The Efficient Storage of Text Documents in Digital Libraries

Przemysław Skibiński and
Jakub Swacha

In this paper we investigate the possibility of improving the efficiency of data compression, and thus reducing storage requirements, for seven widely used text document formats. We propose an open-source text compression software library, featuring an advanced word-substitution scheme with static and semidynamic word dictionaries. The empirical results show an average storage space reduction as high as 78 percent compared to uncompressed documents, and as high as 30 percent compared to documents compressed with the free compression software gzip.

It is hard to expect the continuing rapid growth of global information volume not to affect digital libraries.[1] The growth of stored information volume means growth in storage requirements, which poses a problem in both technological and economic terms. Fortunately, the digital librarys' hunger for resources can be tamed with data compression.[2]

The primary motivation for our research was to limit the data storage requirements of the student thesis electronic archive in the Institute of Information Technology in Management at the University of Szczecin. The current regulations state that every thesis should be submitted in both printed and electronic form. The latter facilitates automated processing of the documents for purposes such as plagiarism detection or statistical language analysis. Considering the introduction of the three-cycle higher education system (bachelor/master/doctorate), there are several hundred theses added to the archive every year.

Although students are asked to submit Microsoft Word–compatible documents such as DOC, DOCX, and RTF, other popular formats such as TeX script (TEX), HTML, PS, and PDF are also accepted, both in the case of the main thesis document, containing the thesis and any appendixes that were included in the printed version, and the additional appendixes, comprising materials that were left out of the printed version (such as detailed data tables, the full source code of programs, program manuals, etc.). Some of the appendixes may be multimedia, in formats such as PNG, JPEG, or MPEG.[3] Notice that this paper deals with text-document compression only. Although the size of individual text documents is often significantly smaller than the size of individual multimedia objects, their collective volume is large enough to make the compression effort worthwhile. The reason for focusing on text-document compression is that most multimedia formats have efficient compression schemes embedded, whereas text document formats usually either are uncompressed or use schemes with efficiency far worse than the current state of the art in text compression.

Although the student thesis electronic archive was our motivation, we propose a solution that can be applied to any digital library containing text documents. As the recent survey by Kahl and Williams revealed, 57.5 percent of the examined 1,117 digital library projects consisted of text content, so there are numerous libraries that could benefit form implementation of the proposed scheme.[4]

In this paper, we describe a state-of-the-art approach to text-document compression and present an open-source software library implementing the scheme that can be freely used in digital library projects.

In the case of text documents, improvement in compression effectiveness may be obtained in two ways: with or without regard to their format. The more nontextual content in a document (e.g., formatting instructions, structure description, or embedded images), the more it requires format-specific processing to improve its compression ratio. This is because most document formats have their own ways of describing their formatting, structure, and nontextual inclusions (plain text files have no inclusions).

For this reason, we have developed a compound scheme that consists of several subschemes that can be turned on and off or run with different parameters. The most suitable solution for a given document format can be obtained by merely choosing the right schemes and adequate parameter values. Experimentally, we have found the optimal subscheme combinations for the following formats used in digital libraries: plain text, TEX, RTF, text annotated with XML, HTML, as well as the device-independent rendering formats PS and PDF.[5]

First we discuss related work in text compression, then describe the basis of the proposed scheme and how it should be adapted for particular document formats. The section "Using the scheme in a digital library project" discusses how to use the free software library that implements the scheme. Then we cover the results of experiments involving the proposed scheme and a corpus of test files in each of the tested formats.

Text compression

There are two basic principles of general-purpose data compression. The first one works on the level of character sequences, the second one works on the level of

Przemysław Skibiński (inikep@ii.uni.wroc.pl) is Associate Professor, Institute of Computer Science, University of Wrocław, Poland. **Jakub Swacha** (jakubs@uoo.univ.szczecin.pl) is Associate Professor, Institute of Information Technology in Management, University of Szczecin, Poland.

individual characters. In the first case, the idea is to look for matching character sequences in the past buffer of the file being compressed and replace such sequences with shorter code words; this principle underlies the algorithms derived from the concepts of Arbraham Lempel and Jacob Ziv (LZ-type).[6]

In the second case, the idea is to gather frequency statistics for characters in the file being compressed and then assign shorter code words for frequent characters and longer ones for rare characters (this is exactly how Huffman coding works—what arithmetic coding assigns are value ranges rather than individual code words).[7]

As the characters form words, and words form phrases, there is high correlation between subsequent characters. To produce shorter code words, a compression algorithm either has to observe the context (understood as several preceding characters) in which the character appeared and maintain separate frequency models for different contexts, or has to first decorrelate the characters (by sorting them according to their contexts) and then use an adaptive frequency model when compressing the output (as the characters' dependence on context becomes dependence on position). Whereas the former solution is the foundation of Prediction by Partial Match (PPM) algorithms, Burrows-Wheeler Transform (BWT) compression algorithms are based on the latter.[8]

Witten et al., in their seminal work *Managing Gigabytes*, emphasize the role of data compression in text storage and retrieval systems, stating three requirements for the compression process: good compression, fast decoding, and feasibility of decoding individual documents with minimum overhead.[9] The choice of compression algorithm should depend on what is more important for a specific application: better compression or faster decoding.

An early work of Jon Louis Bentley and others showed that a significant improvement in text compression can be achieved by treating a text document as a stream of space-delimited words rather than individual characters.[10] This technique can be combined with any general-purpose compression method in two ways: by redesigning character-based algorithms as word-based ones or by implementing a two-stage scheme whose first step is a transform replacing words with dictionary indices and whose second step is passing the transformed text through any general-purpose compressor.[11] From the designer's point of view, although the first approach provides more control over how the text is modeled, the second approach is much easier to implement and upgrade to future general-purpose compressors.[12] Notice that the separation of the word-replacement stage from the compression stage does not imply that two distinct programs have to be used—if only an appropriate general-purpose compression software library is available, a single utility can use it to compress the output of the transform it first performed.

An important element of every word-based scheme is the dictionary of words that lists character sequences that should be treated as single entities. The dictionary can be dynamic (i.e., constructed on-line during the compression of every document),[13] static (i.e., constructed off-line before the compression stage and once for every document of a given class—typically, the language of the document determines its class),[14] or semidynamic (i.e., constructed off-line before compression stage but individually for every document).[15] Semidynamic dictionaries must be stored along with the compressed document. Dynamic dictionaries are reconstructed during decompression (which makes the decoding slower than in the other cases). When the static dictionary is used, it must be distributed with the decoder; since a single dictionary is used to compress multiple files, it usually attains the best compression ratios, but it is only effective with documents of the class it was originally prepared for.

The basic compression scheme

The basis of our approach is a word-based, lossless text compression scheme, dubbed Compression for Textual Digital Libraries (CTDL). The scheme consists of up to four stages:

1. document decompression
2. dictionary composition
3. text transform
4. compression

Stages 1–2 are optional. The first is for retrieving textual content from files compressed poorly with general-purpose methods. It is only executed for compressed input documents. It uses an embedded decompressor for files compressed using the Deflate algorithm,[16] but an external tool—Precomp—is used to decode natively compressed PDF documents.[17]

The second stage is for constructing the dictionary of the most frequent words in the processed document. Doing so is a good idea when the compressed documents have no common set of words. If there are many documents in the same language, a common dictionary fares better—it usually does not pay off to store an individual dictionary with each file because they all contain similar lists of words. For this reason we have developed two variants of the scheme. The basic CTDL includes stage 2; therefore it can use a document-specific semidynamic dictionary in the third stage. The CTDL+ variant uses a static dictionary common for all files in the same language; therefore it can omit stage 2.

During stage 2, all the potential dictionary items that meet the word requirements are extracted from the document and then sorted according to their frequency to form a dictionary. The requirements define the minimum length and frequency of a word in the document (by default, 2 and 6 respectively) as well as its content. Only the following kinds of strings are accepted into the dictionary:

- a sequence of lowercase and uppercase letters ("a"–"z", "A"–"Z") and characters with ASCII code values from range 128–255 (thus it supports any typical 8-bit text encoding and also UTF-8)
- URL address prefixes of the form "http://domain/," where domain is any combination of letters, digits, dots, and dashes
- e-mails—patterns of the form "login@domain," where login and domain are any combination of letters, digits, dots, and dashes
- runs of spaces

Stage 3 begins with parsing the text into tokens. The tokens are defined by their content; as four types of content are distinguished, there are also four classes of tokens: words, numbers, special tokens, and characters. Every token is then encoded in a way that depends on the class it belongs to.

The words are those character sequences that are listed in the dictionary. Every word is replaced with its dictionary index, which is then encoded using symbols that are rare or nonexistent in the input document. Indexes are encoded with code words that are between one and four bytes long, with lower indexes (denoting more frequent words) being assigned shorter code words.

The numbers are sequences of decimal digits, which are encoded with a dense binary code, and, similarly to letters, placed in a separate location in the output file.

The special tokens can be decimal fractions, IP numerical addresses, dates, times, and numerical ranges. As they have a strict format and differ only in numerical values, they are encoded as sequences of numbers.[18]

Finally, the characters are the tokens that do not belong to any of the aforementioned group. They are simply copied to the output file, with the exception of those rare characters that were used to construct code words; they are copied as well, but have to be preceded with a special escape symbol.

The specialized transform variants (see the next section) distinguish three additional classes from the character class: letters (words not in the dictionary), single white spaces, and multiple white spaces.

Stage 4 could use any general-purpose compression method to encode the output of stage 3. For this role, we have investigated several open-licensed, general-purpose compression algorithms that differ in speed and efficiency. As we believe that document access speed is important to textual digital libraries, we have decided to focus on LZ–type algorithms because they offer the best decompression times. CTDL has two embedded back-end compressors: the standard Deflate and LZMA, well-known for its ability to attain high compression ratios.[19]

Adapting the transform for individual text document formats

The text document formats have individual characteristics; therefore the compression ratio can be improved by adapting the transform for a particular format. As we noted in the introduction, we propose a set of subschemes (modifications of the original processing steps or additional processing steps) that can help compression—provided the issue that a given subscheme addresses is valid for the document format being compressed. There are two groups of subschemes: the first consists of solutions that can be applied to more than one document format. It includes

- changing the minimum word frequency threshold (the "MinFr" column in table 1) that a word must pass to be included in the semidynamic dictionary (notice that no word can be added to a static dictionary);
- using spaceless word model ("WdSpc" column in table 1) in which a single space between two words is not encoded at all; instead, a flag is used to mark two neighboring words that are not separated by a space;
- run-length encoding of multiple spaces ("SpRuns" column in table 1);
- letter containers ("LetCnt" column in table 1), that is, removing sequences of letters (belonging to words that are not included in the dictionary) to a separate location in the output file (and leaving a flag at their original position).

Table 1 shows the assignment of the mentioned subschemes to document formats, with "+" denoting that a given subscheme should be applied when processing a given document format. Notice that we use different subschemes for the same format depending on whether a semidynamic (CTDL) or static (CTDL+) dictionary is used.

The remaining subschemes are applied for only one document format. They attain an improvement in compression performance by changing the definition of acceptable dictionary words, and, in one case (PS), by changing the definition of number strings.

The encoder for the simplest of the examined formats—plain text files—performs no additional format-specific processing.

The first such modification is in the TEX encoder. The difference is that words beginning with "\" (TEX instructions) are now accepted in the dictionary.

The modification for PDF documents is similar. In this case, bracketed words (PDF entities)—for example "(abc)"—are acceptable as dictionary entries. Notice that PDF files are internally compressed by default—the transform can be applied after decompressing them into textual format. The Precomp tool is used for this purpose.

The subscheme for PS files features two modifications: Its dictionary accepts words beginning with "/" and "\" or ending with "(", and its number tokens can contain not only decimal but also hexadecimal digits (though a single number must have at least one decimal digit). The hexadecimal number must be at least 6 digits long, and is encoded with a flag: a byte containing its length (numbers with more than 261 digits are split into parts) and a sequence of bytes, each containing two digits from the number (if the number of digits is odd, the last byte contains only one digit).

For RTF documents, the dictionary accepts the "\"-preceded words, like the TEX files. Moreover, the hexadecimal numbers are encoded in the same way as in the PS subscheme so that RTF documents containing images can be significantly reduced in size.

Specialization for XML is roughly the transform described in our earlier article, "Revisiting Dictionary-Based Compression."[20] It allows for XML start tags and entities to be added to dictionary, and it replaces every end tag respecting the XML well-formedness rule (i.e., closing the element opened most recently) with a single flag. It also uses a single flag to denote XML attribute value begin and end marks.

HTML documents are handled similarly. The only difference is that the tags that, according to the HTML 4.01 specification, are not expected to be followed by an end-tag (BASE, LINK, XBASEHREF, BR, META, HR, IMG, AREA, INPUT, EMBED, PARAM and COL) are ignored by the mechanism replacing closing tags (so that it can guess the correct closing tag even after the singular tags were encountered).[21]

Table 1. Universal transform optimizations

		CTDL Settings			CTDL+ Settings		
Format	**MinFr**	**WdSpc**	**SpRuns**	**LetCnt**	**WdSpc**	**SpRuns**	**LetCnt**
HTML	3	+	+	+	+	+	-
PDF	3	-	-	-	-	-	-
PS	6	-	+	-	-	+	-
RTF	3	+	-	+	+	-	-
TEX	3	+	+	+	+	+	+
TXT	6	+	+	+	+	+	+
XML	3	+	+	+	+	+	-

Using the scheme in a digital library project

Many textual digital libraries seriously lack text compression capabilities, and popular digital library systems, such as Greenstone, have no embedded efficient text compression.[22] Therefore we have decided to develop CTDL as an open-source software library. The library is free to use and can be downloaded from www.ii.uni.wroc.pl/~inikep/research/CTDL/CTDL09.zip.

The library does not require any additional nonstandard libraries. It has both the text transform and back-end compressors embedded. However, compressing PDF documents requires them to be decompressed first with the free Precomp tool.

The compression routines are wrapped in a code selecting the best algorithm depending on the chosen compression mode and the input document format. The interface of the library consists of only two functions: CTDL_encode and CTDL_decode, for, respectively, compressing and decompressing documents.

CTDL_encode takes the following parameters:

- char* filename—name of the input (uncompressed) document
- char* filename_out—name of the output (compressed) document
- EFileType ftype—format of the input document, defined as:
  enum EFileType { HTML, PDF, PS, RTF, TEX, TXT, XML};
- EDictionaryType dtype—dictionary type, defined as:
  enum EDictionaryType { Static, SemiDynamic };

CTDL_decode takes the following parameters:

- char* filename—name of the input (compressed) document
- char* filename_out—name of the output (decompressed) document

The library was written in the C++ programming language, but a compiled static library is also distributed; thus it can be used in any language that can link such libraries. Currently, the library is compatible with two platforms: Microsoft Windows and Linux.

To use static dictionaries, the respective dictionary file must be available. The library is supplied with an English dictionary trained on a 3 GB text corpus from Project Gutenberg.[23] Seven other dictionaries—German, Spanish, Finnish, French, Italian, Polish, and Russian—can be freely downloaded from www.ii.uni.wroc.pl/~inikep/research/dicts. There also is a tool that helps create a new dictionary from any given corpus of documents, available from Skibiński upon request via e-mail (inikep@ii.uni.wroc.pl).

The library can be used to reduce the storage requirements or also to reduce the time of delivering a requested document to the library user. In the first case, the decompression must be done on the server side. In the second case, it must be done on the client side, which is possible because stand-alone decompressors are available for Microsoft Windows and Linux. Obviously, a library can support both options by providing the user with a choice whether a document should be delivered compressed or not. If documents are to be decompressed client-side, the basic CTDL, using a semidynamic dictionary, seems handier, since it does not require the user to obtain the static dictionary that was used to compress the downloaded document. Still, the size of such a dictionary is usually small, so it does not disqualify CTDL+ from this kind of use.

Table 2. Test set documents specification

File Name	Title	Author	TEX Size (bytes)
13601-t	Expositions of Holy Scripture: Romans Corinthians	Maclaren	1,443,056
16514-t	A Little Cook Book for a Little Girl	Benton	220,480
1noam10t	North America, V. 1	Trollope	804,813
2ws2610	Hamlet	Shakespeare	194,527
alice30	Alice in Wonderland	Carroll	165,844
cdscs10t	Some Christmas Stories	Dickens	127,684
grimm10t	Fairy Tales	Grimm	535,842
pandp12t	Pride and Prejudice	Austen	727,415

Experimental results

We tested CTDL experimentally on a benchmark set of text documents. The purpose of the tests was to compare the storage requirements of different document formats in compressed and uncompressed form.

In selecting the test files we wanted to achieve the following goals:

- test all the formats listed in table 1 (therefore we decided to choose documents that produced no errors during document format conversion)
- obtain verifiable results (therefore we decided to use documents that can be easily obtained from the Internet)
- measure the actual compression improvement from applying the proposed scheme (apart from the RTF format, the scheme is neutral to the images embedded in documents; therefore we decided to use documents that have no embedded images)

For these reasons, we used the following procedure for selecting documents to the test set. First, we searched the Project Gutenberg library for TEX documents, as this format can most reliably be transformed into the other formats. From the fifty-one retrieved documents, we removed all those containing images as well as those that the htlatex tool failed to convert to HTML. In the eleven remaining documents, there were four Jane Austen books; this overrepresentation was handled by removing three of them. The resulting eight documents are given in table 2.

From the TEX files we generated HTML, PDF, and PS documents. Then we used Word 2007 to transform HTML documents into RTF, DOC, and XML (thus this is the Microsoft Word XML format, not the Project Gutenberg XML format). The TXT files were downloaded from Project Gutenberg.

The tests were conducted on a low-end AMD Sempron 3000+ 1.80 GHz system with 512 MB RAM and a Seagate 80 GB ATA drive, running Windows XP SP2.

For comparison purposes, we used three general-purpose compression programs:

- gzip implementing Deflate
- bzip2 implementing a BWT-based compression algorithm

Table 3. Compression efficiency and times for the TXT documents

		Deflate			LZMA			
File Name	gzip	CTDL	CTDL+	7-zip	CTDL	CTDL+	bzip2	PPMVC
13601-t	2.944	2.244	2.101	2.337	2.057	1.919	2.158	1.863
16514-t	2.566	2.150	1.969	2.228	1.993	1.838	2.010	1.780
1noam10t	2.967	2.337	2.109	2.432	2.151	1.958	2.160	1.946
2ws2610	3.217	2.874	2.459	2.871	2.659	2.312	2.565	2.343
alice30	2.906	2.533	2.184	2.585	2.360	2.056	2.341	2.090
cdscs10t	3.222	2.898	2.298	2.928	2.721	2.192	2.694	2.436
grimm10t	2.832	2.275	2.090	2.357	2.079	1.931	2.112	1.886
pandp12t	2.901	2.251	2.097	2.366	2.061	1.930	2.032	1.835
Average	2.944	2.445	2.163	2.513	2.260	2.017	2.259	2.022
Comp. Time	0.688	1.234	0.954	6.688	2.640	2.281	2.110	3.281
Dec. Time	0.125	0.454	0.546	0.343	0.610	0.656	0.703	3.453

Table 4. Compression efficiency and times for the TEX documents

		Deflate			LZMA			
File Name	gzip	CTDL	CTDL+	7-zip	CTDL	CTDL+	bzip2	PPMVC
13601-t	2.927	2.233	2.092	2.328	2.049	1.913	2.146	1.852
16514-t	2.277	1.904	1.794	1.957	1.744	1.645	1.746	1.534
1noam10t	2.976	2.370	2.142	2.445	2.186	1.986	2.195	1.976
2ws2610	3.206	2.906	2.482	2.864	2.674	2.323	2.562	2.340
alice30	2.897	2.526	2.183	2.573	2.350	2.048	2.332	2.085
cdscs10t	3.224	2.931	2.328	2.941	2.759	2.222	2.723	2.466
grimm10t	2.831	2.304	2.120	2.364	2.113	1.960	2.143	1.910
pandp12t	2.881	2.239	2.090	2.346	2.049	1.916	2.013	1.817
Average	2.902	2.427	2.154	2.477	2.241	2.002	2.233	1.998
Comp. Time	0.688	1.250	0.969	6.718	2.703	2.406	2.140	3.329
Dec. Time	0.109	0.453	0.547	0.360	0.609	0.672	0.703	3.485

- PPMVC implementing a PPM-derived compression algorithm[24]

Tables 3–10 show

- the bitrate attained on each test file by the Deflate-based gzip in default mode, the proposed compression scheme in the semidynamic and static variants with Deflate as the back-end compression algorithm, 7-zip in LZMA mode, the proposed compression scheme in the semidynamic and static variants with LZMA as the back-end compression algorithm, bzip2 and PPMVC;
- the average bitrate attained on the whole test corpus; and
- the total compression and decompression times (in seconds) for the whole test corpus, measured on the test platform (they are total elapsed times including program initialization and disk operations).

Bitrates are given in output bits per character of an uncompressed document in a given format, so a smaller

Table 5. Compression efficiency and times for the XML documents

	Deflate			LZMA				
File Name	gzip	CTDL	CTDL+	7-zip	CTDL	CTDL+	bzip2	PPMVC
13601-t	2.046	1.551	1.514	1.585	1.405	1.339	1.451	1.242
16514-t	0.871	0.698	0.670	0.703	0.612	0.590	0.599	0.552
1noam10t	2.383	1.870	1.736	1.914	1.711	1.575	1.724	1.515
2ws2610	0.691	0.539	0.497	0.561	0.474	0.440	0.461	0.422
alice30	1.477	1.258	1.140	1.248	1.131	1.034	1.116	0.999
cdscs10t	2.106	1.892	1.576	1.862	1.741	1.462	1.721	1.538
grimm10t	1.878	1.485	1.422	1.521	1.337	1.276	1.337	1.198
pandp12t	1.875	1.404	1.349	1.465	1.263	1.207	1.252	1.105
Average	1.666	1.337	1.238	1.357	1.209	1.115	1.208	1.071
Comp. Time	0.750	1.844	1.390	10.79	4.891	5.828	7.047	3.688
Dec. Time	0.141	0.672	0.750	0.421	0.859	0.953	1.140	3.907

bitrate (of, e.g., RTF documents compared to the plain text) does not mean the file is smaller, only that the compression was better. Uncompressed files have a bitrate of 8 bits per character.

Looking at the results obtained for TXT documents (table 3), we can see an average improvement of 17 percent for CTDL and 27 percent for CTDL+ compared to the baseline Deflate implementation. Compared to the baseline LZMA implementation, the improvement is 10 percent for CTDL and 20 percent for CTDL+. Also, CTDL+ combined with LZMA compresses TXT documents 31 percent better than gzip, 11 percent better than bzip2, and slightly better than the state-of-the-art PPMVC implementation.

In case of TEX documents (table 4), the gzip results were improved, on average, by 16 percent using CTDL and by 26 percent using CTDL+; the numbers for LZMA are 10 percent for CTDL and 19 percent for CTDL+. In a cross-method comparison, CTDL+ with LZMA beats gzip by 31 percent, bzip2 by 10 percent, and attains results very close to PPMVC.

On average, Deflate-based CTDL compressed XML documents 20 percent better than the baseline algorithm (table 5), and with CTDL+ the improvement rises to 26 percent. CTDL improves LZMA compression by 11 percent, and CTDL+ improves it by 18 percent. CTDL+ with LZMA beats gzip by 33 percent, bzip2 by 8 percent, and loses only 4 percent to PPMVC.

Similar results were obtained for HTML documents (table 6): they were compressed with CTDL and Deflate 18 percent better than with the Deflate algorithm alone, and 27 percent better with CTDL+. LZMA compression efficiency is improved by 11 percent with CTDL and 20 percent with CTDL+. CTDL+ with LZMA beats gzip by 33 percent, bzip2 by 9 percent, and loses only 2 percent to PPMVC.

For RTF documents (table 7), the gzip results were improved, on average, by 18 percent using CTDL, and 25 percent using CTDL+; the numbers for LZMA are respectively 9 percent for CTDL and 17 percent for CTDL+. In a cross-method comparison, CTDL+ with LZMA beats gzip by 34 percent, bzip2 by 7 percent, and loses 5 percent to PPMVC.

Although there is no mode designed especially for DOC documents in CTDL (table 8), the basic TXT mode was used, as it was found experimentally to be the best choice available. The results show it managed to improve Deflate-based compression by 9 percent using CTDL, and by 21 percent using CTDL+, whereas LZMA-based compression was improved respectively by 4 percent for CTDL and 14 percent for CTDL+. Combined with LZMA, CTDL+ compresses DOC documents 30 percent better than gzip, 13 percent better than bzip2, and 1 percent better than PPMVC.

In case of PS documents (table 9), the gzip results were improved, on average, by 5 percent using CTDL, and by 8 percent using CTDL+; the numbers for LZMA improved 3 percent for CTDL and 5 percent for CTDL+. In a cross-method comparison, CTDL+ with LZMA beats gzip by 8 percent, losing 5 percent to bzip2 and 7 percent to PPMVC.

Finally, CTDL improved Deflate-based compression of PDF documents (table 10) by 9 percent using CTDL and 10 percent using CTDL+ (compared to gzip; the numbers are much higher if compared to the embedded PDF compression—see "native" column in table 10); the numbers for LZMA are respectively 7 percent for CTDL and 10 percent for CTDL+. Combined with LZMA, CTDL+ compresses PDF documents 28 percent better than gzip, 4 percent better than bzip2, and 5 percent worse than PPMVC.

The results presented in tables 3–10 show that CTDL manages to improve compression efficiency of the general-purpose algorithms it is based on. The scale of improvement varies between document types, but for most of them it is more than 20 percent for CTDL+ and 10 percent for CTDL. The smallest improvement is achieved in case of PS (about 5 percent). Figure 1 shows the same results in another perspective: the bars show how much better compression ratios were obtained for the same documents using different compression schemes compared to gzip with default options (0 percent means no improvement).

Compared to gzip, CTDL offers a significantly better compression ratio at the expense of longer processing time. The relative difference is especially high in case of decompression. However, in absolute terms, even in the worst case of PDF, the average delay between CTDL+ and gzip is below 180 ms for compression and 90 ms for decompression per file. Taking into consideration the low-end specification of the test computer, these results

Table 6. Compression efficiency and times for the HTML documents

	Deflate			LZMA				
File Name	**gzip**	**CTDL**	**CTDL+**	**7-zip**	**CTDL**	**CTDL+**	**bzip2**	**PPMVC**
13601-t	2.696	2.054	1.940	2.121	1.868	1.751	1.932	1.670
16514-t	1.726	1.405	1.310	1.436	1.258	1.180	1.257	1.113
1noam10t	2.768	2.159	1.972	2.244	1.979	1.815	1.973	1.785
2ws2610	2.084	1.747	1.504	1.743	1.525	1.344	1.499	1.303
alice30	2.451	2.124	1.829	2.128	1.929	1.701	1.888	1.684
cdscs10t	2.880	2.593	2.084	2.597	2.410	1.966	2.348	2.131
grimm10t	2.603	2.074	1.916	2.138	1.883	1.752	1.889	1.688
pandp12t	2.640	2.037	1.891	2.120	1.826	1.717	1.777	1.596
Average	2.481	2.024	1.806	2.066	1.835	1.653	1.820	1.621
Comp. Time	0.750	1.438	1.078	8.203	3.421	3.328	2.672	3.500
Dec. Time	0.140	0.515	0.594	0.359	0.688	0.750	0.812	3.672

Table 7. Compression efficiency and times for the RTF documents

	Deflate			LZMA				
File Name	**gzip**	**CTDL**	**CTDL+**	**7-zip**	**CTDL**	**CTDL+**	**bzip2**	**PPMVC**
13601-t	1.882	1.431	1.372	1.428	1.267	1.200	1.300	1.120
16514-t	0.834	0.701	0.696	0.662	0.601	0.591	0.568	0.529
1noam10t	2.244	1.774	1.637	1.765	1.594	1.462	1.601	1.404
2ws2610	0.784	0.630	0.581	0.629	0.545	0.500	0.520	0.485
alice30	1.382	1.196	1.065	1.134	1.046	0.948	0.995	0.922
cdscs10t	2.059	1.882	1.558	1.784	1.704	1.432	1.645	1.488
grimm10t	1.618	1.301	1.227	1.285	1.150	1.082	1.149	1.010
pandp12t	1.742	1.340	1.264	1.336	1.169	1.115	1.142	1.012
Average	1.568	1.282	1.175	1.253	1.135	1.041	1.115	0.996
Comp. Time	0.766	2.047	1.500	12.62	6.500	7.562	8.032	3.922
Dec. Time	0.156	0.688	0.766	0.469	0.875	0.953	1.312	4.157

certainly seem good enough for practical applications.

Compared to LZMA, CTDL offers better compression and a shorter compression time at the expense of longer decompression time. Notice that the absolute gain in compression time is several times the loss in decompression time, and the decompression time remains short, noticeably shorter than bzip2's and several times shorter than PPMVC's. CTDL+ beats bzip2 (with the sole exception of PS documents) in terms of compression ratio and achieves results that are mostly very close to the resource-hungry PPMVC.

Conclusions

In this paper we addressed the problem of compressing text documents. Although individual text documents rarely exceed several megabytes in size, their entire collections can have very large storage space requirements.

Although text documents are often compressed with general-purpose methods such as Deflate, much better compression can be obtained with a scheme specialized for text, and even better if the scheme is additionally specialized for individual document formats. We have developed such a scheme (CTDL), beginning with a text transform designed earlier for XML documents and

Table 8. Compression efficiency and times for the DOC documents

	Deflate			LZMA				
File Name	gzip	CTDL	CTDL+	7-zip	CTDL	CTDL+	bzip2	PPMVC
13601-t	2.798	2.183	2.062	2.181	1.976	1.854	2.115	1.818
16514-t	2.226	2.213	2.073	1.712	1.712	1.652	1.919	1.686
1noam10t	2.851	2.250	2.025	2.289	2.057	1.869	2.113	1.870
2ws2610	2.497	2.499	2.210	2.095	2.095	1.890	2.251	1.999
alice30	2.744	2.714	2.270	2.345	2.345	2.038	2.348	2.058
cdscs10t	2.916	2.891	2.231	2.559	2.560	2.062	2.475	2.196
grimm10t	2.691	2.677	2.059	2.179	2.179	1.856	2.075	1.833
pandp12t	2.761	2.171	2.050	2.189	1.955	1.843	1.983	1.770
Average	2.686	2.450	2.123	2.194	2.110	1.883	2.160	1.904
Comp. Time	0.718	1.312	1.031	7.078	4.063	3.001	2.250	3.421
Dec. Time	0.125	0.375	0.547	0.344	0.547	0.718	0.735	3.625

Table 9. Compression efficiency and times for the PS documents

	Deflate			LZMA				
File Name	gzip	CTDL	CTDL+	7-zip	CTDL	CTDL+	bzip2	PPMVC
13601-t	2.847	2.634	2.589	2.213	2.105	2.074	2.011	1.778
16514-t	3.226	3.129	3.039	2.730	2.707	2.699	2.613	2.505
1noam10t	2.718	2.551	2.490	2.147	2.060	2.015	1.892	1.694
2ws2610	3.064	2.922	2.795	2.600	2.521	2.450	2.336	2.186
alice30	3.224	3.154	3.026	2.750	2.745	2.691	2.553	2.400
cdscs10t	3.110	3.029	2.890	2.657	2.683	2.579	2.447	2.276
grimm10t	2.833	2.664	2.597	2.288	2.200	2.162	2.074	1.863
pandp12t	2.814	2.533	2.468	2.193	2.049	1.998	1.858	1.644
Average	2.980	2.827	2.737	2.447	2.384	2.334	2.223	2.043
Comp. Time	1.328	3.015	2.500	14.23	10.96	11.09	4.171	5.765
Dec. Time	0.203	0.688	0.781	0.609	1.063	1.125	1.360	6.063

modifying it for the requirements of each of the investigated document formats. It has two operation modes: basic CTDL and CTDL+ (the latter uses a common word dictionary for improved compression) and uses two back-end compression algorithms: Deflate and LZMA (differing in compression speed and efficiency).

The improvement in compression efficiency, which can be observed in the experimental results, amounts to a significant reduction of data storage requirements, giving the reasons to use the library in both new and existing digital library projects instead of general-purpose compression programs. To facilitate this process, we implemented the scheme as an open-source software library under the same name, freely available at http://www.ii.uni.wroc.pl/~inikep/research/CTDL/CTDL09.zip.

Although the scheme and the library are now complete, we plan future extensions aiming both to increase the level of specializations for currently handled document formats and to extend the list of handled document formats.

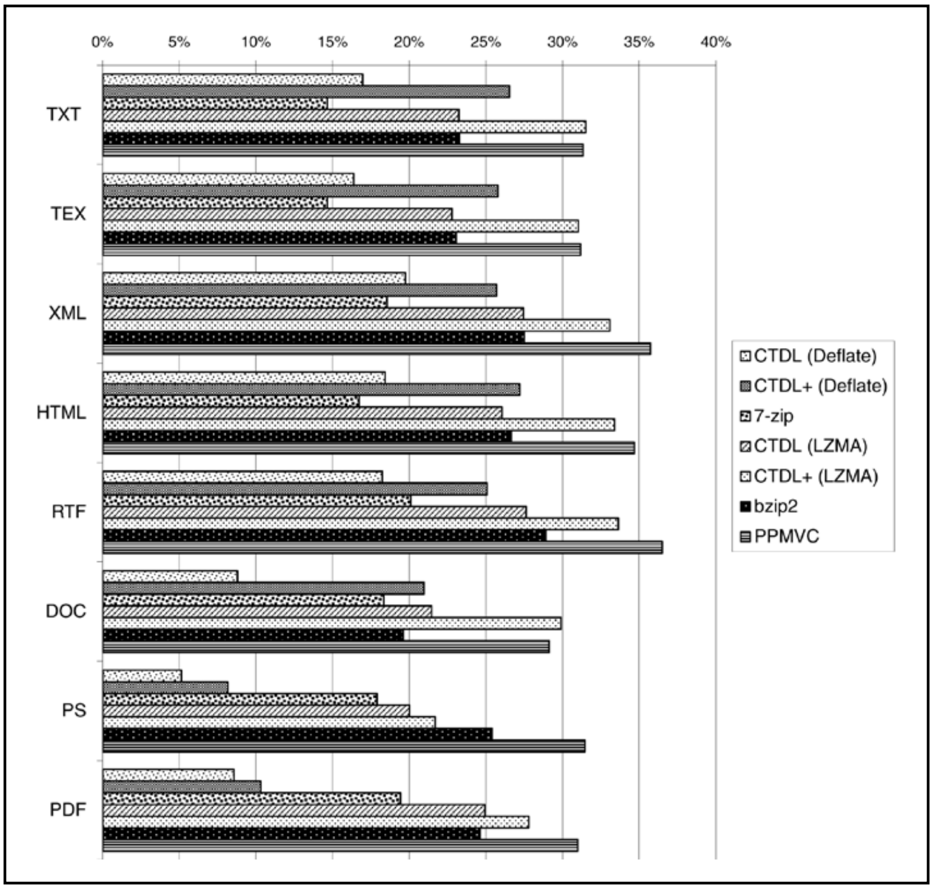

Figure 1. Compression improvement relative to gzip

Table 10. Compression efficiency and times for the (uncompressed) PDF documents

		Deflate				LZMA			
File Name	**native**	**gzip**	**CTDL**	**CTDL+**	**7-zip**	**CTDL**	**CTDL+**	**bzip2**	**PPMVC**
13601-t	3.443	2.624	2.191	2.200	1.986	1.708	1.656	1.852	1.659
16514-t	4.370	2.839	2.836	2.810	2.422	2.422	2.328	2.378	2.241
1noam10t	3.379	2.522	2.103	2.094	1.924	1.659	1.603	1.770	1.587
2ws2610	3.519	2.204	2.346	2.248	1.781	1.947	1.860	1.625	1.480
alice30	3.886	2.863	2.753	2.668	2.429	2.308	2.216	2.315	2.137
cdscs10t	3.684	2.835	2.688	2.557	2.399	2.276	2.164	2.260	2.079
grimm10t	3.543	2.557	2.135	2.120	2.008	1.713	1.661	1.858	1.696
pandp12t	3.552	2.684	2.267	2.256	2.071	1.831	1.769	1.870	1.705
Average	3.672	2.641	2.415	2.369	2.128	1.983	1.907	1.991	1.823
Comp. Time	n/a	1.594	3.672	3.250	19.62	13.31	16.32	5.641	7.375
Dec. Time	n/a	0.219	0.844	0.969	0.719	1.219	1.360	1.765	7.859

Acknowledgements

Szymon Grabowski is the coauthor of the XML-WRT transform, which served as the basis for the CTDL library.

References

1. John F. Gantz et al., *The Diverse and Exploding Digital Universe: An Updated Forecast of Worldwide Information Growth Through 2011* (Framingham, Mass.: IDC, 2008), http://www.emc.com/collateral/analyst-reports/diverse-exploding-digital-universe.pdf (accessed May 7, 2009).

2. Timothy C. Bell, Alistair Moffat, and Ian H. Witten, "Compressing the Digital Library," in *Proceedings of Digital Libraries '94* (College Station: Texas A&M Univ. 1994): 41.

3. Ian H. Witten and David Bainbridge, *How to Build a Digital Library* (San Francisco: Morgan Kaufmann, 2002).

4. Chad M. Kahl and Sarah C. Williams, "Accessing Digital Libraries: A Study of ARL Members' Digital Projects," *The Journal of Academic Librarianship* 32, no. 4 (2006): 364.

5. Donald E. Knuth, *TeX: The Program* (Reading, Mass.: Addison-Wesley, 1986); Microsoft Technical Support, *Rich Text Format (RTF) Version 1.5 Specification,* 1997, http://www.biblioscape.com/rtf15_spec.htm (accessed May 7, 2009); Tim Bray et al., eds., *Extensible Markup Language (XML) 1.0 (Fourth Edition)*, 2006, http://www.w3.org/TR/2006/REC-xml-20060816 (accessed May 7, 2009); Dave Raggett, Arnaud Le Hors, and Ian Jacobs, eds., *W3C HTML 4.01 Specification*, 1999, http://www.w3.org/TR/REC-html40/ (accessed May 7, 2009); *PostScript Language Reference*, 3rd ed. (Reading, Mass.: Addison-Wesley, 1999), http://www.adobe.com/devnet/postscript/pdfs/PLRM.pdf (accessed May 7, 2009); *PDF Reference*, 6th ed., version 1.7, 2006, http://www.adobe.com/devnet/acrobat/pdfs/pdf_reference_1-7.pdf (accessed May 7, 2009).

6. Jacob Ziv and Abraham Lempel, "A Universal Algorithm for Sequential Data Compression," *IEEE Transactions on Information Theory* 23, no. 3 (1977): 337.

7. Ian H. Witten, Alistair Moffat, and Timothy C. Bell, *Managing Gigabytes: Compressing and Indexing Documents and Images*, 2nd ed. (San Francisco: Morgan Kaufmann, 1999).

8. John G. Cleary and Ian H. Witten, **"**Data Compression using Adaptive Coding and Partial String Matching,**"** *IEEE Transactions on Communication* 32, no. 4, (1984): 396; Michael Burrows and David J. Wheeler, "A Block-Sorting Lossless Data Compression Algorithm," Digital Equipment Corporation SRC Research Report 124, 1994, www.hpl.hp.com/techreports/Compaq-DEC/SRC-RR-124.pdf (accessed May 7, 2009).

9. Witten, Moffat, and Bell, *Managing Gigabytes.*

10. Jon Louis Bentley et al., "A Locally Adaptive Data Compression Scheme," *Communications of the ACM* 29, no. 4 (1986): 320; R. Nigel Horspool and Gordon V. Cormack, "Constructing Word-Based Text Compression Algorithms," *Proceedings of the Data Compression Conference* (Snowbird, Utah, 1992): 62.

11. See for example Andrei V. Kadach, "Text and Hypertext Compression," *Programming & Computer Software* 23, no. 4 (1997): 212; Alistair Moffat, "Word-based text compression," *Software—Practice & Experience* 2, no. 19 (1989): 185; Przemysław Skibiński, Szymon Grabowski, and Sebastian Deorowicz, "Revisiting Dictionary-Based Compression," *Software—Practice & Experience* 35, no. 15 (2005): 1455.

12. Przemysław Skibiński, Jakub Swacha, and Szymon Grabowski, "A Highly Efficient XML Compression Scheme for the Web," *Proceedings of the 34th International Conference on Current Trends in Theory and Practice of Computer Science*, *LNCS* 4910 (2008): 766.

13. Jon Louis Bentley et al., "A Locally Adaptive Data Compression Scheme," *Communications of the ACM* 29, no. 4 (1986): 320.

14. Skibiński, Grabowski, and Deorowicz, "Revisiting Dictionary-Based Compression," 1455.

15. Skibiński, Swacha, and Grabowski, "A Highly Efficient XML Compression Scheme for the Web," 766.

16. Peter Deutsch, "DEFLATE Compressed Data Format Specification version 1.3," RFC1951, Network Working Group, 1996, www.ietf.org/rfc/rfc1951.txt (accessed May 7, 2009).

17. Christian Schneider, Precomp—A Command Line Precompressor, 2009, http://schnaader.info/precomp.html (accessed May 7, 2009).

18. The technical details of the algorithm constructing code words and assigning them to indexes, and encoding numbers and special tokens, are given in Skibiński, Swacha, and Grabowski, "A Highly Efficient XML Compression Scheme for the Web," 766.

19. David Solomon, *Data Compression: The Complete Reference,* 4th ed. *(*London: Springer-Verlag, 2006).

20. Skibiński, Swacha, and Grabowski, "A Highly Efficient XML Compression Scheme for the Web," 766.

21. Dave Raggett, Arnaud Le Hors, and Ian Jacobs, eds., *W3C HTML 4.01 Specification*, 1999, http://www.w3.org/TR/REC-html40/ (accessed May 7, 2009).

22. Ian H. Witten, David Bainbridge, and Stefan Boddie, "Greenstone: Open Source DL Software," *Communications of the ACM* 44, no. 5 (2001): 47.

23. Project Gutenberg, 2008, http://www.gutenberg.org/ (accessed May 7, 2009).

24. Przemysław Skibiński and Szymon Grabowski, "Variable-Length Contexts for PPM," *Proceedings of the IEEE Data Compression Conference* (Snowbird, Utah, 2004): 409.

Index to Advertisers

ALCTS	cover 2
LITA	cover 3, cover 4