# Lempel-Ziv Compression of Highly Structured Documents *†

Joaquín Adiego ‡ Gonzalo Navarro § Pablo de la Fuente‡

**Abstract**

We describe a novel Lempel-Ziv approach, called *LZCS*, suitable for compressing structured documents. LZCS takes advantage of repeated substructures that may appear in the documents, by replacing them with a backward reference to their previous occurrence. The result of the LZCS transformation is still a valid structured document which is human-readable and can be transmitted by ASCII channels. Moreover, LZCS transformed documents are easy to search, display, access at random, and navigate. In a second stage, the transformed documents can be further compressed using any semistatic technique, so that it is still possible to do all those operations efficiently, or with any adaptive technique to boost compression. LZCS is especially efficient to compress collections of highly structured data, such as XML forms, invoices, e-commerce and web-service exchange documents. The comparison against other structure-aware 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. When joined to an adaptive compressor, LZCS obtains by far the best compression ratios.

*Keywords:* Lempel-Ziv, XML Data, Structured Documents, 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 business-to-business applications 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.

*This work was partially supported by CYTED VII.19 RIBIDI project (all authors); TIC2003-09268 project, MCyT, Spain (first and third authors); and Millennium Nucleus Center for Web Research, Grant P01-029-F, Mideplan, Chile (second author).

†A preliminary version of this article appeared in *Proc. 14th IEEE Data Compression Conference (DCC)*, pages 112–121, 2004.

‡GRINBD, Departamento de Informática, Universidad de Valladolid, Valladolid, Spain. `{jadiego, pfuente}@infor.uva.es`

§Center for Web Research, Department of Computer Science, University of Chile, Santiago, Chile. `gnavarro@dcc.uchile.cl`

Although the information stored by an organization is usually kept in relational databases and/or data warehouses, it is important to store digital sources, 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 in compressed form. The most effective compression methods do not account for these capabilities: texts have to be uncompressed before they can be accessed.

It is usually argued that disk space is cheap and thus compression is not interesting. Compression, however, does not only save space. It saves disk and network transfer time, which are highly valuable resources. Hence the interest of compression by itself. Moreover, the types of texts we are focusing on in this paper are highly compressible: We will show that we can compress them to 1% of their original size. With this compression ratio, it is for example possible that we can load the compressed text database in main memory, albeit we are unable to decompress it wholly in main memory. Hence the interest of manipulating and navigating the structure in compressed form, extracting only the documents we actually need.

We develop a compression method, *Lempel-Ziv to Compress Structure (LZCS)*, inspired in Lempel-Ziv compression, where repeated substructures are factored out. That is, every time a repeated substructure is detected, it is replaced by a backward reference to its previous occurrence. The result of this *LZCS transformation* is a text that is still human-readable and well structured. Thus, it can be seamlessly transmitted over ASCII channels, handled by structured text management tools, and visualized in compressed form with conventional means. It is very fast and simple to decompress, in whole or in parts, it can be searched for the presence of words and phrases in the text with conventional algorithms directly in compressed form (about 100 times faster than the original text), and it can be accessed at random without need of decompressing the preceding text. With little additional effort, the compressed document can be browsed and navigated without decompressing it.

Compared to LZ77 [ZL77], which can factor out *any* repeated text substring, LZCS is restricted to consider only whole substructures. As a result, LZ77 compresses more than the LZCS transformation, yet the compressed text lacks all of the LZCS features described above, except for the fast decompression. It is interesting that we build on an adaptive compressor (LZ77) not permitting local decompression, and obtain a compressor that does permit local decompression, navigation, and many other features.

To improve compression, the LZCS transformed text can be further compressed with a classical compressor. The use of a semistatic compressor retains fast decompression in whole or in parts, random access, and the possibility of browsing and navigating the compressed document. Alternatively, an adaptive compressor can boost the compression ratio, yet losing all those features.

In particular, we show that the use of a semi-static word-based Huffman method to compress the LZCS transformed text yields very competitive compression ratios, only beaten by adaptive schemes that do not permit any of the features we have described above. Adaptive schemes are suitable to compress an archival collection, but not a database of documents that must frequently retrieve individual documents. On the other hand, we show that the combination of LZCS and an adaptive PPM compressor is unbeaten in compression ratio.

We show how the LZCS transformation can be carried out in linear expected time and in a single pass over the text. This means that we can start producing the transformed text shortly after starting reading the source text. This makes LZCS suitable for use over a communication network without introducing any delay in the transmission. For example, LZCS can be transparently used to transmit structured documents, even over a plain ASCII channel, in order to reduce communication time. The receiver needs very little computational power to uncompress, and it can even navigate or display parts of the document without uncompressing all of it.

The paper is organized as follows. In Section 2 we cover related work on compression, both for plain and structured text. In Section 3 we describe the LZCS transformation. In Section 4 we explain how the transformation can be carried out in linear expected time. In Section 5 we show empirical results comparing the compression ratio, as well as compression and decompression performance, of LZCS compared to other standard and structure-aware compressors. We conclude in Section 6 with future work directions.

## 2 Related Work

### 2.1 Standard Text Compression

In general, classic text compression methods [BCW90, MT02] do not take into account the structure of the documents they compress. Our aim is not to cover the whole area but just to focus on three families of compressors that are relevant for this paper. *Lempel-Ziv* and *k-th order modelling* families are *adaptive* compressors, which learn the statistical structure of the text as they process it, updating the model on the fly. *Huffman* family is *semistatic*, that is, it first obtains the statistics of the whole text and then compresses all the text with the same model.

**Lempel-Ziv.** At the end of the seventies, Lempel and Ziv designed new technologies of data compression based on replacing text substrings by previous repeated occurrences. Their two most famous algorithms are called LZ77 [ZL77] and LZ78 [ZL78]. A well-known variant of the latter is called LZW, by Welch [Wel84].

LZ77 maintains a window of the last $N$ processed characters. In each step, it reads the longest possible string $s$ from the input that also appears in the window. If $s$ is of length $\ell$, it is followed by character $a$ in the input, and it was found at window position $p$ (counting right to left), then the compressor outputs the triple $(p, \ell, a)$. Thus input string $sa$ is replaced by the triple, and compression is obtained if the triple needs less bits than the string itself. Once this is done, the window is shifted forward by $\ell + 1$ positions and the algorithm resumes the scanning just past string $sa$.

In principle a longer window improves compression because it is more likely to find longer strings for replacement. However, the representation of position $p$ requires $\log_2 N$ bits, which worsens as $N$ grows. In practice the most convenient window size is not very long (for example, 64 Kbytes).

Decompression of LZ77 is extremely fast and simple. The compressed text is basically a sequence of triples $(p, \ell, a)$. For each such triple we must copy $\ell$ characters starting $p$ positions behind the current output position, and then output $a$. Well-known representatives of LZ77 compression are Info-ZIP's *zip* and Gnu's *gzip*.

Other variants, such as LZ78 and LZW, restrict somehow which previous strings can be referenced. This is done for efficiency reasons of different type, for example to improve compression time or to improve the compression ratio. The choice of strings that can be referenced, however, does not take into account the meaning of those strings. A well-known representative of LZW is Unix's *compress*.

The Lempel-Ziv family is the most popular to compress text because it combines acceptable compression ratios (around 35% on plain English text[1]) with fast compression and decompression. However, being adaptive, Lempel-Ziv compressed text cannot be decompressed at random positions, because one must process all the text from the beginning in order to learn the window that is used to uncompress the desired portion.

**Huffman.** Classical Huffman compression [Huf52] consists of computing the frequencies of the text characters in a first pass, and then assign a variable-length bit-wise code to each character. Then, in a second pass, each character is replaced by its code. Huffman compression reaches the zero-order entropy of the text up to one extra bit per symbol, and being semistatic, it is easy to decompress the text starting at any position. Huffman is said to be a *statistical* compressor, as it relies on text statistics, as opposed to the so-called *dictionary-based* compressors which, as Lempel-Ziv, consist in replacing strings by identifiers.

Huffman is not very popular in text compression because it achieves poor compression ratios compared to other techniques. However, the situation changes drastically when natural language text is compressed and one uses the text *words*, rather than the characters, as the text symbols [Mof89]. The distribution of words is much more skewed than that of symbols, and this permits obtaining much better compression ratios than Huffman-based compressors. On English text, for example, character-based Huffman obtains around 60% compression ratio, while word-based Huffman is around 25% [ZMNBY00]. Actually, similar compression ratios can be obtained by using Lempel-Ziv on words [BSTW86, DPS99].

Word-based Huffman, however, has other advantages. Not only the text can be compressed and decompressed efficiently, as a whole or in parts, but it is also possible to search it without decompressing, *faster* than when searching the uncompressed text [ZMNBY00]. Another advantage is that this type of compression integrates very well with information retrieval systems, because the source alphabet is equivalent to the vocabulary of the inverted index [WMB99, NMN$^+$00, MW01].

One of the best known systems in the public domain relying on word-based Huffman is the MG system [WMB99].

***K*-th order models.** This family of statistical adaptive compressors comprises both Prediction by Partial Matching (PPM) compression and the Burrows-Wheeler Transform (BWT).

PPM [CW84] is a statistical compressor that models the character frequencies according to the *context* given by the $k$ characters preceding it in the text (this is called a $k$-th order model), as

[1] That is, the compressed text size is 35% of the uncompressed text size.

opposed to Huffman that does not consider the preceding characters. Moreover, PPM is adaptive, so the statistics are updated as the compression progresses. The larger $k$, the more accurate is the statistical model and the better the compression, but more memory and time is necessary to compress and uncompress.

More precisely, PPM uses $k+1$ models, of order 0 to $k$, in parallel. It usually compresses using the $k$-th order model, unless the character to compress has never been seen in that model. In this cases it switches to a lower-order model until the character is found. The coding of each character is done with an arithmetic compressor, according to the computed statistics at that point.

The BWT [BW94] is a reversible permutation of the text, which puts together characters having the same $k$-th order context (for any $k$). Local optimization (for example, move-to-front followed by Huffman) over the permuted text obtain results similar to $k$-th order compression.

PPM and BWT usually achieve better compression ratios than other families (around 20% on English text), yet they are much slower to compress and decompress, and cannot uncompress arbitrary portions of the text collection. Well known representatives of this family are Seward's *bzip2*, based on the BWT, and Shkarin/Cheney's *ppmdi* and Bloom/Tarhio's *ppmz*, two PPM-based techniques.

### 2.2 Structured Text Compression

There exist a few approaches specifically designed to compress structured text, taking advantage of its structure.

**XMill [LS00].** Developed at AT&T Labs, XMill is an XML-specific compressor designed to exchange and store XML documents. Its compression approach is not intended for directly supporting querying or updating the compressed documents. XMill is based on the *zlib* library, which combines Lempel-Ziv compression with a variant of Huffman. Its main idea is to split the file into three components: elements and attributes, text, and structure. Each component is compressed separately. Another compressor based Lempel-Ziv, cutting the structure at some depth and using plain Lempel-Ziv compression for the subtrees, is commercial *XMLZip* (`http://www.xmls.com`).

**XMLPPM [Che01].** This compressor uses a PPM-like coder, where the context is given by the path from the root to the tree node that contains the current text. This is an adaptive compressor that does not permit random access to individual documents. The idea is an evolution over XMill, as different compressors are used for each component, and the XML hierarchy information is used to improve compression.

**XCQ [LW02] and Exalt [Tom04].** These are compression methods based on separating structure from data, and using grammar-based compression for the structure. In *XCQ*, the tree shape is compressed using the DTD information, while the text is compressed using a standard Lempel-Ziv software such as *gzip*. In *Exalt*, both elements are compressed using grammar-based methods. In particular, zero-order prediction depending on the structural context, plus arithmetic coding, is used for the tags. Other grammar-based techniques can be found in [Tar01], as well as in *XML-Xpress*, a commercial software (`http://www.ictcompress.com`) that compresses well when the DTD is known.

**XGrind [TH02].** This compressor is interesting because it 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. Structure tags are represented in numeric form, while the text is compressed using character-oriented Huffman. A similar idea is explored in in *XMillau* [GS00].

**SCMHuff [ANdlF03] and SCMPPM [AdlFN04].** SCM 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. SCMHuff uses a word-based Huffman compressor for each different tag, while SCMPPM uses a PPMDI compressor. The former permits random access to individual documents, while the latter cannot.

# 3 The LZCS Transformation

LZCS is a new technique to compress structured text (such as XML or HTML). The main idea is based on the Lempel-Ziv concept, so that repeating substructures and whole text blocks (that is, the whole text inside a structure or between two structural elements) are replaced by a backward reference to their first occurrence 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 classical compressor.

We start by formally describing the LZCS transformation, then present an example, and finally discuss its features.

## 3.1 Formal Definition

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

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

Observe that a text block is either the whole text contained in a structural element which does not have further internal structure, or it is the whole text between two consecutive structural elements. On the other hand, a structural element can contain one or more text blocks, one or more structural elements and/or (after the LZCS transformation) one or more backward reference tags. For simplicity, other types of valid tags (such as, in XML, comment tags and self-contained tags) will be treated as conventional text, and only start-tags and end-tags will be used to identify structural elements. Furthermore, tags will be treated as atomic elements. This means that, for example, the XML attributes and values inside a tag are part of the tag name, and do not form text blocks.

The structure induces a hierarchy that can be represented as a tree. Text blocks will be represented by leaves, and structural elements by subtrees rooted at internal nodes.

**Definition 3 (Node)** *A* node *is either a text block or a structural 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 $\mathcal{N}_2$.*

We are ready to define the LZCS transformation.

**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 transformed 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 delimiters "<@" 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 in the transformed text 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. This way, the transformed text is still ASCII and well-structured. The reference tag has been chosen to void tag name clashes in XML, but it can be changed.

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 elements of type 1 (A, F, M) have their circle drawn with a solid line, those of type 2 (B, E, G, J, N) with a dashed line, and those of type 3 (the rest) with a dotted line. Text blocks are represented by squares. Letters and numbers in the figure represent node identifiers.

To cover all the possibilities, assume 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. As a result, the documents share repeating parts (that is, equal subtrees). Figure 2 shows graphically these correspondences and Figure 3 shows the collection transformed with LZCS.

Finally, Figure 4 shows a textual version of the original and transformed documents. Note that the LZCS transformed text is a valid structured document, provided we accept "<@...>" as a valid self-contained tag.

## 3.3 Properties of the LZCS Transformed Text

As mentioned in the Introduction, the LZCS transformation has a number of attractive features, which we describe now more in depth.

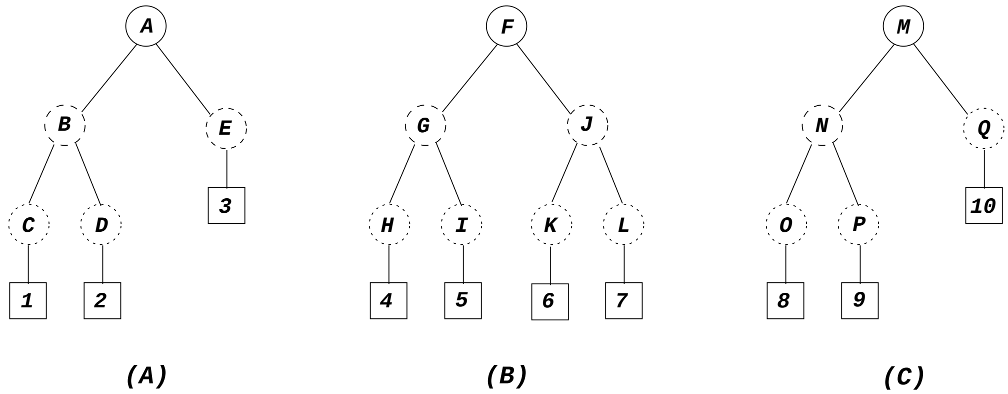



Figure 1: Three example documents.

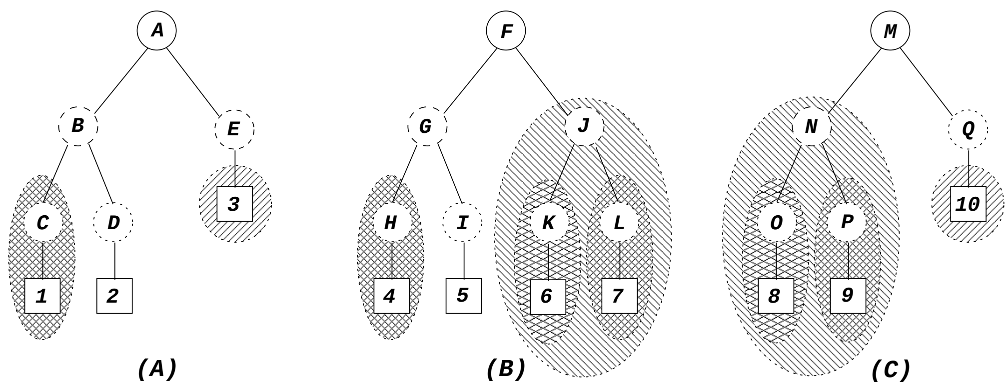



Figure 2: Equivalent subtrees of the documents.

**Human readable:** The output of the transformation is human-readable (see Figure 4). This means that the transformed file can be read with any conventional text editor or terminal.

**ASCII compliant:** The only new characters introduced by LZCS are `'<'`, `'>'`, `'@'`, letters and digits. Therefore, and LZCS transformed document can be transmitted by any ASCII channel. For example it can be sent by email without any concern. Actually LZCS could be transparently used by servers to transfer structured documents to clients, even over ASCII channels.

**Well structured:** The LZCS transformed text is a well formed structured document. As such, it can be handled with any tool that manages structured documents (in XML, for example). The only exception is that LZCS produces a special self-contained tag, `"<@...>"`, which must be dealt with as any other such tag. We could perfectly use instead a conventional self-contained tag to avoid any exception, such as `"<ref pos=.../>`, but we chose otherwise to avoid any possibility of clashing with the actual tags of the documents, and to have shorter references.

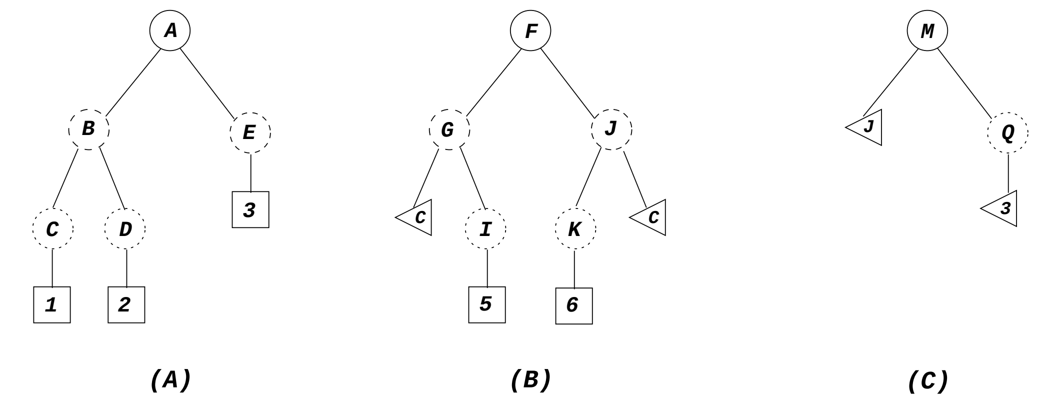



Figure 3: Example documents after applying the LZCS transformation. Backward references are represented by triangles.

**Directly searchable:** The LZCS transformed text contains the same words and phrases of the original documents. A phrase cannot be split unless its words belong to different structural elements, in which case it is arguably not a phrase. Although the number of occurrences of words and phrases will change between the original and the transformed documents, a word or phrase is present in the original text if and only if it is present in the transformed text. Thus, the LZCS transformed text can be searched for words and phrases with any conventional string matching algorithm (such as Gnu's *grep*) to determine whether the phrase appears or not. If the phrase appears, decompression is necessary to point out all the documents where they appear. Note in particular that the search on the LZCS transformed text will be *faster* than on the original text, as the latter is longer (in our experiments, 100 times longer).

**Fast to decompress:** Decompressing an LZCS transformed text is pretty much as decompressing LZ77, and therefore, very fast and simple. An important difference is that LZ77 uses pointers to the uncompressed file, so it can just copy the referenced uncompressed text to the output. LZCS, on the other hand, uses pointers to the compressed file, so it must recursively obtain the output text from the compressed file. This makes LZCS decompression somewhat slower, but in exchange LZCS can navigate the compressed file and extract individual documents without uncompressing the whole text.

**Easily navigable and visualizable:** LZCS transformed documents can be navigated in the usual way (that is, going down and up in the hierarchy as with a tree). Instead of relying on any kind of parent pointer associated to nodes, we must use a stack to keep track of the current ancestors of the current node. Every time we have to go down to a child, it might be that the child is a backward reference or not. In the former case, we just move the current text position to the appropriate point back in the compressed file. All the rest is unchanged. When moving upwards, we pop the corresponding file position from the stack of ancestors.

**Accessible at random positions:** With the same algorithm above we can produce the uncompressed text of any document, by simply starting uncompression at its start-tag and following

```
A: <log>
B:    <entries>
C:       <event>
1:          Bug report
         </event>
D:       <event>
2:          Release announce
         </event>
      </entries>
E:    <entries>
3:       No further events
      </entries>
   </log>
F: <log>
G:    <entries>
H:       <event>
4:          Bug report
         </event>
I:       <event>
5:          New version
         </event>
      </entries>
J:    <entries>
K:       <event>
6:          Bug fix
         </event>
L:       <event>
7:          Bug report
         </event>
      </entries>
   </log>
M: <log>
N:    <entries>
O:       <event>
8:          Bug fix
         </event>
P:       <event>
9:          Bug report
         </event>
      </entries>
Q:    <event>
10:      No further events
      </event>
   </log>
```

```
A: <log>
B:    <entries>
C:       <event>
1:          Bug report
         </event>
D:       <event>
2:          Release announce
         </event>
      </entries>
E:    <entries>
3:       No further events
      </entries>
   </log>
F: <log>
G:    <entries>
H:       <@C>
I:       <event>
5:          New version
         </event>
      </entries>
J:    <entries>
K:       <event>
6:          Bug fix
         </event>
L:       <@C>
      </entries>
   </log>
M: <log>
N:    <@J>
Q:    <event>
10:      <@3>
      </event>
   </log>
```

Figure 4: The same example documents in textual form. The original document is on the left and the LZCS transformed document on the right. For readability we write references to line labels (uppercase letters and numbers) instead of character offsets. We remind that the references are offsets in the compressed text, not in the original text.

any reference as necessary.

Thus, LZCS can be integrated into a structured text retrieval system without loss (and in cases large gains) of efficiency in the search or visualization of results. As demonstrated in our experiments, the compression ratios are so good (1%) that it is feasible to maintain large collections compressed in main memory, even when there is no enough main memory to uncompress all of it. LZCS is perfect for this scenario, as it can navigate, visualize and uncompress individual document without having to uncompress the whole collection.

The LZCS transformed text can be further compressed with any conventional method. Since the documents generated by LZCS are navigable, a good idea is to further compress them using a semistatic compression method, like word-based Huffman. After this process, the documents cannot anymore be handled as plain text (a word-wise decompression is needed), but they are still navigable and accessible at random positions. Direct search over word-based Huffman is also possible and very efficient. On the other hand, we can use an adaptive compression to boost compression ratio. LZCS can be seen as a preprocessing stage that factors out some types of redundancies, so that a further adaptive compressor takes much less time and compresses more than when applied over the original text.

## 4 Efficient Implementation of the LZCS Transformation

A challenge with the LZCS transformation is how to implement it efficiently, as we must detect substructures that have appeared in the past. The simplest way to implement the LZCS transformation is by searching all previously processed text for each new structural element. This way, we have a complexity of $O(n^2)$, which is unacceptable.

We show now how to obtain $O(n)$ average time. The idea is to maintain a hash table with all the whole text blocks, as well as all the structural elements, seen in the past. While hashing text blocks is straightforward, recognizing repeated structural elements in linear expected time requires more careful design.

When a text block is processed, we first obtain its digital signature (for example, using MD5 algorithm [Riv92]). 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 to 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 in the output). Otherwise, if an equivalent text block appears (their digital signatures coincide) a backward reference to the first occurrence of the text block is written to 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 in the output, for nodes that have not appeared before. Node signatures of parent nodes are produced after those of children nodes.

**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 in the output if they are not references, or their referenced positions otherwise.*

As we show in Lemma 2, a node signature is unique within 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 $\mathcal{N}$ and $\mathcal{N}'$ be two nodes that appear in a collection transformed with LZCS up to node $\mathcal{N}'$, $\mathcal{N}$ preceding $\mathcal{N}'$. Then, $\mathcal{N}$ is equivalent to $\mathcal{N}'$ iff $\mathcal{N}'$ is a backward reference to $\mathcal{N}$, or $\mathcal{N}$ and $\mathcal{N}'$ are equal backward references.*

**Proof:** We prove the equivalence in both directions.

1. If $\mathcal{N}$ is equivalent to $\mathcal{N}'$ then the LZCS transformation replaces $\mathcal{N}'$ by a backward reference to its first occurrence:

   (a) If $\mathcal{N}$ is the first occurrence then $\mathcal{N}'$ is replaced by a backward reference to $\mathcal{N}$.

   (b) Otherwise, let $\mathcal{N}_0$ be the first occurrence of $\mathcal{N}'$, then $\mathcal{N}'$ is replaced by a backward reference to $\mathcal{N}_0$, but also $\mathcal{N}$ was replaced by a backward reference to $\mathcal{N}_0$.

   Thus, it holds that either $\mathcal{N}'$ is a backward reference to $\mathcal{N}$, or $\mathcal{N}$ and $\mathcal{N}'$ are equal backward references.

2. If $\mathcal{N}'$ is a backward reference to $\mathcal{N}$, or $\mathcal{N}'$ and $\mathcal{N}$ are equal backward references, then $\mathcal{N}$ is equivalent to $\mathcal{N}'$, because in both cases it holds that $\mathcal{N}$ and $\mathcal{N}'$ contents are textually equal. □

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 repeat if all its children repeat as well. 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 $\mathcal{N}_0$. According to Definition 6, in the first case both children identifiers are $\mathcal{N}_i$, and in the second both are $\mathcal{N}_0$. These conditions are necessary and sufficient for the node signatures of $\mathcal{N}$ and $\mathcal{N}'$ being equal. □

We are now ready to explain the 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 and current node must be written to the output. Also, the current node signature is added to the set of node signatures.

**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 to the output
                      Write current_node to 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 to the output
                      Write current_node to the output
                      PreviousSubtree ← ⟨ ⟩
              fi
      fi
od
Write PreviousSubtree to the output
```

Figure 5: LZCS transformation algorithm.

Figure 5 describes the basic LZCS transformation. List *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.

Also note that, if a subtree is not repeated, then no ancestor of it can be repeated. As all the elements in *PreviousSubtree* have not yet been sent to the output just because it might be that their parent (an ancestor of the current node) might be repeated, as soon as we know that the current node is not repeated we send all *PreviousSubtree* to the output. This is not strictly necessary (one could only send the children of the current node to the output, and previous elements would wait that their parent sends them) but it simplifies the algorithm, as the list to maintain is shorter and always composed of references.

Decompression is very simple. It begins by writing the text to the output. When a backward reference tag is found, we recursively start decompression from the referenced position in the compressed text. If the text at that position begins with a start-tag, the recursive call will finish when the corresponding end-tag is written. Otherwise, it will finish when the first start-tag appears. Upon returning from the recursive call, the main process resumes decompression from past the backward reference tag. Recursion is necessary because further backward references may appear when processing the text referenced by the first one.

Figure 6 gives the pseudocode. This is simplified, for example it is implicit that matching the "corresponding end-tag" that finishes a reference involves keeping track of the current depth in the structure tree.

Note also that uncompression could be faster and simpler if we stored pointers to references in the untransformed file, rather than in the transformed file. In this way, there would be no recursion because the referenced text would be already untransformed. We recall that this, however, prevents navigating in the transformed file without decompressing it.

About memory usage, both the compression and decompression algorithm work better if they maintain all the compressed text in main memory (although they could work with the text on disk). In addition, the compressor needs to maintan the hash tables for text block and node signatures. Note that items are inserted into those tables only when they do not become references but pass to the output, so the space required for those tables is also proportional to the size of the compressed text. The size of *PreviousSubtree* and stacks is negligible. Just like other compressors, LZCS can clean up all its structures and start afresh when the memory consumption exceeds some predefined limit. This would only affect compression ratio, but not correctness.

## 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 7-A.

**LZCS Inverse Transformation**

```
word ← get_word()
while not end of transformed text do
        if (word is a reference tag)
           then position ← get_position(word)
                   SolveReference(position)
            else write word to the output
        fi
        word ← get_word()
od
```

```
procedure SolveReference(position)
do
    go to position in input file
    word ← get_word()
    if (word is a start-structure tag)
       then end_word ← corresponding end-structure tag
        else end_word ← any start-structure tag
    fi
    while word ≠ end_word do
            if (word is a reference tag)
               then position ← get_position(word)
                       SolveReference(position)
                else write word to the output
            fi
            word ← get_word()
    od
od
```

Figure 6: LZCS inverse transformation algorithm.

As the structural elements that contain blocks 4 and 1 also coincide (nodes are equivalent), the previous backward reference is replaced again with another that contains the structural element (Figure 7-B). The same happens to text block 7 (Figures 7-C and 7-D).

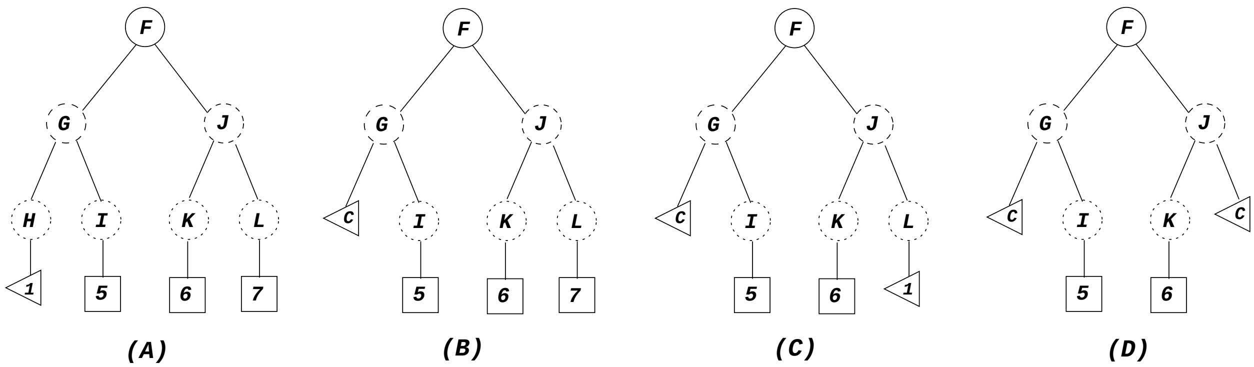



Figure 7: Substitutions performed in the second document.

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 8-A to 8-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 8-E). Finally, text block 10 is replaced by a backward reference since it is equivalent to text block 3 (Figure 8-F). In this case, structural element Q is not substituted because it is not equivalent to E.

The crux of Lemma 2 is illustrated at this point. Note that we detect that the subtree rooted at N in Figure 8-D is a repetition of the subtree rooted at J in Figure 7-D. The left subtree of node J is not a backward reference, so its signature is the very same position of K in the compressed text (let us call it $k$). The left subtree of node N is a backward reference pointing precisely to $k$. The right subtrees of J and N are both a backward reference equal to $c$, the position of node C in the compressed text. According to Definition 6, both signatures are equal to (type-1:$k$:$c$) and thus the equivalence is detected.

## 5 Experimental Evaluation

LZCS compression was tested using different XForms collections, which correspond to real documents in use in small and medium Chilean companies. XForms (`http://www.w3.org/MarkUp/Forms`), 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

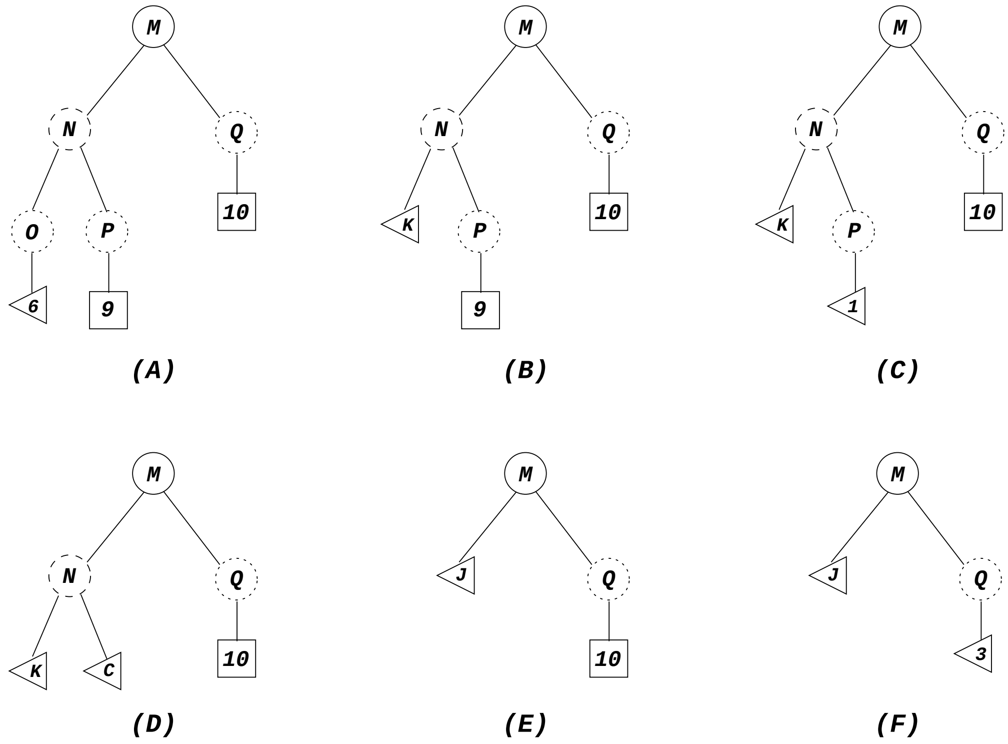



Figure 8: Substitutions carried out in the third document.

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.

- 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 subcollections of XForms types 1, 2, and 3. Collections of XForms types 4 and 5 were smaller so we used them as a whole.

### 5.1 Optimizing the Choice of $l$

We tested LZCS 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 9 shows how compression ratios evolve when different values for $l$ are used, for XForms type 3. Other XForms collections give similar results. We remind that "compression ratio" refers to the size of the compressed text divided by the size of the uncompressed. We do not yet apply further compression after the LZCS transformation.

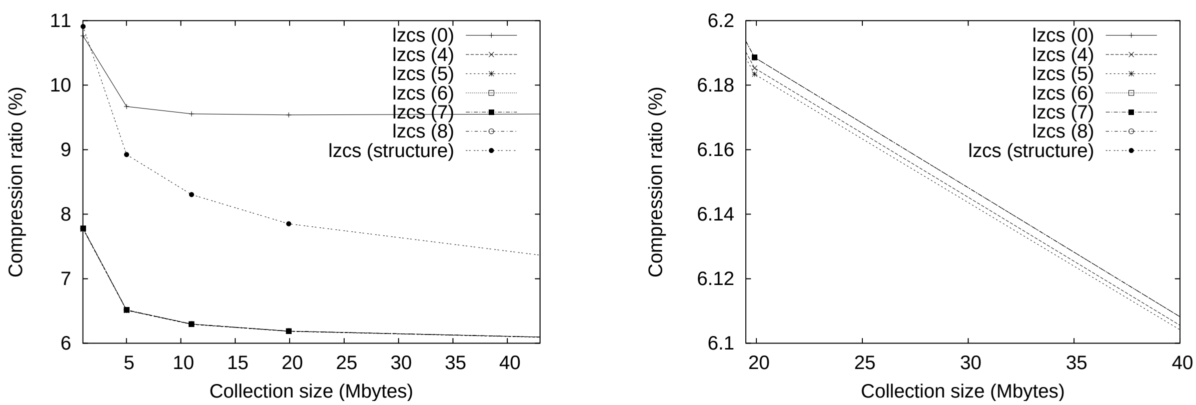



Figure 9: Compression ratios using different values for $l$, for XForms type 3. On the right we show a zoom of the left plot. By "lzcs(structure)" we refer to the setting $l = \infty$.

As it 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 size reductions 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. This shows that most reasonable intermediate values of $l$ are almost optimal and thus fine-tuning of $l$ is not an issue.

We note that our XForms collections are highly compressible, as expected from this densely structured data.

### 5.2 Comparison against Classical Compressors

We first compared LZCS against the basic word-based Huffman method [Mof89] (*Word Huffman*, from the *MG* system, `http://www.cs.mu.oz.au/mg`). We separate this comparison from the rest because word-based Huffman is one of the methods we use for the second step after the LZCS transformation, and because word-based Huffman compression still permits random access to the compressed text. For LZCS, we use the best $l$ value for each collection.

Figure 10 shows the compression ratio obtained for each method and for each document type. Column "LZCS" indicates the compression obtained when the LZCS transformation is applied alone, while column "LZCS+Huff" indicates the compression obtained after applying word-based Huffman to the output of the first stage.

| Collection / Method | Word Huffman | LZCS | LZCS+Huff |
|---|---|---|---|
| XForms 1 | 9.6935% | 0.1760% | 0.05867% |
| XForms 2 | 12.646% | 4.3111% | 0.92209% |
| XForms 3 | 11.550% | 6.0872% | 1.32940% |
| XForms 4 | 13.994% | 4.8861% | 0.89281% |
| XForms 5 | 12.441% | 3.6245% | 0.83933% |

Figure 10: Compression ratios for LZCS versus *Word Huffman*.

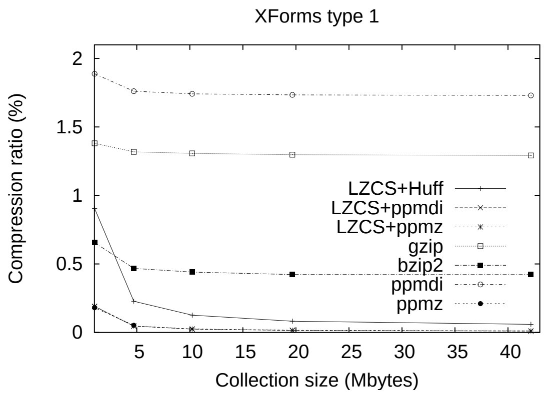
In all cases the compression obtained by LZCS transformation alone is remarkably good. Let us remind that the output obtained by the transformation is still a plain text document, and this already halves the space needed by *Word Huffman*, at the very least. When word-based Huffman coding is applied over the LZCS transformed text the compression is still better, reducing the LZCS transformed text to 20%–25% of its size.

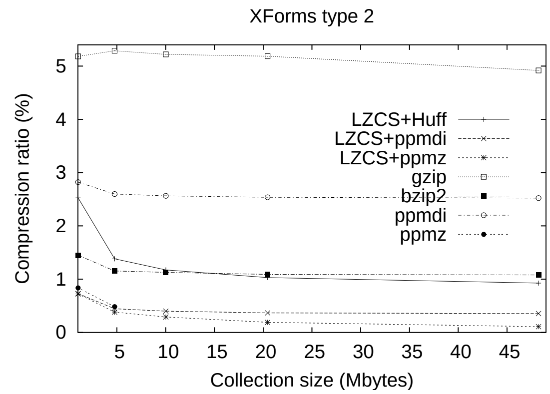
We now compare LZCS against other classical compression systems that allow neither navigation nor random access in the compressed file. Because of this, we consider three variants: LZCS+Huff, LZCS+ppmdi, and LZCS+ppmz. These consist in applying, respectively, word-based Huffman, PPMDI, and PPMZ compression (see next) to the LZCS transformed text. We use $l = 5$ in all the following experiments.

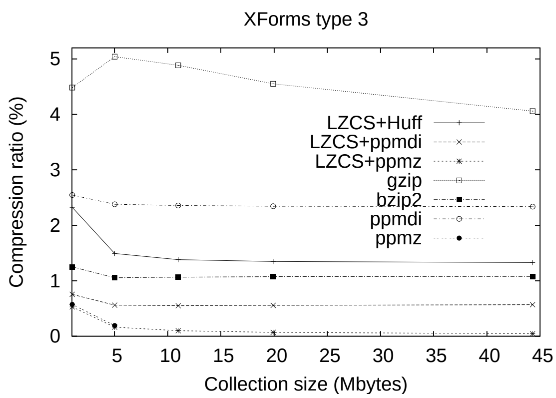
Standard systems used to compare against LZCS are (1) *gzip* v.1.3.5 (`http://www.gnu.org`), which use LZ77 plus a variant of Huffman algorithm (we also tried *zip* with almost identical results); (2) *UNIX's compress* v.4.2.4, which implements LZW algorithm; (3)*bzip2* v.1.0.2 (`http://www.bzip.org`), which uses the Burrows-Wheeler block sorting text compression algorithm, plus Huffman coding; (4)*ppmdi* (extracted from *XMLPPM 0.98.2*, `http://sourgeforge.net/projects/xmlppm`) and *ppmz* v.9.1 (Linux port, `http://www.cs.hut.fi/~tarhio/ppmz`), two PPM compressors. We used standard options for all (yet, letting them use much more memory did not significantly affect the results).

Compression ratios are shown in Figure 11. *Ppmz* compresses much better than *ppmdi*, but it is much slower. For example, it took from 4.5 to 10 hours to compress 5 megabytes of text with *ppmz*. For this reason, we show *ppmz* compression only for the first 5 Mb of XForms 1, 2, and 3, and for the whole XForms 4 and 5. On the other hand, LZCS+ppmz is much faster because *ppmz* is applied over the already transformed text, which is much smaller. As we see in the results, LZCS+ppmz obtains the best compression ratios. It even outperforms *ppmz* alone in many cases, at least for short texts. For longer texts, *ppmz* is simply not a choice. This shows that LZCS serves as a preprocessing stage that maintains (and even improves) the performance of *ppmz*, at the same time dramatically reducing the time needed for compression, at the point of making it a viable alternative for text sizes where *ppmz* alone is not.

The worst performing compressor is *compress*, with compression ratios around 10% in all the texts. This is similar to *Word Huffman* (which in exchange permits random access) and not competitive in this experiment (it is excluded from the plots of XForms types 1, 2, and 3 for readability).




| Method / Size | T.4 (7.19 Mb) | T.5 (5.74 Mb) |
|---|---|---|
| LZCS+Huff | 0.8967% | 0.8463% |
| LZCS+ppmdi | 0.1716% | 0.1787% |
| LZCS+ppmz | 0.1482% | 0.1366% |
| gzip | 2.1040% | 4.4331% |
| compress | 10.300% | 10.396% |
| bzip2 | 0.9528% | 0.8430% |
| ppmdi | 2.2115% | 2.1375% |
| ppmz | 0.1669% | 0.1132% |

Figure 11: Comparison between LZCS and classical compressors.

This is followed by *gzip* and *ppmdi* (with significant differences among them depending on the collection), and then by LZCS+Huff and *bzip2*. These have similar compression ratio, although there are again significant differences depending on the collection. Recall, however, that LZCS+Huff is the only method in the group permitting random access and navigation in the collection. Finally, the best compression ratios are achieved by LZCS+ppmdi, LZCS+ppmz and *ppmz*, which are very close. LZCS+ppmdi usually loses to the others and *ppmz* usually loses to LZCS+ppmz. Moreover, *ppmz* is so slow that it cannot be applied except in small collections. These results show that taking advantage of the structure yields significant gains in compression.

## 5.3 Comparison against Structure-Aware Methods

We now compare LZCS against other structure-aware methods: (1)*XMill* v.0.8 (`http://sourceforge.net/projects/xmill`), (2)*XMLPPM* v.0.98.2 (`http://sourceforge.net/projects/xmlppm`), (3) *SCMHuff* (`http://www.infor.uva.es/~jadiego`), and (4)*SCMPPM* (same page).

*XGrind*, (`http://cvs.sourceforge.net/viewcvs.py/xmill/xmill/XGrind`) was excluded from this comparison because we could not make it work properly on our dataset. To be sure that this exclusion was not important, we altered our collection (in a statistically insignificant way) until producing 1 Mb of text where *XGrind* finally worked. The resulting compression ratio was 32.63%, which is not competitive at all in this experiment. *XCQ* was also excluded because we could not find the code, yet results reported in [LWL03] indicate that the compression ratios achieved are similar to those of *XMill*, which we show to be not competitive in our experiments either. The same happens with *Exalt*, according to the results in [Tom04].

Compression ratios are shown in Figure 12. We used default settings for all (yet, letting them use much more memory did not affect the results).

*SCMHuff* is, apart from LZCS+Huff, the only method permitting navigation and random access. *SCMHuff* compression, however, is not competitive, being only slightly superior to *Word Huffman*. We omitted the results of *SCMHuff* for XForms 1, 2, and 3 for readability, where its compression ratio was within 7%-12%. *SCMPPM* is within bounds but still not competitive in most cases.

With few exceptions, LZCS+Huff is significantly better than *XMill* and *SCMPPM* in all sufficiently large collections, producing compressed texts from just 5% smaller to as much as 25 times smaller than *XMill*. *XMLPPM*, on the other hand, obtains clearly better compression than LZCS+Huff in most cases, except for the notable exception of XForms type 1, where all the LZCS family is by far unbeaten. However, *XMLPPM* uses adaptive compression, and hence it is not suitable for navigation or random access on the compressed text.

If we consider the LZCS variants that do not permit navigation and random access, then LZCS+ppmdi and LZCS+ppmz come into play, beating by far all other competitors.

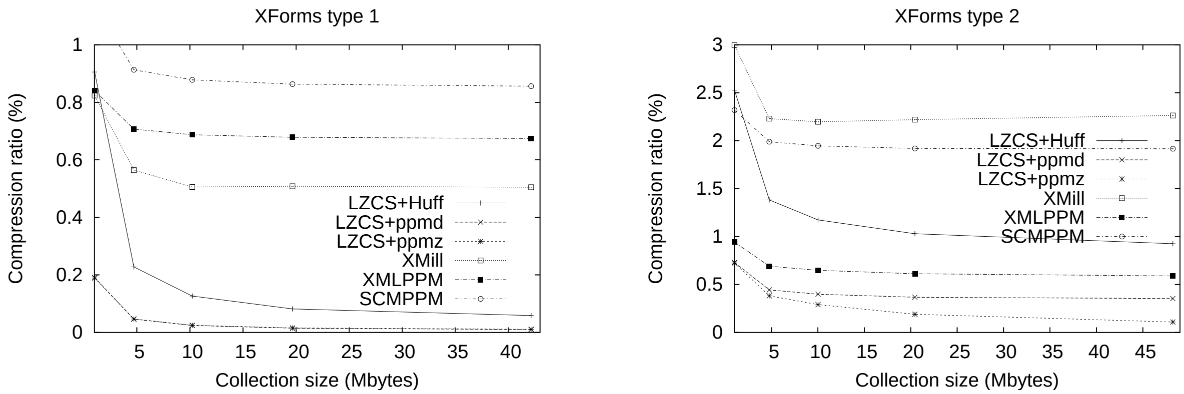
We note the interesting fact that, since it produces structured documents, LZCS can in principle be composed with structure-aware methods, such as SCMPPM, instead of plain text compressors. We have tried some combinations, but the results were no better than those already obtained with the basic PPM compresors.

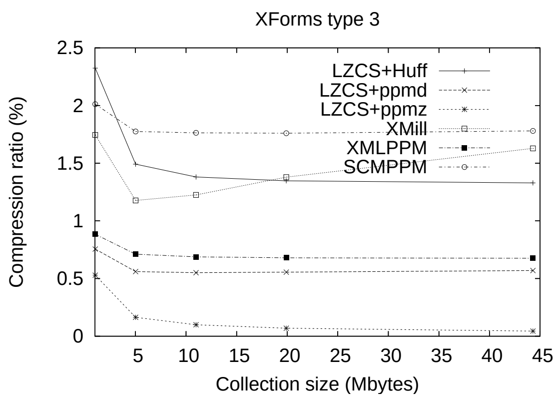
### 5.4 Compression and Decompression Performance

Figure 13 shows compression and decompression speed for all the softwares involved. The times we show are averaged over all the collections, as variations were small among these. For the reasons explained, *ppmz* speed is measured only over the first 5 Mb of the larger collections. The tests were carried out on the SuSE Linux 9.1 operating system, running on a computer with a Pentium IV processor at 1.2 GHz and 384 Mb of RAM.

The fastest at compression/decompression are *gzip* and *XMill* (both based on LZ77), followed by *compress* (based on LZ78). This is expected as this family of compressors is fast, especially at decompression. Shortly after in decompression performance is the LZCS family (also based on Lempel-Ziv), except LZCS+ppmz for obvious reasons. Compression is much slower with the LZCS family, yet not slower than *bzip2*, for example. All other compressors are several times slower to decompress. Other fast options to compress are *ppmdi* and *XMLPPM*.

At compression time, LZCS is not very fast because it has to parse the structure and use the linear time, yet complex, compression algorithm we have explained in Section 4. However, we have managed to make it competitive against start-of-the art compressors. At decompression, LZCS is much faster, benefiting from its Lempel-Ziv nature. Yet, to allow navigability, recursive decompression is necessary, and this slows it down compared to other Lempel-Ziv methods. When combined with other compressors, their overhead must be added to that of LZCS. Yet, this is not as significant as it could be because the other compressors act over the much smaller LZCS transformed




| Method / Size | T.4 (7.19 Mb) | T.5 (5.74 Mb) |
|---|---|---|
| LZCS+Huff | 0.8967% | 0.8463% |
| LZCS+ppmdi | 0.1716% | 0.1787% |
| LZCS+ppmz | 0.1482% | 0.1366% |
| XMill | 0.9426% | 0.9242% |
| XMLPPM | 0.7124% | 0.5530% |
| SCMHuff | 10.971% | 12.055% |
| SCMPPM | 1.3655% | 1.3525% |

Figure 12: Comparison between LZCS and other structure-aware methods.

text.

We note that none of the compressors that significantly outperform LZCS in time get even close to it in compression ratios achieved. Observe also that compression ratios of LZCS stabilize after processing 10–20 Mb of text, so we can process texts in chunks of that size without significantly affecting compression ratio. In practice, the amount of memory we need to compress is 35–45 times the size of the compressed text (which is 1–3 times the size of the original text). In our collections, we need about 25 Mb of main memory to obtain the same compression performance we have shown, by means of partitioning the text. Even when this is rather reasonable, we note that our implementation is not optimized in this aspect, which could be significantly improved.

# 6 Conclusions and Future Work

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 most classical and structure-aware methods; (2) easy random access, visualization and navigation

| Program | Compression | Decompression |
|---|---|---|
| LZCS | 0.385 | 30.262 |
| LZCS+Huff | 0.376 | 21.634 |
| LZCS+ppmdi | 0.387 | 19.200 |
| LZCS+ppmz | 0.154 | 0.779 |
| Word Huffman | 0.388 | 5.438 |
| gzip | 17.858 | 112.212 |
| compress | 4.400 | 43.368 |
| bzip2 | 0.351 | 3.746 |
| ppmdi | 5.073 | 4.990 |
| ppmz | 0.0002 | 0.0002 |
| XMill | 12.751 | 103.038 |
| XMLPPM | 4.943 | 3.855 |
| SCMHuff | 0.187 | 4.169 |
| SCMPPM | 0.964 | 1.310 |

Figure 13: Compression and decompression speeds, in megabytes per second.

of compressed collections; (3) fast and one-pass compression and decompression. Only PPM-based methods compressed better than LZCS in our experiments, but random access to a particular document is impossible with PPM, since it is adaptive and needs to decompress first all the documents that precede the desired one. This is adequate for archival purposes but unsuitable for use in a compressed text database scenario. On the other hand, if we combine LZCS with PPM compression we obtain the best compression ratio among all the PPM-related compressors.

One of the most challenging problems faced was the efficiency problem of the LZCS compression stage, which is quadratic if implemented naively. We overcomed this problem by designing a linear average-time compression algorithm, by using an ad-hoc hashing scheme. The algorithm turns out to be competitive in practice.

We have considered compression of static collections in this paper. In many scenarios, new documents are added to the document collection, but these are never deleted or modified. This is the case, for example, when XML forms are used to keep track of all the transactions made by a company along time (even modifications to previous transactions are expressed by means of a compensating transaction, but the past cannot be changed). LZCS can easily cope with insertion of new documents, as it is a matter of resuming the compression at the point it was left when processing of the previous collection finished. It is a tradeoff decision how much of the data in the hash tables can be maintained to improve compression of future additions to the collection, but this does not affect correctness.

In other cases, for example descriptions of stock, documents may also be updated and deleted. More research is needed in order to accommodate such operations in a text collection compressed with LZCS. The main problem is, of course, that the documents we wish to delete could be referenced elsewhere. One possibility is to maintain a reference count per structure indicating how many references point to it, so the structure can be physically deleted when this counter becomes zero.

An update would consist of inserting the new value and changing the old one by a *forward* pointer to the new one, so that the old one could be deleted or not depending on its reference count. Periodical removal of unused text areas and remapping of pointers would be necessary to avoid the presence of too many gaps due to eliminated documents. Several other alternatives are possible.

The most important future work is to permit searching the compressed structured text. We have seen that the existence of words and phrases in the compressed document can be easily established as their first occurrence cannot appear in compressed form. Yet, this is the most elementary search problem.

A more challenging problem is to answer structural queries, for example XPath queries, on the LZCS compressed collection. One can use the navigation approach to essentially ignore that the text has repeated substructures, and apply any sequential XPath search algorithm. Yet, much more interesting is being able of reusing the results of the search over repeated substructures to avoid working on them again. The final goal is to search in time proportional to the size of the compressed text, not the original text, as would be the case if we ignored the compression. Some approaches to this problem are briefly presented in [LWL03].

Another interesting problem is indexed searching. On very large collections, sequential searching is unacceptable. Index data structures largely improve the sequential search time, at a cost in extra space. For example, a sort of inverted index storing positions of words and structural elements has shown to be useful to solve combined textual and structural queries [NBY97, BYN02]. Although we could, again, build the indexes over the uncompressed text, it would be much better to design indexes that reduce their size when the text is compressible, so that we exploit repetitions in the text to factor out the corresponding repetitions in the indexes.

### Acknowledgment

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

## References

[AdlFN04] J. Adiego, P. de la Fuente, and G. Navarro. Merging prediction by partial matching with structural contexts model. In *Proc. 14th IEEE Data Compression Conference (DCC'04)*, page 522, 2004.

[ANdlF03] J. Adiego, G. Navarro, and P. de la 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 2857, pages 153–167. Springer, 2003.

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

[BSTW86] J. Bentley, D. Sleator, R. Tarjan, and V. Wei. A locally adaptive data compression scheme. *Communications of the ACM*, 29:320–330, 1986.

[BW94] M. Burrows and D. Wheeler. A block sorting lossless data compression algorithm. Technical Report 124, Digital Equipment Corporation, 1994.

[BYN02] R. Baeza-Yates and G. Navarro. XQL and proximal nodes. *J. of the American Society of Information Systems and Technology (JASIST)*, 53(6):504–514, 2002.

[Che01] J. Cheney. Compressing XML with multiplexed hierarchical PPM models. In *Proc. 11th IEEE Data Compression Conference (DCC'01)*, pages 163–172, 2001.

[CW84] J. Cleary and I. Witten. Data compression using adaptive coding and partial string matching. *IEEE Trans. on Communication*, 32:396–402, 1984.

[DPS99] J. Dvorský, J. Pokorný, and V. Snásel. Word-based compression methods and indexing for text retrieval systems. In *Proc. 2nd East European Symp. on Advances in Databases and Information Systems (ADBIS'99)*, LNCS 1691, pages 75–84. Springer, 1999.

[GS00] M. Girardot and N. Sundaresan. Millau: An encoding format for efficient representation and exchange of XML documents over the WWW. In *Proc. 9th Intl. World Wide Web Conf. on Computer Networks*, pages 747–765, 2000.

[Huf52] D.A. Huffman. A method for the construction of minimum-redundancy codes. *Proc. Inst. Radio Engineers*, 40(9):1098–1101, 1952.

[LS00] H. Liefke and D. Suciu. XMill: an efficient compressor for XML data. In *Proc. Intl. ACM Conf. on Management of Data (SIGMOD'00)*, pages 153–164, 2000.

[LW02] M. Levene and P. Wood. XML structure compression. In *Proc. 2nd Intl. Workshop on Web Dynamics*, 2002.

[LWL03] W. Lam, P. Wood, and M. Levene. XCQ: XML compression and querying system. In *Proc. 12th Intl. Conf. on the World Wide Web (WWW'03)*, 2003. Poster.

[Mof89] A. Moffat. Word-based text compression. *Software - Practice and Experience*, 19(2):185–198, 1989.

[MT02] A. Moffat and A. Turpin. *Compression and Coding Algorithms*. Kluwer Academic Publishers, 2002.

[MW01] 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'01)*, pages 162–174. IEEE CS Press, 2001.

[NBY97] G. Navarro and R. Baeza-Yates. Proximal nodes: A model to query document databases by content and structure. *ACM Trans. on Information Systems*, 15(4):400–435, 1997.

[NMN+00] 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.

[Riv92] R. Rivest. The MD5 message-digest algorithm. RFC 1321. MIT Laboratory for Computer Science and RSA Data Security, Inc., April 1992.

[Tar01] J. Tarhio. On compression of parse trees. In *Proc. 8th Intl. Symp. on String Processing and Information Retrieval (SPIRE'01)*, pages 205–211. IEEE Computer Society, 2001.

[TH02] P. Tolani and J. Haritsa. XGRIND: A query-friendly XML compressor. In *Proc. 18th Intl. Conf. of Data Engineering (ICDE'02)*, pages 225–234, 2002.

[Tom04] V. Toman. Syntactical compression of XML data. Presented at *16th Intl. Conf. on Advanced Information Systems Engineering (CAiSE'04)*, Riga, Latvia, June 7–11, 2004.

[Wel84] T. Welch. A technique for high-performance data compression. *IEEE Computer*, 17(6):8–19, 1984.

[WMB99] I. Witten, A. Moffat, and T. Bell. *Managing Gigabytes*. Morgan Kaufmann Publishers, second edition, 1999.

[ZL77] J. Ziv and A. Lempel. An universal algorithm for sequential data compression. *IEEE Trans. on Information Theory*, 23(3):337–343, 1977.

[ZL78] J. Ziv and A. Lempel. Compression of individual sequences via variable-rate coding. *IEEE Trans. on Information Theory*, 24(5):530–536, 1978.

[ZMNBY00] 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.