

# Compressing XML Documents Using Recursive Finite State Automata

Hariharan Subramanian and Priti Shankar\*

Department of Computer Science and Automation,  
Indian Institute of Science,  
Bangalore 560012, India  
{hariharan, priti}@csa.iisc.ernet.in

**Abstract.** We propose a scheme for automatically generating compressors for XML documents from Document Type Definition (DTD) specifications. Our algorithm is a lossless adaptive algorithm where the model used for compression and decompression is generated automatically from the DTD, and is used in conjunction with an arithmetic compressor to produce a compressed version of the document. The structure of the model mirrors the syntactic specification of the document. Our compression scheme is on-line, that is, it can compress the document as it is being read. We have implemented the compressor generator, and provide the results of experiments on some large XML databases whose DTD's are specified. We note that the average compression is better than that of XMLPPM, the only other on-line tool we are aware of. The tool is able to compress massive documents where XMLPPM failed to work as it ran out of memory. We believe the main appeal of this technique is the fact that the underlying model is so simple and yet so effective.

## 1 Introduction

Extensible Markup Language (XML) [1] is a standard meta language used to describe a class of data objects, called XML documents and to specify how they are to be processed by computer programs. XML is rapidly becoming a standard for the creation and parsing of documents. However, a significant disadvantage is document size, which is a consequence of verbosity arising from markup information. It is commonly observed that non-standardized text formats for describing equivalent data are significantly shorter. Theoretically, therefore, one should be able to compress XML documents down to the same size as the compressed versions of their non-standard counterparts. XML documents have their structure specified by DTD which specify the syntax of the documents. From an information-theoretic standpoint, portions of the document that have to do with its layout should not add to its entropy. It is therefore natural to investigate the use of syntactic models for the compression of such data. Present day XML databases are massive and the need for compression is pressing.

---

\* Contact Author.

A desirable feature of a compression scheme is the ability to be able to query the compressed document without decompressing the whole document. Whereas we do not address this problem in this paper, the fact that syntax plays a crucial role in the compression model is an indication that the scheme might be amenable to extensions that can achieve this. At present the scheme *is totally automatic* where the user specifies just the DTD and the compressor and decompressor are generated. DTD syntax, is very similar to that of Extended Context Free Grammars[2]. The left hand side of each rule is an element and the right hand side is a regular expression over elements. One could therefore construct a model that mirrors the DTD in the same way that recursive descent parsers mirror the underlying LL(1) [2] grammar. What is of importance here is that the model tracks the structure of the document, and is able to make accurate predictions of the expected symbols. More importantly, whenever the predicted symbol is unique, there is no need to encode it at all as the decoder generates the same model from the DTD and is thus able to generate the unique expected symbol. Most markup symbols fall into this category of symbols. Character data associated with a single element is automatically directed to the same model for arithmetic compression irrespective of the *instance* of the element in the DTD.

The syntax directed translation scheme converts the DTD into a set of Deterministic Finite Automata (DFA) one for each element in the DTD. Each transition is labeled by an element, and the action associated with a transition is a call to a simulator for the DFA for the element labeling that transition. Every element that has some attributes or character data has an associated container. The scheme we describe automatically groups all data for the same element into a single container which is compressed incrementally using a single model for an arithmetic order-4 compressor[3, 4]. We have run experiments on five large databases and compared the performance of our tool with that of two well known XML-aware compression schemes, XMill[5] and XMLPPM[6]. Two of these databases, DBLP[7] and UniProt[8] are well known. The other three (XMark[9], Michigan[10] and XOO7[11]) are XML benchmark projects. The tool XMLPPM could not compress two of the databases as it ran out of memory. The average compression ratio of our scheme is better than that of XMLPPM and significantly better than that of XMill. The time and space overheads are somewhat larger than those of XMill. This is an inherent drawback of a scheme based on arithmetic coding, which has to perform costly table updating operations after seeing every symbol. However XMill cannot perform on-line compression as can XMLPPM and our tool XAUST(XML compression with *A*utomata and a *S*Tack). Section 2 describes related work. Section 3 is a short background on arithmetic coding. Section 4 is a summary of the structure of XML documents and DTD, and this is followed by a description of our scheme. Section 5 compares our compression results with those of XMill and XMLPPM and that of a general purpose compressor `bzip2`[12]. Section 6 concludes the paper.

## 2 Related Work

The use of syntax in the compression of program files is not new. Cameron[13] has used Context Free Grammars (CFG) for compressing programs. Given estimates for derivation step probabilities, he has shown how to construct practical encoding systems for compression of programs whose syntax is defined by a CFG. The models are, however, fairly complex in their operation. For the scheme to be effective, these probabilities have to be learned on sample text. Syntax based schemes have also been used for machine code compression [14, 15, 16, 17]. The XML-specific compression schemes that we are aware of are XMLZIP[18], XMill and XMLPPM. The last two have tried to take advantage of the structure in XML data by either transforming the file after parsing, breaking up the tree into components[5] or injecting hierarchical element structure symbols into a model that multiplexes several models based on the syntactic structure of XML [6]. They do not require the DTD to compress the document, and even if it is available it is not used (XMill can use it but only interactively).

XMLZIP parses XML data and creates the underlying tree. It then breaks up the tree into many components, the *root* component at depth  $d$  and a component for each of the subtrees at depth  $d$ . Each of the subtrees is compressed using Java's ZIP-DEFLATE archive library. The advantage of such a scheme is that it allows limited random access to parts of the document without the need to have the whole tree in main memory.

XMill separates the structure from the content and compresses them separately. Data items are grouped into containers and each container is compressed separately. Different compressors are applied to compress different containers depending on the content. The criterion for grouping data into a container is not just the tagname but also the path from the root to the tagname. XMill does not compress the document on-line.

XMLPPM uses a modeling technique called Multiplexed Hierarchical Modeling (MHM), based on the SAX[19] encoding and on PPM[20] modeling. The technique employs two basic ideas: multiplexing several text compression models based on the syntactic structure of XML (one model for element structure, one for attributes, and so on), and injecting hierarchical element structure symbols into the multiplexed models (these are essentially root to leaf paths to the element). Multiplexing enables more effective hierarchical structure modeling. A common case for these dependencies is for the enclosing element tag to be strongly correlated with enclosed data. MHM exploits this by injecting the enclosing tag symbol into the element, attribute or string model immediately before an element, attribute or string is encoded. Injecting a symbol means telling the model that it has been seen but not explicitly encoding or decoding it.

At the cost of a degraded compression quality tools have been designed that allow certain kinds of queries on the compressed versions. Examples of such schemes are [21, 22, 23].

We first describe the well known arithmetic encoding technique that is embedded into our scheme and is an essential component of it.

### 3 Arithmetic Coding and Finite Context Modeling

#### 3.1 Arithmetic Coding

Arithmetic coding does not replace every input symbol with a specific code. Instead it processes a stream of input symbols and replaces it with a single floating point output number. The longer (and more complex) the message, the more bits are needed in the output number.

The output from an arithmetic coding process is a single number less than 1 and greater than or equal to 0. This single number can be uniquely decoded to create the exact stream of symbols that went into its construction. In order to construct the output number, the symbols being encoded need to have a set of probabilities assigned to them. Initially the range of the message is the interval  $[0, 1)$ . As each symbol is processed, the range is narrowed to that portion of it allocated to the symbol.

#### 3.2 Finite Context Modeling

In a finite context scheme, the probabilities of each symbol are calculated based on the *context* the symbol appears in. In its traditional setting, the context is just the symbols that have been previously encountered. The *order* of the model refers to the number of previous symbols that make up the context. In an adaptive order  $k$  model, both the compressor and the decompressor start with the same model. The compressor encodes a symbol using the existing model and then updates the model to account for the new symbol. Typically a model is a set of frequency tables one for each context. After seeing a symbol the frequency counts in the tables are updated. The frequency counts are used to approximate the probabilities and the scheme is adaptive because this is being done as the symbols are being scanned. The decompressor similarly decodes a symbol using the existing model and then updates the model. Since there are potentially  $q^k$  possibilities for level  $k$  contexts where  $q$  is the size of the symbol space, update can be a costly process, and the tables consume a large amount of space. This causes arithmetic coding to be somewhat slower than dictionary based schemes like the Ziv-Lempel[24] scheme.

## 4 Automata Representing XML Documents

XML documents contain *element tags* which include start tags like `<name>` and end tags like `</name>`. Elements can nest other elements and therefore a tree structure can be associated with an XML document. Elements can also contain plain text, comments and special processing instructions for XML processors. In addition, opening element tags can have attributes with values such as `gender` in `<person gender='female'>`. Detailed specifications are given in [1].

XML documents have to conform to a specified syntax usually in the form of a DTD. Usually XML documents are parsed to ensure that only valid data reaches an application. Most XML parsing libraries use either the SAX interface

or the DOM(Document Object Model) interface. SAX is an event based interface suitable for search tools and algorithms that need one pass. The DOM model on the other hand is suitable for algorithms that have to make multiple passes.

Since XML documents are stored as plain text files one possibility is to use standard compression tools like `bzip2` or `ppm*`. Cheney[6] has performed a study of the compression using such general purpose tools and observed that each general purpose compressor performs poorly on at least one document. Since XML documents are governed by a rather restrictive set of rules the obvious way to go, is to try to use the rules to predict what symbols to expect. Further if the rules are already known a-priori then the compressor which is tuned to take advantage of the rules can be generated directly from the rules themselves. This is what we achieve in our scheme XAUST.

The scheme proposed in this paper assumes that the DTD describing the data is known to both the sender and the receiver. Typically, an element of a DTD consists of distinct beginning and ending tags enclosing regular expressions over other elements. Elements can also contain plain text, comments and special instructions for XML processors (“processing instructions”). Opening element tags can have attributes with values.

*Example 1.* Consider a DTD defined as follows:

```
<!DOCTYPE addressBook[
<!ELEMENT addressBook (card*)>
<!ELEMENT card ((name | (givenName, familyName)), email, note?)>
<!ELEMENT name (#PCDATA)>
<!ELEMENT givenName (#PCDATA)>
<!ELEMENT familyName (#PCDATA)>
<!ELEMENT email (#PCDATA)>
<!ELEMENT note (#PCDATA)>
]>
```

Below is an instance of an XML document conforming to this DTD.

```
<addressBook>
<card>
<givenName>Hariharan</givenName>
<familyName>Iyer</familyName>
<email>hari@gmail.com</email>
</card>
<card>
<name>Priti Shankar</name>
<email>priti@gmail.com</email>
<note>Hariharan's advisor</note>
</card>
</addressBook>
```

The strings following each element declaration are just regular expressions over element names and therefore each of them can be associated with a DFA.

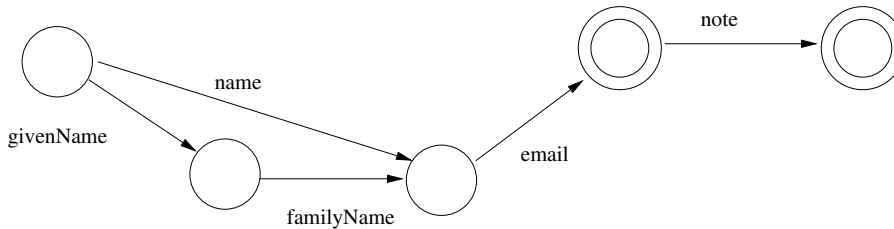


Fig. 1. DFA for the right hand side of the production for  $n_n$  in example 1

The DFA for the right hand side of the rule for element `card` is shown in Fig., 1. There are two kinds of states in this automaton, those having a single output transition and those with multiple output transitions. Symbols that label single output transitions need not be encoded as their probability is 1. Thus encoding of symbols by the arithmetic compressor needs to be performed only at states with more than one outgoing transition. An arithmetic encoding procedure is called at each such state for each element. As we observed in Section 3, the arithmetic encoder maintains tables of frequencies which it updates each time it encodes a symbol. Each element which has a `#PCDATA` attribute will result in a call to an arithmetic encoder which uses a common model for all instances of that element attribute and encodes them using the same set of frequency tables. A typical sequence of actions is then as follows: Enter the start state of a DFA representing the right side of a rule; if there is only one edge out of the state then do nothing; if that element has a `#PCDATA` attribute then encode the string of symbols using the frequency tables associated with that element; if there is more than one edge encode the element labeling the edge taken, using an arithmetic encoder for that state, and transit to the the start state of the DFA for that element; the decoder mimics the action of the encoder generating symbols that are certain and using the arithmetic decoder for symbols that are not.

XAUST uses a single container for the character data associated with each element though this has the capability to use the context (i.e. the path along which it reached this element). The reason is best illustrated by the example below:

*Example 2.* Consider the element below

```

<!ELEMENT Project (date, date, ...)>
<!ELEMENT Employee (date, ...)>
<!ELEMENT date (#PCDATA)>
    
```

The `date` in `Employee` is the joining date. The first and second `date` in `Project` are the starting and ending dates respectively of the project. XAUST uses a single model for `date` and the reason is clear. Experimentation indicates that having different models for `date` in this case is counter-productive as different models for essentially the same kind of data consume an inordinate amount of memory with little or no gain in compression ratio.

#### 4.1 Compression and Decompression Using XAUST

A state of the compressor is a pair (**element**, **state**) where **element** represents the current element whose DFA XAUST is traversing, and **state** the state of the DFA where it currently is. Assume that the current state of the Encoder is  $(i, j)$ . When an open tag is encountered for element  $k$  in the document, the current state pair of the encoder is stored on the calling stack and the DFA for the element  $k$  is entered. The current state of the encoder now becomes  $(k, 0)$ . When the end tag is encountered for element  $k$ , the stack is popped and the new state of the encoder becomes  $(i, j + 1)$ . As mentioned earlier, tags are not encoded if the number of output transitions is equal to 1. For example, for the case below we need not encode the tag D but we have to encode B and C.

```
<!ELEMENT A ((B | C), D)>
```

Every state has an arithmetic model which it uses to encode the next state. Note that this is different from the model used to encode character data, which is handled as described below.

Consider the element below.

```
<!ELEMENT A ((#PCDATA | B)*)>
```

There are two transitions from the start state of the DFA for element A. One of them invokes the arithmetic model for PCDATA which is common for all PCDATA associated with any instance of element A in the document. The other transition invokes the DFA for element B after pushing the current state in the stack.

The pseudo-code for Encoder (Compressor) and Decoder (Decompressor) is given below. For the sake of brevity only these two routines are shown. Encoding and decoding attributes are also not shown. We can see the similarity between Encoder and Decoder routines.

```
void Encoder()
{
    ExitLoop = true;
    //StateStruct is a pair of int(ElementIndex, StateIndex)
    //ElementIndex represents the automaton
    //StateIndex is the state in the above automaton
    StateStruct CurrState(0, 0);

    while(ExitLoop == false)
    {
        Type = GetNextType(FilePointer, ElementIndex);

        switch(Type)
        {
        case OPENTAG:
            //Encode ElementIndex in CurrState context
            EncodeOpenTag(CurrState, ElementIndex);
            Stack.push(CurrState);
            CurrState = StateStruct(ElementIndex, 0);
            break;
```

```

case CLOSETAG:
    //Encode CLOSETAG in CurrState context
    EncodeCloseTag(CurrState);
    if(Stack.empty() == true)
    {
        ExitLoop = true;
    }
    else
    {
        CurrState = Stack.pop();
        //Make state transition in CurrState.ElementIndex
        //automaton and get the next state
        CurrState.StateIndex = MakeStateTransition(CurrState,
                                                    ElementIndex);
    }
    break;

case PCDATA:
    //Encode PCDATA in Currstate context
    EncodePcdata(CurrState);
    CurrState.StateIndex = MakeStateTransition(CurrState, PCDATA);
    break;
}
}
}

void Decoder()
{
    ExitLoop = true;
    StateStruct CurrState(0, 0);

    while(ExitLoop == false)
    {
        //Decode the type in CurrState context
        Type = DecodeNextType(FilePointer, CurrState, ElementIndex);

        switch(Type)
        {
        case OPENTAG:
            //Write open tag of the Element of ElementIndex
            WriteOpenTag(ElementIndex);
            Stack.push(CurrState);
            CurrState = StateStruct(ElementIndex, 0);
            break;

        case CLOSETAG:
            //Write close tag of the Element of ElementIndex
            WriteCloseTag(ElementIndex);
            if(Stack.empty() == true)

```

```

    {
        ExitLoop = true;
    }
    else
    {
        CurrState = Stack.pop();
        CurrState.StateIndex = MakeStateTransition(CurrState,
                                                    ElementIndex);
    }
    break;

case PCDATA:
    DecodePcdata(CurrState);
    CurrState.StateIndex = MakeStateTransition(CurrState, PCDATA);
    break;
}
}
}
}

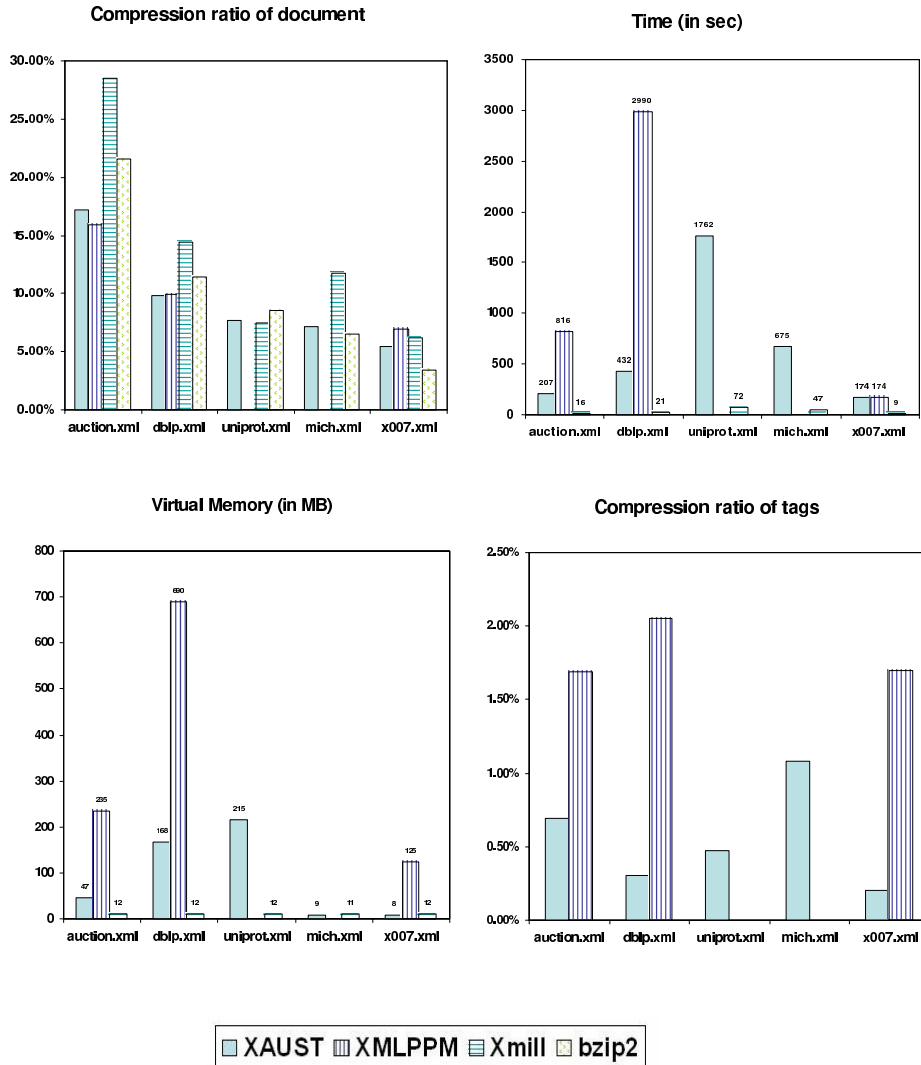
```

## 5 Experimental Results

We have examined the comparative performance of three tools XMill, XMLPPM and XAUST on five large XML documents. The sizes of these documents are displayed in Table 1. We define the *Compression Ratio* as the ratio of the size of the compressed document to the size of the original document expressed as a percentage. The compression ratios for all three schemes are shown in Fig. 2 along with that of a general purpose compressor `bzip2`. The compression ratios of XAUST and XMLPPM are considerably better than that of XMill for all but one of the documents. XMLPPM, however, ran out of memory for two documents. It also takes significantly longer than XAUST whereas XMill is very fast and economical in its use of space. The disadvantage of XMill is that it cannot perform on-line compression. We expect that our scheme will do well wherever the markup content is high as tags whose probability of occurrence is 1 are not included in the compressed stream. Figure 2 also shows the compression ratios for tags alone. XAUST compresses tags more efficiently than in other schemes.

**Table 1.** Sizes of XML documents that were compressed

Name	Size (in MB)
auction	113
dblp	253
uniprot	1070
michigan	495
x007	128



**Fig. 2.** Statistics for Compression Ratios, Running Times and Memory Usage for XMill, XAUST, XMLPPM and bzip2. XMLPPM ran out of memory for uniprot.xml and mich.xml. Running times are shown for only XML-aware schemes.

XAUST does not need a SAX parser as do XMill and XMLPPM as some form of parsing is already embedded in its action.

## 6 Conclusion and Future Work

We have presented a scheme for the compression of XML documents where the underlying arithmetic model for the compression of tags is a finite state automa-

ton generated directly from the DTD of the document. The model is automatically switched on transiting from one automaton to another storing enough information on the stack so that return to the right state is possible; this ensures that the correct model is always used for compression. (In fact it precisely achieves the *multiplexing of models* mentioned in XMLPPM in a completely natural manner). On return, the stack is used to recover the state from which a transition was made. The scheme is reminiscent of a recursive descent parser except that it is not subject to the LL(1) restrictions. Our technique directly generates the compressor from the DTD in the appropriate format with no user interaction except the input of the DTD. Our experiments on five large databases indicate that the scheme is better on the average than XMLPPM in terms of compression ratio, much faster in terms of running time and more economical in terms of memory usage. In fact XMLPPM ran out of memory for UniProt and Michigan documents. The tool XMill runs much faster and in less space, but its average performance is considerably inferior to that of XAUST as can be observed from Fig. 2.

The dynamic space requirements for the compressor are dominated by the size of the tables for the arithmetic compressor which grow exponentially with the size of the context (order 4 is used here). Also updating these tables after each symbol is processed makes the compression rather slow in comparison with dictionary based schemes.

Future work will concentrate on modifying this scheme to facilitate simple tree queries on the XML text.

## References

1. XML: W3C recommendation. <http://www.w3.org/TR/REC-xml> (2004)
2. Backhouse, R.C.: Syntax of Programming Languages - Theory and Practice. Prentice Hall International, London (1979)
3. Witten, I.H., Neal, R.M., Cleary, J.G.: Arithmetic coding for data compression. *Commun. ACM* **30** (1987) 520–540
4. Nelson, M.: Arithmetic coding and statistical modeling. <http://dogma.net/markn/articles/arith/part1.htm>. *Dr. Dobbs Journal* (1991)
5. Liefke, H., Suciu, D.: XMILL: An efficient compressor for XML data. In: SIGMOD Conference. (2000) 153–164
6. Cheney, J.: Compressing XML with Multiplexed Hierarchical PPM Models. In: Proceedings of the Data Compression Conference, IEEE Computer Society (2001) 163–172
7. DBLP: (<http://www.informatik.uni-trier.de/~ley/db>)
8. UniProt: (<http://www.ebi.uniprot.org>)
9. XMark: (<http://monetdb.cwi.nl/xml/generator.html>)
10. Michigan: (<http://www.eecs.umich.edu/db/mbench>)
11. XOO7: (<http://www.comp.nus.edu.sg/~ebh/xoo7.html>)
12. Bzip2: (<http://www.bzip.org>)
13. Cameron, R.D.: Source encoding using syntactic information source models. *IEEE Transactions on Information Theory* **34** (1988) 843–850
14. Ernst, J., Evans, W.S., Fraser, C.W., Lucco, S., Proebsting, T.A.: Code compression. In: PLDI. (1997) 358–365

15. Franz, M.: Adaptive compression of syntax trees and iterative dynamic code optimization: Two basic technologies for mobile object systems. In: *Mobile Object Systems: Towards the Programmable Internet*. Springer-Verlag: Heidelberg, Germany (1997) 263–276
16. Franz, M., Kistler, T.: Slim binaries. *Commun. ACM* **40** (1997) 87–94
17. Fraser, C.W.: Automatic inference of models for statistical code compression. In: *PLDI*. (1999) 242–246
18. XMLZIP: (<http://www.xmls.com>)
19. SAX: (<http://www.megginson.com/sax>)
20. Cleary, J.G., Teahan, W.J.: Unbounded length contexts for PPM. *The Computer Journal* **40** (1997) 67–75
21. Tolani, P.M., Haritsa, J.R.: XGRIND: A query-friendly XML compressor. In: *ICDE*. (2002) 225–234
22. Min, J.K., Park, M.J., Chung, C.W.: XPRESS: A queryable compression for XML data. In: *SIGMOD Conference*. (2003) 122–133
23. Arion, A., Bonifati, A., Costa, G., D’Aguanno, S., Manolescu, I., Pugliese, A.: Efficient query evaluation over compressed XML data. In: *EDBT*. (2004) 200–218
24. Ziv, J., Lempel, A.: A universal algorithm for sequential data compression. *IEEE Transactions on Information Theory* **23** (1977) 337–343