contain simply text, while `figure` elements are empty (presumably with data for the figure provided by an attribute, which we do not consider here).

The following XML document d is *valid* with respect to (or conforms to) the DTD D:

```
<book>
  <author>Darrell Huff</author>
  <title>How to Lie with Statistics</title>
  <chapter>
    <title>Introduction</title>
    <figure ... />
    <paragraph>With prospects of ...</paragraph>
```

```
    <paragraph>Then a Sunday newspaper ...</paragraph>
    [ 8 more paragraphs ]
  </chapter>
  <chapter>
    <title>The Sample with the Built-in Bias</title>
    [ 53 paragraphs and 7 figures ]
  </chapter>
</book>
```

The parse tree of document d with respect to DTD D, denoted by PARSE(d, D) is shown in Figure 1. The nodes in PARSE(d, D) correspond to the elements of the XML document, d, and operators from the regular expressions used in content models in D. We call the latter category of nodes *structure nodes*. In particular, in Figure 1, there are three types of structure nodes: nodes labelled by '+' are *repetition* nodes, nodes labelled by ',' are *sequence* nodes, and nodes labelled by '|' are *decision* nodes.

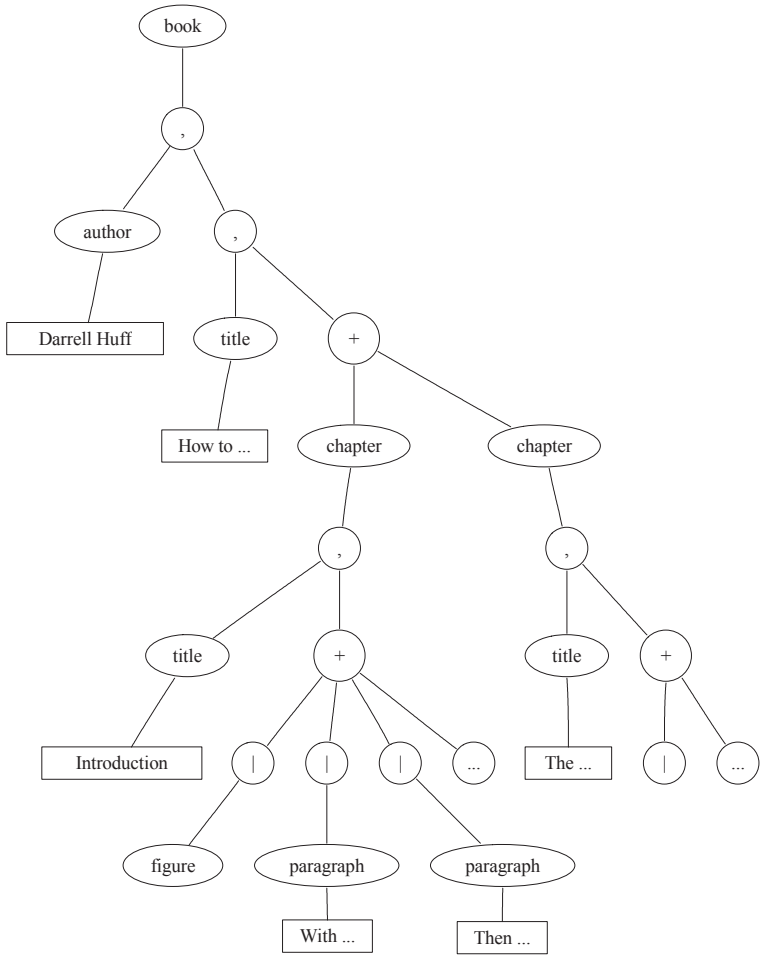

Figure 1: The parse tree for the XML document of Example 1.1

From now on we will assume that each XML document we deal with is valid with respect

to a given DTD D. We now give a conceptual overview of the compression algorithm we have devised. We split the compression of an XML document, say d, into two parts. First we obtain the parse tree PARSE(d, D) (see Figure 1) and prune from it all the leaf nodes containing text; we call the sequence of leaf nodes in a left-to-right fashion with a fixed delimiter between them, *the data*.

In the second step, we apply further pruning to the parse tree maintaining a tree representation only of the structure that needs to be encoded in order that the decoding can reconstruct the document given DTD D; we denote the resulting tree by PRUNE(d, D). Figure 2 shows the pruned tree corresponding to the parse tree of Figure 1 (note that, in this example, the pruned parse tree does not contain sequence nodes since they can be deduced from DTD D). We can then encode PRUNE(d, D) using a breadth-first traversal, in such a way that each repetition node is encoded by a number of bits, say B, encoding the number of children of the repetition node, and each decision node is encoded by a single bit, which may be 0 or 1 according to its child; we call the resulting output *the encoding*.

The algorithm presented in Section 2 does not explicitly construct PRUNE(d, D). Instead it traverses PARSE(d, D) while generating the encoding. However it is effectively only those nodes of PRUNE(d, D) which cause the algorithm to generate any output.

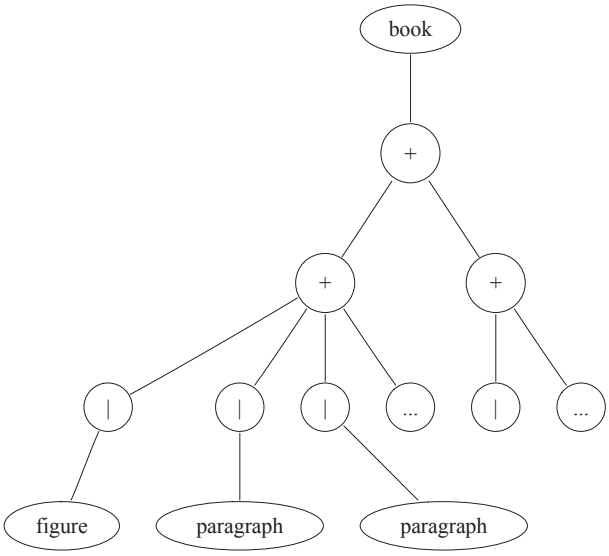

Figure 2: The pruned parse tree for the XML document of Example 1.1

The compression of the document thus contains three elements: (1) the DTD, which is fixed, (2) the encoding of the document's structure given the DTD, and (3) the textual data contained in the document given the DTD. These outputs can be compressed further by piping them through standard text compression tools. We now give an example to illustrate our algorithm.

Example 1.2 In order to encode the structure of the document d given in Example 1.1, we observe that every `book` must have an `author` and `title` as its first two children, so there is no need to encode this information, since the DTD D will be known to the decoder. All that needs to be encoded for children of `book` is the number of `chapter` elements present in the

document. This is suggested by the top repetition node labelled with '+' in Figure 2. In this case, there are two `chapter` elements.

For each `chapter` element in d, we need to encode the number of `paragraph` and `figure` elements which occur. This is suggested by the lower two repetition nodes labelled with '+' in Figure 2. Since arbitrary sequences of `paragraph` and `figure` elements are permitted, all we can do in the encoding is to list the actual sequence which occurs, simply encoding the element names. This is suggested by the decision nodes labelled with '|' and their children in Figure 2. Such an encoding is no better than if there were no DTD, but on the other hand we cannot possibly do any better (unless we ignore the order of elements). We can encode the occurrence of a `paragraph` in d by 0 and that of a `figure` by 1. Thus the encoding for the first chapter would be

11 1 0 0 0 0 0 0 0 0 0 0

where 11 is the number of `paragraph` and `figure` elements represented in decimal. The second `chapter` requires 60 bits to represent the sequence of `paragraph` and `figure` elements.

In order to provide an analysis of our algorithm we turn to information theory [Rez94] and the concept of *entropy* (or uncertainty), which implies that a minimum encoding of a message is a function of the likelihood of the message. Our algorithm is in the spirit of the two-part *Minimum Description Length* (MDL) encoding [HY01], which is based upon the idea of choosing the model that minimises the sum of the lengths of the encodings of (1) the model, and (2) the data encoded given the chosen model. In our case the DTD provides us with a model for the data, i.e. for the XML document, which we then use to encode the document's structure. Then, having this encoding available we can transmit the document structure and the actual data separately. We observe that if we view the DTD as an equivalence class of models, such that the document is valid with respect to all members of the class, then MDL encoding could be used to choose the preferred model.

The techniques we present here are also inspired by work on using DTDs to optimise queries on XML repositories [Woo00, Woo01]. Constraints present in DTDs can be used to detect redundant subexpressions in queries. For example, assume that a DTD implies that every `date` element which has a `day` element as a child must also have `month` and `year` elements as children. Now a query (on a set of documents valid with respect to the DTD) which asks for `date` elements which have both a `day` element and a `month` element as children is equivalent to one which asks for `date` elements which just have a `day` element as a child.

The rest of the paper is organised as follows. In In Section 2 we present the detail of our minimum length encoding algorithm. In Section 3 we provide some analysis of our algorithm, and finally, in Section 4 we give our concluding remarks.

2 Minimum Length Encoding of XML Document Structures

Given an XML document d and DTD D, recall that PARSE(d, D) is the parse tree of d with respect to the DTD D. Now let STRUCT(d) be the tree representation of d with the leaf nodes containing text pruned from it. In this section we define encoding and decoding algorithms for STRUCT(d), assuming that d is valid with respect to DTD D. The encoding algorithm takes PARSE(d, D) as input and produces a minimal length encoding ENCODING(d, D)

of PARSE(d, D). The decoding algorithm takes ENCODING(d, D) and D as input and reconstructs STRUCT(d).

The encoding and decoding algorithms operate in breadth-first order, considering the children of each element from the root in turn. The following example, along with Examples 1.1 and 1.2, illustrates the essence of one such step of the encoding algorithm.

Example 2.1 Consider the following DTD (with simplified syntax):

```
bookstore ((book|magazine)+)
book      (author*, title?, date, isbn)
author    (((first-name|first-initial), middle-initial?)?, last-name)
magazine  (title, volume?, issue?, date)
date      ((day?, month)?, year)
```

This DTD uses two postfix operators not found in the DTD of Example 1.1: `*` represents zero or more occurrences of the preceding expression, while `?` represents that the preceding expression is optional. Occurrences of the operator `*` give rise to repetition nodes in the parse tree of a document, while those of operator `?` give rise to decision nodes.

The encoding for the sequence of children of an element with name n in the document is based on how that sequence is parsed using the regular expression in the content model for n in the DTD. Consider the children of a `date` element. If there were only a `year` element as a child, then the encoding is simply 0, reflecting the fact that the optional subexpression `(day?, month)?` was not used in parsing. On the other hand, if `date` has both `year` and `month` children, then the encoding is 10, reflecting the fact that the subexpression `(day?, month)?` *was* used in the parsing (hence 1), but that the `day?` subexpression was not used (hence 0). For `day`, `month` and `year` children, the encoding would be 11. Hence the maximum length of the encoding is 2 bits, which is the shortest possible for representing the three possible sequences of children for `date`: `(day, month, year)`, `(month, year)` and `(year)`.

At the other extreme, the content model for `bookstore` allows for any sequence of children (over the alphabet of `book` and `magazine`) whatsoever (apart from the empty sequence). Thus, as for `paragraph` and `figure` elements in Example 1.1, in the encoding all we can do is to list the actual sequence which occurs, simply encoding the element names using 0 for `book` and 1 for `magazine`.

For encoding the children of a `book` element, we encode the number of `author` occurrences followed by 1 or 0 indicating whether or not `title` occurs.

The encoding algorithm, called ENCODE-STRUCTURE, is shown in Figure 3. The algorithm makes use of a procedure called ENCODE which is shown in Figure 4. The algorithm takes as input the parse tree, PARSE(d, D), of a document d with respect to DTD D, and produces ENCODING(d, D), the encoding of STRUCT(d) with respect to D.

As shown in Figure 1, nodes labelled with the operator | in PARSE(d, D) have a single child which is the root of a parse tree for either the left-hand or right-hand operand of | in the regular expression in which it appears (unless the operand which is used in the parsing is ϵ, in which case there will be no child). The encoding algorithm assumes that, rather than using the operator | in PARSE(d, D), the operators $|_L$ and $|_R$ are used, where L (respectively,

algorithm ENCODE-STRUCTURE:
Input: PARSE(d, D)
Output: ENCODING(d, D)
Method:

```
begin
    /* Assume we have a single queue available,
    with operations enqueue(node) and dequeue() */
    let i be the root node of PARSE(d, D);
    enqueue(i);
    while queue not empty do ENCODE(dequeue())
end
```

Figure 3: Algorithm for encoding an XML tree.

R) indicates that the child of the operator node corresponds to the left-hand (respectively, right-hand) operand in the corresponding regular expression.

In the construction of PARSE(d, D), we assume that operators '`*`', '`+`' and '`?`' are left-associative and have the highest precedence. The operators '`,`' and '`|`' are right-associative, with '`,`' having higher precedence than '`|`'.

The encoding algorithm also assumes that the document type of the document d being encoded corresponds to the first element name defined in the DTD D. Thus the document type of any document being encoded with respect to the DTD of Example 2.1 is assumed to be `bookstore`. A trivial addition to the encoding algorithm can overcome this restriction.

Example 2.2 Consider the following XML document, which is valid with respect to the DTD given in Example 2.1:

```
<bookstore>
  <book>
    <author>
      <first-initial>J</first-initial>
      <middle-initial>M</middle-initial>
      <last-name>Coetzee</last-name>
    </author>
    <date><year>1990</year></date>
    <isbn>0-436-20012-0</isbn>
  </book>
  <magazine>
    <title>The Economist</title>
    <date><day>24</day><month>June</month><year>2000</year></date>
  </magazine>
  <book>
    <author>
      <first-name>Nadine</first-name>
      <last-name>Gordimer</last-name>
    </author>
```

```
    procedure ENCODE(i):
    begin
1.      if i labelled with n then
            if i has child node j labelled with an operator then enqueue(j);
                /* else node is empty or contains only text */
2.      if i labelled with ',' then
            begin
                let j and k be the first and second child nodes, respectively, of node i;
                ENCODE(j); ENCODE(k)
            end
3.      if i labelled with '|_L' or '|_R' then
            begin
                if i labelled with '|_L' then output 0 else output 1;
                if i has child node j then ENCODE(j)
            end
4.      if i labelled with '?' then
            if i has child node j then
                begin
                    output 1;
                    ENCODE(j)
                end
            else output 0
5.      if i labelled with '*' or '+' then
            begin
                let i have m child nodes, j_1, ..., j_m;
                output m;
                for k = 1 to m do ENCODE(j_k)
            end
6.      return
    end
```

Figure 4: Procedure used in encoding algorithm.

```
    <title>Something Out There</title>
    <date><year>1984</year></date>
    <isbn>0-670-65660-7</isbn>
  </book>
</bookstore>
```

Algorithm ENCODE-STRUCTURE starts by calling the ENCODE procedure with the node labelled `bookstore`. So ENCODE executes line 1, and since this node has a child labelled with operator +, the child node is added to the queue. The procedure returns and is then called with the node labelled +. This causes line 5 to be executed and, since there are 3 child nodes, the number 3 is output followed by ENCODE being called for each of the child nodes, which are labelled with $|_L$, $|_R$ and $|_L$, respectively.

Processing the first of these decision nodes results in 0 being output from line 3, after which ENCODE is called with the child node labelled `book`. Line 1 causes this node to be added to the queue. Then 1 is output (line 3) and the node labelled `magazine` added to the queue (line 1), followed by 0 being output (line 3) and the second node labelled `book` being added to the queue. At this point the children of `bookstore` in STRUCT(d) have been encoded and they are on the queue ready for their own structures to be encoded.

The above procedure continues until all the nodes in PARSE(d, D) have been processed. The final encoding is as follows (where the boxed items are integers representing numbers of occurrences, represented in decimal):

				children of:
3	0	1	0	`bookstore` (3 children: `book`, `magazine`, `book`)
1	0			1st `book` (1 `author`, no `title`)
0	0			`magazine` (no `volume`, no `issue`)
1	1			2nd `book` (1 `author`, `title`)
1	1	1		`author` of 1st `book` (more than `last-name`: `first-initial`, `middle-initial`)
0				`date` of 1st `book` (`year` only)
1	1			`date` of `magazine` (`month` and `day`)
1	0	0		`author` of 2nd `book` (more than `last-name`: `first-name`, no `middle-initial`)
0				`date` of 2nd `book` (`year` only)

Note that the length of an encoding for a particular element name can vary: the length of the encoding for the first and third `date`s above is 1, while that for the second `date` is 2. Assuming that each integer is encoded using a fixed length of 2 bits the total encoding length of the structure of d is 23 bits.

We observe that the parenthesization of regular expressions (either explicitly or implicitly) in content models can affect the length of the encoding used. For example, let r_1 be `a|b|c|d` and r_2 be `(a|b)|(c|d)`. Clearly $r_1 \equiv r_2$, but the maximum encoding length for r_1 is 3 (an occurrence of `d` is encoded as 111), while that for r_2 is 2. DTD designers can use this fact along with knowledge about the expected occurrences of elements to write their DTDs in such a way as to minimise the expected encoding length. For example, r_1 above gives the minimum expected encoding length of 1.75, if the probabilities of occurrence for `a`, `b`, `c` and `d` are, respectively, 0.5, 0.25, 0.125 and 0.125.

algorithm DECODE-STRUCTURE:
Input: ENCODING(d, D), denoted e for short below, and DTD D
Output: STRUCT(d)
Method:

```
begin
    /* Assume we have a single queue available,
    with operations enqueue(node) and dequeue() */
    let n be the first element name defined in D (the assumed document type);
    create node i with element name n;
    enqueue(i);
    while queue not empty do
        begin
            i = dequeue();
            let r be the content model in D of the element name of i;
            e = DECODE(e, r, i)
        end
end
```

Figure 5: Algorithm for decoding an encoded XML tree.

We now describe how to decode ENCODING(d, D), that is, how to recover STRUCT(d). The decoding algorithm, called DECODE-STRUCTURE, is shown in Figure 5. The algorithm makes use of a procedure called DECODE which is shown in Figure 6. The procedure outputs STRUCT(d) while working through the encoding, the remainder of which it returns after each call.

Note that because DECODE is guided by the structure of the regular expression for the content model of the element being processed, it knows what structure to expect at each stage, either an integer or a single bit. Furthermore, if DECODE is passed an encoding for more than the elements comprising the children of an element, the extra unused encoding is returned. Content models which can be empty do not cause problems because zero occurrences of a content model defined using `*` and instances of `EMPTY` as an operand of `|` (or used implicitly in `?`) are encoded explicitly.

Example 2.3 Assume that the DTD of Example 2.1 and the encoding produced in Example 2.2 are given as input to algorithm DECODE-STRUCTURE. The algorithm starts by creating a node labelled `bookstore` and calls DECODE with the full encoding, the regular expression `(book|magazine)+` and this node. Line 5 of DECODE is executed, n is found to be 3 and DECODE is called with the rest of the encoding after 3, the regular expression `(book|magazine), ((book|magazine), (book|magazine))` and the same node labelled `bookstore`. This results in line 3 being executed, with u being `(book|magazine)`.

DECODE is then called with u, which causes line 4 to be executed. Since the next bit in the encoding is 0, DECODE is called with u being `book`. Line 2 then creates a new node labelled `book` and adds it to the queue. The remaining encoding (with the 3 and the 0 removed) is returned to line 3, where it is used as the first argument of a call of DECODE with regular expression `(book|magazine), (book|magazine)`. The decoding of this consumes the

```
    procedure DECODE(e, r, i):
    begin
1.      if r = ε then return e
2.      if r = n then
            begin
                create node j with element name n as next child of node i;
                enqueue(j);
                return e
            end
3.      if r = (u, v) then return DECODE(DECODE(e, u, i), v, i)
4.      if r = (u|v) then
            if e = 0 e' then return DECODE(e', u, i)
            else /* e = 1 e' */ return DECODE(e', v, i)
5.      if r = (u)* or r = (u)+ then
            begin
                let e = n e';
                if n = 0 then return e'
                else return DECODE(e', (u_1, (··· (u_{n-1}, u_n))), i),
                    where u_i = u, 1 ≤ i ≤ n
            end
6.      if r = (u)? then return DECODE(e, (ε, u), i)
    end
```

Figure 6: Procedure used in decoding algorithm.

next 1 and 0 of the encoding, while adding nodes labelled `magazine` and `book` to the queue.

Control now returns to the loop in DECODE-STRUCTURE, the children of `bookstore` having been decoded and added to the queue. The decoding continues in this way until STRUCT(d) has been reconstructed. The data can then be added to STRUCT(d), in a straightforward way, to obtain d.

3 Analysis of the Compression Algorithm

The simplicity of the analysis we present for our algorithm hinges on the fact that the length of the encoding of the document's structure is $O(n)$ bits, where n is the number of nodes in the parse tree; see [KM90] for a survey on tree compression methods.

Given a parse tree $p = \text{PARSE}(d, D)$ of an XML document d with respect to a given DTD, D, let m be the number of repetition nodes in p, and q be the number of decision nodes in p. It is evident that the length of the encoding of a document's structure output by our algorithm, is given by

$$LEN(d, D) = mB + q, \tag{1}$$

where B is the number of bits needed to encode the maximum number of children of any repetition node in the parse tree. The encoding length, $LEN(d, D)$, can be viewed as the number of binary choices one needs to make to reach all the leaf nodes in the parse tree p. So, in Example 1.1 in order to reach a leaf we need to know the chapter number, the paragraph or figure number, and finally whether the leaf is a paragraph or a figure. As an example of a special case we have a *nested relational database* [LL99] (which subsumes the standard relational database model) where we only need mB bits to encode its structure, as in this case there are no decision nodes.

We note that although DTD rules are independent of each other, since DTDs induce a context-free grammar, there may be dependencies within rules that affect the expected value of $LEN(d, D)$. For example, consider the definition of date given by

```
date ((day?, month)?, year)
```

where a maximum of two decisions need to be made, i.e. whether the `date` has a `month` and then whether it also has a `day`. In this case, assuming choices in the parsing process are equally likely, the expected length of an encoding of `date` is 1.5, since we can encode the case when no `month` is specified by one bit and the case when a `month` is specified by two bits, the second bit indicating whether a `day` is present or not.

There are also situations when we can improve on the use of B bits to encode a repetition node. For example, if the number of children of most repetition nodes is bounded by B_1 and there are only few nodes having B_2 children, where B_2 is much larger than B_1, then a shorter code can be obtained by using a delimiter to signify the end of the code for the number of children, i.e. using a variable length code. In practice our technique of using B bits should work well, since the shorter codes, that only require $B_1 \ll B$ bits but use $B_2 = B$ bits instead, are likely to be compressed by a standard compressor at a later stage.

Now in what sense is the encoding length given by (1) optimal? For this we can derive a probability distribution over parse trees of documents given a DTD, assuming that their leaf

text nodes having been pruned. Thus for a decision node we assume that the probability of choosing a 0 or a 1 is 1/2 and for a repetition node we assume that the probability of having any number of repetitions is $1/2^B$. On the assumption that all choices are independent of each other, it follows that the probability of a document structure given a DTD is given by

$$P(d, D) = 2^{-(mB+q)}$$

and therefore

$$LEN(d, D) = -\log_2 P(d, D)$$

as required. (Obviously if the decisions are biased in some way we can improve on this bound.) Moreover, we can compute the entropy of a DTD, D, by

$$H(D) = \sum_{\text{PARSE}(d,D)} P(d, D)\ LEN(d, D),$$

with the sum being over all possible parse trees with respect to D that have had their text leaf nodes pruned. The entropy gives the expected encoding length of a document's structure given a DTD. We observe that, given a DTD, D, $H(D)$ may be computed directly from D by computing the probability of DTD rules one by one according to their regular structure, noting, as above, that the rules are independent of each other.

The above analysis assumes that the DTD is not recursive. It remains an open problem to extend the analysis to recursive DTDs.

4 Concluding Remarks

We have presented a novel algorithm to compress XML documents which are valid with respect to a given DTD. Our approach seems promising as we have shown that it minimises the length of encoding under the assumption of an independent distribution of choices. Our approach differs from previous approaches as it utilises the DTD to encode the structure of XML documents which is separated from its data. We are looking at ways for further improving the encoding length of XML documents by dropping the assumption that the DTD is fixed, and MDL, in particular, may provide such a framework.

In order to test the utility of our approach in practice, we plan to carry out experiments to compare it to existing systems for XML compression.

References

[BCW90] T. Bell, , J.G. Cleary, and I.H. Witten. *Text Compression*. Prentice Hall, Englewood Cliffs, NJ, 1990.

[BLHL01] T. Berners-Lee, J. Hendler, and O. Lassila. The semantic Web. *Scientific American*, 284:35–43, May 2001.

[Che01] J. Cheney. Compressing XML with multiplexed hierarchical models. In *Proceedings of IEEE Data Compression Conference*, pages 163–172, Snowbird, Utah, 2001.

[GP01] Charles F. Goldfarb and Paul Prescod. *The XML Handbook*. Prentice-Hall, third edition, 2001.

[HY01] M. H. Hansen and Bin Yu. Model selection and the principle of minimum description length. *American Statistical Association Journal*, 96:746–774, 2001.

[KM90] J. Katajainen and E. Mäkinen. Tree compression and optimization with applications. *International Journal of Foundations of Computer Science*, 1:425–447, 1990.

[LL99] M. Levene and G. Loizou. *A Guided Tour of Relational Databases and Beyond*. Springer-Verlag, London, 1999.

[LS00] H. Liefke and D. Suciu. XMILL: An efficient compressor for XML data. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*, pages 153–164, Dallas, Tx., 2000.

[Rez94] F.M. Reza. *An Introduction to Information Theory*. Dover, New York, NY, 1994.

[SM01] N. Sundaresan and R. Moussa. Algorithms and programming methods for efficient representation of XML for internet applications. In *Proceedings of International World Wide Web Conference*, pages 366–375, Hong Kong, 2001.

[Woo00] Peter T. Wood. Rewriting XQL queries on XML repositories. In *Proceedings 17th British National Conference on Databases (University of Exeter, UK, July 3–5)*, number 1832, pages 209–226, Berlin, 2000. Springer-Verlag.

[Woo01] Peter T. Wood. Minimising simple XPath expressions. In *Proceedings WebDB 2001: Fourth International Workshop on the Web and Databases (Santa Barbara, Ca., May 24–25)*, pages 13–18, 2001.