Departamento de
Informática
Universidad de Valladolid

Computer Science Department
University of Valladolid
Valladolid - Spain

Technical Report DI-2003-5

Lempel-Ziv Compression of Structured Text*

Joaquín Adiego[1], Gonzalo Navarro[2], and Pablo de la Fuente[1]

[1]GRINBD, Dpto. de Informática, Universidad de Valladolid, Valladolid, España.
`{jadiego, pfuente}@infor.uva.es`
[2]CIW, Dpto. de Ciencias de la Computación, Universidad de Chile, Santiago, Chile.
`gnavarro@dcc.uchile.cl`

Abstract We describe a novel Lempel-Ziv approach suitable for compressing structured documents, called *LZCS*, which takes advantage of redundant information that can appear in the structure. The main idea is that frequently repeated subtrees may exist and these can be replaced by a backward reference to their first ocurrence. The main advantage is that compressed documents generated by LZCS are easy to display, access at random, and navigate. In a second stage, processed documents can be further compressed using some semiadaptive technique, so that random access and navigability remain possible. LZCS is especially efficient to compress collections of highly structured data, such as XML forms,

* This work was partially supported by CYTED VII.19 RIBIDI project (all authors) and Millennium Nucleus Center for Web Research, Grant P01-029-F, Mideplan, Chile (second author).

invoices, e-commerce and web-service exchange documents. The comparison against structure-based and standard compressors shows that LZCS is a competitive choice for this type of documents, while the others are not well-suited to support navigation or random access.

Keywords: Ziv-Lempel, XML Data, Text Compression.

1 Introduction

The storage, exchange, and manipulation of structured text as a device to represent semistructured data is spreading across all kinds of applications, ranging from text databases and digital libraries to web-services and electronic commerce. Structured text, and in particular the XML format, is becoming a standard to encode data with simple or complex, fixed or varying structure. Although XML has been envisioned as a mechanism to describe structured data from some time ago, it has been the recent explosion of "electronic business" that has shown its potential to describe all sorts of documents exchanged between organizations and stored inside an organization. Examples are invoices, receipts, orders, payments, accounting, and other forms.

Although the information stored by an organization is usually kept in relational databases and/or data warehouses, it is important to store digital copies, in XML format, of all the documents that have been exchanged and/or produced along time. A structured text retrieval engine should provide random access to those structured documents, so that they should be easily searched, visualized, and navigated. On the other hand, as usual, we would like this repository to take as little space as possible.

In this paper we focus on the compression of structured text. We aim specifically at compression of highly structured data, such as forms where there is little text in each field. Collections formed by those types of forms contain a lot of redundancy that is not captured well enough by classical compression methods. At the same time, we want the compressed collection to be easily accessed, visualized and navigated. Existing structure-aware compression methods do not account for these capabilities: texts have to be uncompressed first before they can be accessed.

We develop a compression method, LZCS, inspired in Lempel-Ziv compression, where repeated substructres are factored out. We obtain very good compression ratios, much better than those of classical methods, and competitive against other structure-aware methods. Only *XMLPPM* compresses better than our LZCS. However, text collections compressed with LZCS are easily accessed at random, visualized and navigated, which is not possible with *XMLPPM*, which is adaptive and hence needs to uncompress the whole collection before extracting a single document.

Moreover, LZCS algorithm is one-pass, which means that it can output the compressed text almost immediately after seeing the source text. This makes it suitable for use over a communication network without introducing any delay in the transmission. The output of LZCS is still plain text, which easies transmission over plain ASCII channels. In a second pass, the output of LZCS can be further compressed using a coding method that retains navigability and random access.

2 Text compression

2.1 Compressing plain text

In general, classic methods of text compression do not take into account the structure of the documents they compress. At the end of the seventies, Lempel and Ziv designed new technologies of data compression based on replacing text substrings by previous repeated ocurrences. Their two most famous algorithms are called LZ77 [15] and LZ78 [16], as well as the later variant LZW [13]. Depending on the variants, different previous strings can be referenced, while others cannot. These techniques do not consider the semantic meaning of sequences replaced. The Lempel-Ziv family is the most popular to compress text because it combines good compression ratios with fast compression and decompression.

With regard to compressing natural language texts in order to permit efficient retrieval from the collection, the most successful techniques are based on models where the text words are taken as the source symbols [8], as opposed to the traditional models where the characters are the source symbols.

Words reflect much better than characters the true entropy of the text [3]. For example, a semiadaptive Huffman coder over the model that considers characters as symbols typically obtains a compressed file whose size is around 60% of the original size, on natural language. A Huffman coder when words are the symbols obtains 25% [17]. Another example is the WLZW algorithm, which uses Ziv-Lempel on words [4,6].

On the other hand, most information retrieval systems use words as their main information atoms, so a word-based compression easies the integration with an information retrieval system. Some examples of successful integration are [14,10,9].

2.2 Compressing Structured Text

SCM [1,2] is a generic model used to compress semistructured documents, which takes advantage of the context information usually implicit in the structure of the text. The idea is to use a separate model to compress the text that lies inside each different structure type (e.g., each different XML tag). The idea is that the distribution of all the texts that belong to a given structure type should be similar, and different from that of other structure types.

Another compression method that considers the document structure is *XMill* [7], developed in AT&T Labs. XMill is an XML-specific compressor designed to exchange and store XML documents, and its compression approach is not intended for directly supporting querying or updating of the compressed document. XMill is based on the *zlib* library, which combines Ziv-Lempel compression with a variant of Huffman.

Yet another XML compressor is *XGrind* [12], which directly supports queries over the compressed files. An XML document compressed with XGrind retains the structure of the original document, permitting reuse of the standard XML techniques for processing the compressed document. It does not, however, take full advantage of the structure.

Other approaches to compress XML data exist, based on the use of a PPM-like coder, where the context is given by the path from the root to the tree node that contains the current text. One example is *XMLPPM* [5], which is an adaptive compressor based on PPM, where the context is given by the structure.

3 LZCS description

LZCS is a new technique to compress structured text (such as XML and HTML) that allows one to easily navigate the compressed structure. Thus, LZCS can be integrated into a structured text retrieval system without loss of efficiency in the search or visualization of results. The main idea is based on the Ziv-Lempel concept, so that repeating substructures and text blocks are replaced by a backward reference to their first ocurrence in the processed document. The result is a valid structured text with additional special tags (backward reference tags), which can be transmitted, handled or visualized in a conventional way, or further compressed using some existing compressor.

These documents are visualized in the usual way up to meeting a backward reference. When a backward reference appears, we push current text position in a stack and move to the indicated text position. If the referenced text begins with a start-tag, then the backward reference will finalize when the corresponding end-tag appears. Otherwise, it will finalize when a start-tag appears. When the referenced text finishes we pop previous text position from the stack and continue. Further backward references can appear in referenced text, in which case we repeat the same process. A similar procedure can be used to traverse or navigate the structure in tree form.

Since the documents generated by LZCS are navigable, a good idea is to further compress them using a semiadaptive compression method, like word-based Huffman. After this process, the documents cannot anymore be visualized as plain text (a word-wise decompression is needed), but they are still navigable and accessible at random positions.

In the following we formally define the LZCS transformation.

3.1 Formal definition

Definition 1 (Text Block) *A text block will be any maximal consecutive alphanumeric character sequence not containing structure or backward reference tags.*

Definition 2 (Structural Element) *A structural element will be any consecutive character sequence that begins with a start-tag and finalizes with its corresponding end-tag.*

Bearing in mind last definition, a structural element can contain one or more text blocks, one or more structural elements and/or one or more backward reference tags. For simplicity, other types of valid tags (e.g. comment tags, autocontained tags and so on) will be treated as conventional text, and only start-tags and end-tags will be used to identify structural elements.

The structure induces a hierarchy that can be represented as a tree. Let us regard documents in tree form. Text blocks will be represented by leaves, and structural elements by subtrees.

Definition 3 (Node) *A* node *will be either a text block or a structure element.*

The main point of LZCS is to replace some subtrees by references to equivalent subtrees seen before.

Definition 4 (Equivalent Nodes) *Let $\mathcal{N}_1$ and $\mathcal{N}_2$ be two nodes that appear in a collection. We will say that node $\mathcal{N}_1$ is equivalent to node $\mathcal{N}_2$ iff $\mathcal{N}_1$ is textually equal to $\mathcal{N}_2$.*

Definition 5 (LZCS Transformation) *LZCS replaces each maximal node that is equivalent to a previous node by a backward reference to its first occurrence in the text. Other elements are left unchanged. "Maximal" means that the node replaced does not descend from another that can be replaced.*

A *backward reference* is represented by a special tag in the output. The special tag is constructed by means of the symbols `<@` and `>` that mark the beginning and end of the backward reference tag. The content of this tag will be formed by digits that express an unsigned integer indicating the absolute position where the referenced element begins. For space optimization, this number will be expressed in base 62, using `0..9`, `A..Z` and `a..z` as digits.

It may happen that a referenced text block is smaller than the reference itself (for example, when the text block is formed only by character `'\n'`). In these circumstances, replacing it by a reference is not a good choice. Hence we do not replace text blocks that are shorter than a user-specified parameter l. The choice of l influences compression ratio, but not correctness.

3.2 Example

Assume that we are going to compress a collection of three documents using LZCS. The documents are represented in Figure 1. In the figure, there exist three different structural elements represented by circles. The structural element of type 1 has the circle drawn with a continuous line, that of type 2 with a dashed line, and that of type 3 with a dotted line. Text blocks are represented by squares. Letters and numbers in the figure represent node identifiers.

To cover all the possibilities, suppose that text blocks numbered 1, 4, 7 and 9 in the figure are equivalent. Also text blocks numbered 3 and 10 are equivalent, as well as those numbered 6 and 8. With this, the documents share repeating parts (that is, equal subtrees). Figure 2 shows graphically these correspondences and Figure 3 shows the collection transformed with LZCS.

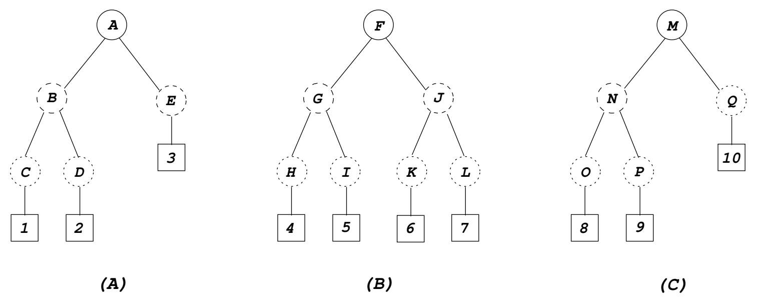


Figure1. Three example documents.

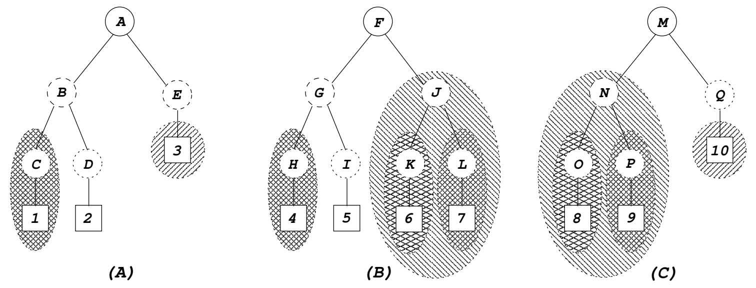


Figure2. Equivalent subtrees of the documents.

4 An efficient implementation

The simplest way to implement LZCS transformation is by searching all previously processed text for each new structural element. This way, we have a complexity $O(n^2)$, which is unacceptable. We show now how to obtain $O(n)$ average time by hashing.

When a text block is processed, we first obtain its digital signature (for example, using MD5 algorithm [11]). If the text block is not equivalent to any previous text block (its signature does not coincide with previous ones), then the text block is copied verbatim in the output and its signature is added to the (hashed) set of signatures of original text blocks, together with the text position of the block (which is the first occurrence of this block). Otherwise, if an equivalent text block appears (their digital signatures coincide) a backward reference to the first occurrence of the text block is written in the output. (Since digital signature algorithms do not ensure that signatures are unique, texts are also directly compared when a coincidence arises.)

In order to apply hashing to structure elements too, a node signature is generated and stored, along with its start position, for nodes that have not appeared before. Node signatures of parent nodes are produced after those of children nodes.

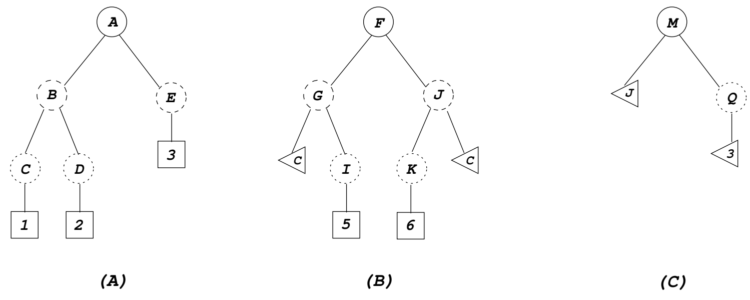


Figure3. Example documents after applying the LZCS transformation. Backward references are represented by triangles.

Definition 6 (Node Signature) *A node signature is formed by concatenating its start-tag identifier and children identifiers. These are either their start text positions if they are not references, or their referenced positions otherwise.*

As we show in Lemma 2, a node signature is unique for a collection. For each new structure element, its node signature is generated and searched for among the existing ones. If a coincidence is found then the current structure element is equivalent to a previous one, and it can be replaced.

Next lemma is useful to prove the correctness of this hashing scheme.

Lemma 1. *Let $\mathcal{N}_1$ and $\mathcal{N}_2$ be two nodes that appear in a collection transformed with LZCS up to node $\mathcal{N}_2$, $\mathcal{N}_1$ preceding $\mathcal{N}_2$. Then, $\mathcal{N}_1$ is equivalent to $\mathcal{N}_2$ iff $\mathcal{N}_2$ is a backward reference to $\mathcal{N}_1$, or $\mathcal{N}_1$ and $\mathcal{N}_2$ are equal backward references.*

Proof: We prove the equivalence in both directions.

1. If $\mathcal{N}_1$ is equivalent to $\mathcal{N}_2$ then either $\mathcal{N}_2$ is a backward reference to $\mathcal{N}_1$ or $\mathcal{N}_1$ and $\mathcal{N}_2$ are equal backward references, because LZCS transformation replaces $\mathcal{N}_2$ by a backward reference to its first ocurrence:
   (a) If $\mathcal{N}_1$ is the first ocurrence then $\mathcal{N}_2$ is replaced by a backward reference to $\mathcal{N}_1$.
   (b) Otherwise, let $\mathcal{N}_0$ be the first ocurrence of $\mathcal{N}_2$, then $\mathcal{N}_2$ is replaced by a backward reference to $\mathcal{N}_0$, but also $\mathcal{N}_1$ was replaced by a backward reference to $\mathcal{N}_0$.
2. If $\mathcal{N}_2$ is a backward reference to $\mathcal{N}_1$, or $\mathcal{N}_2$ and $\mathcal{N}_1$ are equal backward references, then $\mathcal{N}_1$ is equivalent to $\mathcal{N}_2$, because in both cases it holds that either $\mathcal{N}_1$ and $\mathcal{N}_2$ contents are textually equal, or $\mathcal{N}_1$ contents are textually equal to $\mathcal{N}_0$ and $\mathcal{N}_2$ contents are textually equal to $\mathcal{N}_0$. □

Bearing in mind Lemma 1, we show next that the node signature is unique and works correctly.

Lemma 2. *Nodes $\mathcal{N}$ and $\mathcal{N}'$ are equivalent iff their node signature are equal.*

Proof: We observe that a node only can be previously repeated if all its children are also repeated. Therefore, a node $\mathcal{N}$, parent of $\mathcal{N}_1 \ldots \mathcal{N}_k$, is textually equal to a later node $\mathcal{N}'$, parent of $\mathcal{N}'_1 \ldots \mathcal{N}'_k$, iff tag identifiers of $\mathcal{N}$ and $\mathcal{N}'$ are equal and $\forall i \in 1..k, \mathcal{N}'_i$ is equivalent to $\mathcal{N}_i$. By Lemma 1, the latter means that either $\mathcal{N}'_i$ points to $\mathcal{N}_i$, or $\mathcal{N}'_i$ points to some $\mathcal{N}_0$ and $\mathcal{N}_i$ points to $\mathcal{N}_0$. Then, if we use Definition 6 to construct the node signatures of $\mathcal{N}$ and $\mathcal{N}'$, the node signatures will be equal if and only if the nodes are equivalent. □

We explain now LZCS transformation algorithm. When an end-tag appears its corresponding node signature is obtained and searched for in the (hashed) set of node signatures. If the current node signature is present in the set, then it can be replaced by a backward reference. However, at this point we are not sure that the current node is a maximal repeated subtree. Therefore the substitution is done only in memory, but nothing is yet written to the output. On the other hand, if the current node signature is not present in the set, then the current subtree is not equivalent to any previous one and, therefore, nonwritten children (in plain form or references) and current node (in plain form) must be written to the output. Also, the current node signature is added to the set of node signatures.

Figure 4 describes the basic LZCS transformation. Set *PreviousSubtree* contains the elements that have been converted to references but are not yet output because we do not know whether they are maximal. If we are currently processing some tree node, then *PreviousSubtree* may contain siblings to the left of the node and of ancestors of the node. By adding new nodes at the end of the set we know that, once we go back to the parent node, the latter elements of the set are all the children of that parent node. This permits implementing *PreviousSubtree.erase_ children* easily, just by knowing the arity of current node.

Decompression is very simple. It begins by writing the text to the output and, when it finds a backward reference tag, it writes the text that is at the referenced position. If that text begins with a start-tag, the "backward jump" will finish when the corresponding end-tag is written. If the text does not begin with a start-tag, the "backward jump" will finish when the first start-tag appears. Next, process is continued from the backward reference tag.

4.1 Example

Let us go back to the documents shown in the example of Section 3.2. The documents will be processed left to right, as they appear in Figure 1. In the first document no substitution is carried out, since there are no equivalent nodes in the document. At this moment, the output will contain an exact copy of the first document. Then the second document is processed. Since text block 4 is equivalent to 1, it is replaced by a backward reference, represented by triangles in Figure 5-A. As the structural elements that contain blocks 4 and 1 also coincide (nodes are equivalent), the previous backward reference is replaced again with

LZCS Transformation

```
NodeSigSet ← ∅
TextSigSet ← ∅
PreviousSubtree ← ∅
while there are more nodes do
      current_node ← get_node()    // in postorder
      if (current_node is a Text_Block)
        then
             current_signature ← MD5(current_node)
             if (current_signature ∈ TextSigSet)
               then
                    reference ← TextSigSet.reference(current_signature)
                    PreviousSubtree.add(reference)
                else
                    current_position ← StartPosition(current_node)
                    TextSigSet.add(current_signature, current_position)
                    Write PreviousSubtree in the output
                    Write current_node in the output
                    PreviousSubtree ← ∅
             fi
         else
             current_signature ← NodeSignature(current_node)
             if (current_signature ∈ NodeSigSet)
               then
                    reference ← NodeSigSet.reference(current_signature)
                    PreviousSubtree.erase_children(current_node)
                    PreviousSubtree.add(reference)
                else
                    current_position ← StartPosition(current_node)
                    NodeSigSet.add(current_signature, current_position)
                    Write PreviousSubtree in the output
                    Write current_node in the output
                    PreviousSubtree ← ∅
             fi
      fi
od
Write PreviousSubtree in the output
```

Figure4. LZCS transformation algorithm.

another that contains the structural element (Figure 5-B). The same happens to text block 7 (Figures 5-C and 5-D).

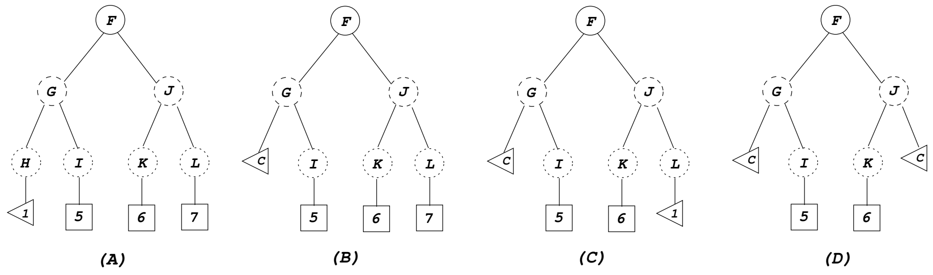


Figure5. Representation of substitutions performed in the second document.

Finally, the third document is processed. First, the substitutions of text blocks 8 and 9 are carried out, as well as those for their corresponding structural elements (Figures 6-A to 6-D). When structural element N has just been processed, it is verified that it can be completely replaced by a backward reference to J, because they are equivalent elements: They have the same number of children and children are equivalent one by one left to right (Figure 6-E). Finally, text block 10 is replaced by a backward reference since it is equivalent to text block 3 (Figure 6-F). In this case, structural element Q is not substituted because it is not equivalent to E.

5 Evaluation

The LZCS model was tested using different XForms collections, which correspond to real documents in use in small and medium Chilean companies. XForms[1], an XML dialect, is a W3C Candidate Recommendation for a specification of Web forms that clearly separate semantic from presentation aspects. In particular, XForms is becoming quite common in the representation and exchange of information and transactions between companies.

For privacy reasons we cannot use actual XForms databases, but we can get rather close. We have obtained five different types of forms (e.g., invoices). Each such form has several fields. Each field has a controlled vocabulary (e.g., names of parts) we have access to. Hence, we have generated actual forms by randomly choosing the contents of each field from their controlled vocabulary. We remark that this is pessimistic, since actual data may contain more regularities than randomly generated data.

A brief description of the five types of forms used follows.

[1] http://www.w3.org/MarkUp/Forms.

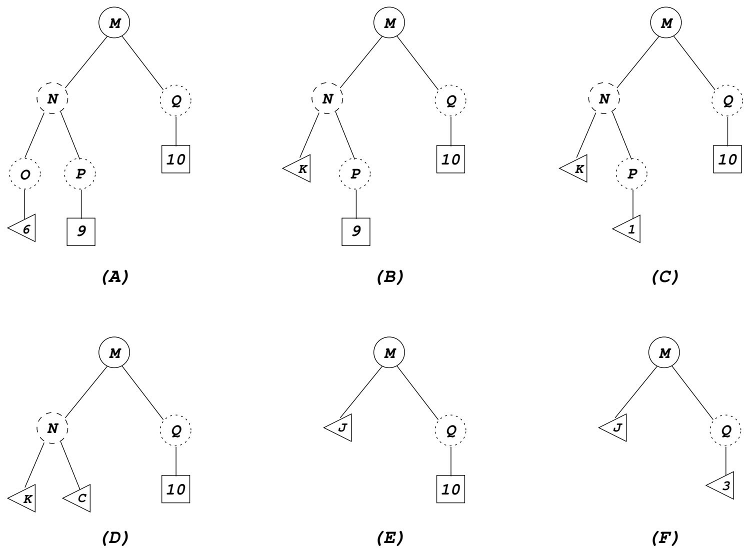


Figure6. Representation of substitutions carried out in the third document.

- XForms type 1: Centralization of Remunerations. It represents the accounting of the monthly remunerations, both for total quantities and with itemization. This is a frequently used document.
- XForms type 2: Sales Invoice. It is a legal Chilean document.
- XForms type 3: Purchase Invoice. It is a legal Chilean document, similar to the previous one.
- XForms type 4: Work Order. It is the document used in companies that install heating systems, to register the account detail of contracted work.
- XForms type 5: Work Budget. It is the document used in companies that build signs and publicity by request, to determine the parts and costs of works to carry out. Construction companies use a similar document.

For the experiments we selected different size collections of XForms types 1, 2 and 3. Collections of XForms types 4 and 5 were smaller so we used them as a whole.

In all cases, LZCS was tested with different l values. Value $l = 0$ means that all possible substitutions are made, whereas $l = \infty$ means that no text block is replaced, just structural elements.

Figure 7 shows how compression ratios evolve when different values for l are used, for XForms type 3. Other XForms collections give similar results. Compression ratio is defined as the compressed text size divided by the uncompressed text size. We do not yet apply further compression after the LZCS transformation.

As can be seen, the worst compression has been obtained in all cases for $l = 0$, this is, when all possible text blocks are replaced. Compression for $l = \infty$ has obtained intermediate results, obtaining on large collections reductions in text

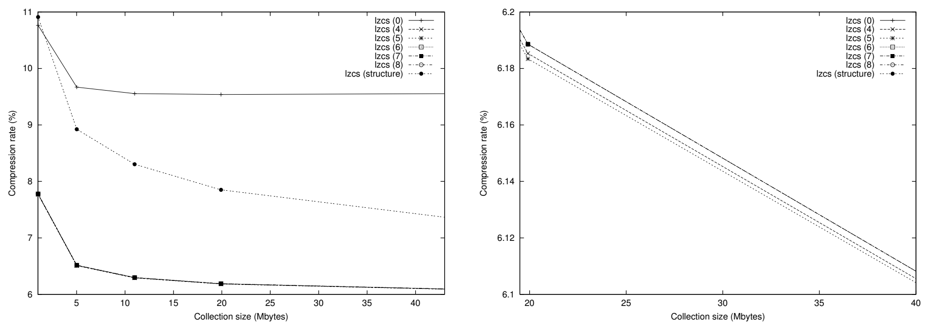


Figure7. Compression ratios using different values for l, for XForms type 3. Right representation is a zoom of left plot.

size of 28% compared to the option $l = 0$. However, choice $l = \infty$ is still much worse than intermediate choices. Different intermediate values for l yield similar compression, with very small variations. Their compression improves upon $l = \infty$ by 18% and upon $l = 0$ by 42% for large collection sizes.

Next, we compared LZCS against the basic word-based Huffman method [8]. Figure 8 shows the best compression ratio obtained for each method and for each document type. Column "LZCS (first stage)" indicates the compression obtained when the LZCS transformation is applied alone, while column "LZCS (complete)" indicates the compression obtained after applying Word Huffman to the output of the first stage.

Collection / Method	Word Huffman	LZCS (first stage)	LZCS (complete)
XForms 1	9.6935%	0.037432%	0.021522%
XForms 2	12.646%	4.3111%	0.92209%
XForms 3	11.550%	6.0872%	1.3294%
XForms 4	13.994%	4.8861%	0.89281%
XForms 5	12.441%	3.6245%	0.83933%

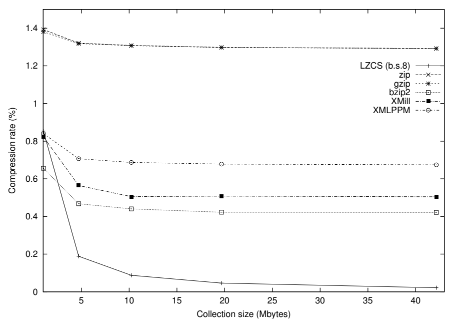
Figure8. Best compression ratios for each method and collection.

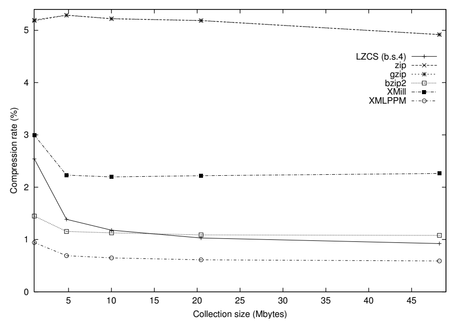
In all cases the compression obtained by LZCS transformation alone is surprisingly good. Let us remark that the output obtained by the transformation is still a plain text document. When Word Huffman codification is aplied over the transformed text the compression is still better, reducing the LZCS transformed text to 20%–60% of its size.

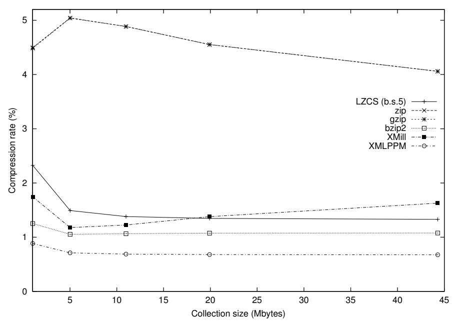
Finally, we compared LZCS against other compression systems that allow neither navigation nor random access on compressed file.

These compression systems either are structure-aware (like *XMill* and *XML-PPM* explained in Section 2), or they are standard. Most standard systems are based on classical LZ-schemes. Standard systems used to compare against LZCS are (1)*zip* and (2)*gzip*, using LZ77 plus a variant of Huffman algorithm; (3) *UNIX's compress*, that implements LZW algorithm; (4)*bzip2*, which uses the Burrows-Wheeler block sorting text compression algorithm, plus Huffman coding.

Bzip2 compression is generally considerably better than that achieved by more conventional LZ77/LZ78-based compressors, and approaches the performance of the PPM family of statistical compressors.







Method / Size	T.4 (7.19 Mb)	T.5 (5.74 Mb)
zip	2.1056%	4.4351%
gzip	2.1040%	4.4331%
compress	10.300%	10.396%
bzip2	0.95276%	0.84302%
XMill	0.94260%	0.92420%
XMLPPM	0.71240%	0.55300%
Word Huffman	13.994%	12.441%
LZCS $l = 0$	0.89281%	0.93944%
LZCS $l = 4$	0.89373%	0.84746%
LZCS $l = 5$	0.89672%	0.84630%
LZCS $l = 6$	0.90421%	0.83933%
LZCS $l = 7$	0.89704%	0.83935%
LZCS $l = 8$	0.89309%	0.84139%
LZCS $l = \infty$	1.9535%	1.2566%

Figure9. Comparison between LZCS and others, for XForms types 1 (upper left), 2 (upper right), 3 (bottom left), 4 and 5 (bottom right).

We compressed our collections with all the systems described. Compression ratios are shown in Figure 9.

Let us first consider the general compressors. Word Huffman and *compress* obtained the worst compression ratios, and they are not competitive in this experiment. They are followed by *zip* and *gzip*, both with very similar compression ratios. The best by far in this category is *bzip2*, which is still inferior to LZCS, in most cases by a slight margin. The reason for these results is that these four methods do not consider the structure of the documents, from which LZCS takes significant advantage. Also, we stress that LZCS allows navigation and random access over compressed text, which is not easy for *bzip2*.

Let us now consider the structure-aware methods. In general, LZCS is significantly better than *XMill* in all collections, producing compressed texts from just 5% smaller to as much as 25 times smaller. *XMLPPM*, on the other hand, obtains by far the best compression in most cases, except for the notable exception of XForms type 1, where LZCS is largely unbeaten. The problem of *XMLPPM* is that its compression is adaptive, and hence it is not suitable for navigation or random access on the compressed text.

6 Conclusions

We have presented LZCS, a compression scheme based on Lempel-Ziv which is aimed at compressing highly structured data. The main idea of LZCS is to replace whole substructures by previous occurrences thereof. The main advantages of LZCS are (1) very good compression ratios, outperforming all classical methods and most structure-aware methods; (2) easy random access, visualization and navigation of compressed collections; (3) fast and one-pass compression and decompression. Only *XMLPPM* compressed better than LZCS in our experiments, but random access to a particular document is impossible with *XMLPPM*, since it is adaptive and needs to decompress first all the documents that precede the desired one. This outrules *XMLPPM* for use in a compressed text database scenario.

One of the most challenging problems faced was the efficiency problem of the compression stage, which is quadratic if one follows the definition. We managed to overcome this problem and designed a linear average-time compression algorithm, by using a particular hashing scheme.

In many scenarios, new documents are added to the document collection, but these are never deleted or modified. LZCS can easily cope with insertion of new documents, but more research is needed in order to accomodate deletions and modifications of documents. It would also be interesting to design indexing schemes for fast searching of documents containing some given words or substructures, keeping in mind that the collection is compressed.

Acknowledgement. We thank Pablo Palma, from Hypernet Ltd. (Chile), for providing us with massive samples of almost-real data for the experiments.

References

1. J. Adiego, G. Navarro, and P. Fuente. Compressing semistructured text databases. In *Proc. European Conference on Information Retrieval (ECIR'03)*, LNCS 2633, pages 482–490. Springer, 2003.
2. J. Adiego, G. Navarro, and P. Fuente. SCM: Structural contexts model for improving compression in semistructured text databases. In *Proc. 10th Intl. Symp. on*

String Processing and Information Retrieval (SPIRE'03), LNCS. Springer, 2003. To appear.

3. T. Bell, J. Cleary, and I. Witten. *Text Compression.* Prentice Hall, Englewood Cliffs, N.J., 1990.
4. J. Bentley, D. Sleator, R. Tarjan, and V. Wei. A locally adaptive data compression scheme. *Communications of the ACM*, 29:320–330, 1986.
5. J. Cheney. Compressing XML with multiplexed hierarchical PPM models. In *Proc. Data Compression Conference (DCC 2001)*, pages 163–, 2001.
6. J. Dvorský, J. Pokorný, and V. Snásel. Word-based compression methods and indexing for text retrieval systems. In *Proc. ADBIS'99*, LNCS 1691, pages 75–84. Springer, 1999.
7. H. Liefke and D. Suciu. XMill: an efficient compressor for XML data. In *Proc. ACM SIGMOD 2000*, pages 153–164, 2000.
8. A. Moffat. Word-based text compression. *Software - Practice and Experience*, 19(2):185–198, 1989.
9. A. Moffat and R. Wan. RE-store: A system for compressing, browsing and searching large documents. In *Proc. 8th Intl. Symp. on String Processing and Information Retrieval (SPIRE 2001)*, pages 162–174, 2001.
10. G. Navarro, E. Silva de Moura, M. Neubert, N. Ziviani, and R. Baeza-Yates. Adding compression to block addressing inverted indexes. *Information Retrieval*, 3(1):49–77, 2000.
11. R. Rivest. The MD5 message-digest algorithm. RFC 1321. MIT Laboratory for Computer Science and RSA Data Security, Inc., April 1992.
12. P. Tolani and J. Haritsa. XGRIND: A query-friendly XML compressor. In *Proc. of 18th International Conference of Data Engineering (ICDE'02)*, pages 225–234, 2002.
13. Terry A. Welch. A technique for high-performance data compression. *IEEE Computer*, 17(6):8–19, 1984.
14. I. Witten, A. Moffat, and T. Bell. *Managing Gigabytes.* Morgan Kaufmann Publishers, second edition, 1999.
15. J. Ziv and A. Lempel. An universal algorithm for sequential data compression. *IEEE Trans. on Information Theory*, 23(3):337–343, 1977.
16. Jacob Ziv and Abraham Lempel. Compression of individual sequences via variable-rate coding. *IEEE Transactions on Information Theory*, IT-24(5):530–536, 1978.
17. N. Ziviani, E. Moura, G. Navarro, and R. Baeza-Yates. Compression: A key for next-generation text retrieval systems. *IEEE Computer*, 33(11):37–44, November 2000.