Improving HTML Compression

Przemysław Skibiński
University of Wrocław, Institute of Computer Science,
Joliot-Curie 15, 50-383 Wrocław, Poland
E-mail: inikep@ii.uni.wroc.pl

Keywords: hypertext markup language, HTML compression, HTML transform

Received: March 6, 2008

The verbosity of the Hypertext Markup Language (HTML) remains one of its main weaknesses. This problem can be solved with the aid of HTML specialized compression algorithms. In this work, we describe a lossless HTML transform which, combined with generally used compression algorithms, allows to attain high compression ratios. Its core is a fully reversible transform featuring substitution of words in an HTML document using a static English dictionary or a semi-static dictionary of the most frequent words in the document, effective encoding of dictionary indexes, numbers, and specific patterns.

The experimental results show that the proposed transform improves the HTML compression efficiency of general purpose compressors on average by 15% in case of gzip achieving comparable processing speed. Moreover, we show that the compression ratio of gzip can be improved by up to 28% for the price of higher memory requirements and much slower processing.

Povzetek: Opisan je izvirni algoritem za izboljšavo zgoščevanja HTML.

1 Introduction[1]

Since the origin of World Wide Web, the Hypertext Markup Language (HTML) is a standard for Internet web pages. HTML has many advantages. One of its main advantages is that it is a textual format, what means that HTML is human-readable and can be edited by any text editor. The textual format of HTML is also one of its main disadvantages as it introduces verbosity. Nevertheless verbosity can be coped with by applying data compression.

Currently HTML files are usually compressed with common LZ77-based compression algorithms [23] like gzip or deflate [7]. LZ77-based algorithms can be substituted with more powerful, but slower and much more memory-demanding BWT-based [3] and PPM-based [5] algorithms. All these algorithms, however, are general-purpose and much better results can be achieved with a compression algorithm specialized for dealing with HTML documents.

In recent years there is slow progress in general-purpose compression thus many researchers and practitioners have directed towards specialized compression. Specialized compression algorithms can be divided into two groups: completely novel algorithms and preprocessors coupled with general-purpose algorithms. A good example for the first group is DNA compression, where many specialized algorithms exist. The second group of algorithms transforms the input data and then passes the transform's output to a general-purpose compressor. The transform removes redundancies and correlations not exploited by a general-purpose algorithm, what makes output data more compressible.

In our previous work we have presented algorithms specialized for text [18] and XML compression [19]. The biggest gain from our algorithms was achieved by creating dictionary of frequent words and replacing the words with shorter codewords. We have observed over 27% improvement with an LZ77-based algorithm on English texts using a fixed English dictionary. Using a fixed dictionary for XML documents was problematic because of the hardness to select a proper set of words, relevant across a wide range of real-world XML documents. Therefore we achieved the best results (over 35% improvement with an LZ77-based algorithm) with a semi-dynamic dictionary obtained in a preliminary pass over the input data. About 20% of this improvement was achieved by effective encoding of numbers, dates and times found in XML documents.

HTML format is similar to XML as both are SGML-based. On the other side, XML is mainly used as a standard for storage and interchange of structured information and HTML is mainly used to publish text-based information. In this usage HTML is more similar to texts.

The primary objective of our research was to design an efficient way of compressing HTML documents, which will reduce Internet traffic or will reduce storage requirements of HTML data. In this work we will present our two specialized algorithms for HTML compression. One uses a fixed English dictionary and the second one uses a semi-dynamic dictionary obtained in a preliminary

[1] This is an extended version of the poster: Skibiński, P.: *Improving HTML Compression*. Proceedings of the IEEE Data Compression Conference, Snowbird, UT, USA, (2008), pp. 545.

pass over the input data. Both algorithms are designed in four variants for proper general-purpose algorithms.

The map of the paper is as follows. Section 2 contains a short review of HTML format and existing HTML compression methods thus putting our work in a proper context. We also describe related word-based and XML compressors. In Section 3 we describe step-by-step our HTML transform, its main ideas, and its most significant details. The next section presents back-end compression algorithms used with our transform and details about optimizations for these algorithms. Section 5 contains implementation details, description of files used for experiments, and experimental results. Section 6 gives our conclusions and points out several issues for further study.

2 Related work

2.1 HTML description

HTML is a language that describes a structure of text-based information in a document. It denotes certain text as links, headings, paragraphs, and lists. It also supplements text with embedded images and other objects. XHTML is a new, XML-based version of HTML.

An HTML document consists of elements. An HTML element always has a start tag (e.g. `<element-name>`) and may have an end tag (e.g. `</element-name>`) in opposite to XML or XHTML, where the end tag is required. Elements may have two basic properties: attributes, contained in the start tag (e.g. `<element-name attribute="value">`), and content, located between the tags (e.g. `<element-name>Content</element-name>`). If there is no content a start tag and an end tag can be presented in short form that is `<element-name/>`. Comments in HTML are delimited by `<!--` and `-->` sequences. Some HTML elements, for example `<br>`, do not have any content and must not have a closing tag.

The following example [10] contains document title (`<title>` element), heading (`<h1>` element), paragraphs (`<p>` elements) and links (`<a>` elements):

```
<html>
<head><title>About the Test Data</title></head>
<body>
<h1 align="center">About the Test Data</h1>
<p align="center">Matt Mahoney<br>
Last update: Dec. 17, 2006.
<a href="text.html#history">History</a>
<p>The test data for the <a
href="text.html">Large Text Compression
Benchmark</a> is the first 10<sup>9</sup> bytes
of the English Wikipedia dump on Mar. 3, 2006.
```

2.2 HTTP protocol

Hypertext Transfer Protocol (HTTP) is a communications protocol used to transfer information on World Wide Web. HTTP is a request/response protocol between a client and a server. The client is making an HTTP request to the server, which delivers HTML files, images and other.

HTTP compression [13] is the technology used to compress contents from a web server (an HTTP server) and to decompress them in an user's browser. HTTP compression is a recommendation of the HTTP 1.1 protocol specification as it reduces network traffic and improves page download time on slow networks [15]. It is especially useful when size of the web pages is large.

The experiments conducted by Wan [21] showed that HTTP compression can be improved utilizing the previously requested files in a browsing session as a dictionary, but this idea was not embedded in HTTP protocol until today.

The popular LZ77-based gzip was intended to be the HTTP compression algorithm. Currently, HTTP servers and clients supports also LZ77-based deflate format. Lighttpd server supports also BWT-based bzip2 compression, but this format is only supported by lynx and some other console text-browsers. Deflate, gzip, and bzip2, however, are general-purpose compression algorithms and much better results can be achieved with a compression algorithm specialized for dealing with HTML documents.

2.3 Word-based compression

StarNT [20] is a dictionary-based scheme, which replaces natural language words with references to an external dictionary. A word in StarNT dictionary is a sequence of symbols over the alphabet [a..z]. There is no need to use uppercase letters in the dictionary, as there are two one-byte flags (reserved symbols), f_{cl} and f_{uw}, in the output alphabet to indicate that either a given word starts with a capital letter while the following letters are all lowercase, or a given word consists of capitals only. Another introduced flag, f_{or}, prepends an unknown word. Finally, there is yet a collision-handling flag, f_{esc}, used for encoding occurrences of flags f_{cl}, f_{uw}, f_{or}, and f_{esc} in the text.

The ordering of words in the dictionary D, as well as mapping the words to unique codewords, are important for the compression effectiveness. StarNT uses the following rules:

- The most popular words are stored at the beginning of the dictionary. This group has 312 words.
- The remaining words are stored in D according to their increasing lengths. Words of same length are sorted according to their frequency of occurrence in some training corpus.
- Only letters [a..zA..Z] are used to represent the codeword (with the intention to achieve better compression performance with the backend compressor).

Each word in D has assigned a corresponding codeword. Codewords' length varies from one to three bytes. As only the range [a..zA..Z] for codeword bytes is used, there can be up to $52 + 52 \times 52 + 52 \times 52 \times 52 = 143,364$ entries in the dictionary. The first 52 words have codewords: a, b, . . . , z, A, B, . . . , Z. Words from the 53rd to the 2756th have codewords of length 2: aa, ab, …, ZY, ZZ; and so on.

WRT [18] is an English text preprocessor, a successor of StarNT. WRT replaces words in input text file with shorter codewords and uses several other techniques to improve performance of latter compression

The dictionary is sorted according to the frequency of words as more frequent messages should be represented with shorter codes than less frequent messages. WRT English dictionary have 80,000 words. Each word in D has assigned a corresponding codeword. Codewords' length is variable and span from one to four symbols. Ordinary text files, at least English ones, consist solely of ASCII symbols not exceeding 127, so codewords' alphabet has 128 symbols (ASCII values from 128 to 255). If there is a symbol from codewords' alphabet in the input file, then WRT outputs token t_{esc} and this symbol. Codewords' alphabet (128 symbols) is divided into four separate parts. WRT uses the mapping <101, 9, 9, 9> for codewords, and thus there are 101 + 101·9 + 101·9·9 + 101·9·9·9 = 82,820 distinct codewords available. It is enough for 80,000 words WRT dictionary. The codeword bytes are emitted in the reverse order, i.e., the range for the last codeword byte has always 101 values.

WRT uses several additional techniques to improve the compression performance. First is q-gram replacement, which is based on substituting frequent sequences of q consecutive characters, i.e., q-grams, with single symbols. The next technique that improves the compression performance is End-of-Line (EOL) coding. The general idea is to replace EOL symbols with spaces and to encode information enabling the reverse operation in a separate stream. The last technique used by WRT is surrounding words with spaces, which converts all words to be surrounded by space characters. This technique gives gain only if there are at least a few occurrences of the word, because it joins similar contexts in PPM compressor (helps in better prediction of the word's first symbol as well as the next symbol just after the word).

mPPM [1] is a text compressor, which is based on Shkarin's PPMd [16]. mPPM splits text into alternating sequences of words and non-words. Words and non-words use a common dynamic dictionary.

Each item has assigned a codeword, which always consists of two bytes. Therefore the dictionary may include up to 2^{16} items. If the dictionary is bigger least recently used (LRU) words are removed. Two codewords are reserved. The first is the End-Of-File flag and the second signals occerence of a new item.

mPPM uses two separate PPM models. The first encodes only codewords. The second, auxilary model encodes new items with a standard character-based PPM.

HufSyl [9] and LZWL [9] are the first text compressors that use syllables as units, instead of characters or words. Syllables are obtained by one of algorithms of decomposition into syllables. These algorithms use syllable-based compression in combination with respectively, adaptive Huffman and LZW coding.

These methods have their counterpart variants for whole words, which gave better results in our experiments. We decided to include only results of word-based versions in our Table 2.

2.4 XML compression

Cheney's XMLPPM [4] is a streaming compressor which uses a technique named multiplexed hierarchical modeling (MHM). It switches between four models: one for element and attribute names, one for element structure, one for attributes, one for strings, and encodes them in one stream using PPMD+ or, in newer implementations, Shkarin's PPMd [16]. The tag and attribute names are replaced with shorter codes. An important idea in Cheney's algorithm is injecting the previous symbol from another model into the current symbol's context. Injecting means that both the encoder and decoder assume there is such a symbol in the context of the current symbol but do not explicitly encode nor decode it. The idea of symbol injection is to preserve (at least to some degree) contextual dependencies across different structural models.

SCMPPM [2] can be seen as an extreme case of XMLPPM. Instead of using only few models, it maintains a separate model for each element class. Every class contains elements having the same name and the same path from the document root. This technique, called Structure Context Modeling (SCM), wins over XMLPPM on large documents (tens of megabytes), but loses on smaller files. Also, SCMPPM requires lots of memory for maintaining multiple statistical models and under limited memory scenarios it may lose significantly, even compared to pure PPMd.

3 HTML Transform

In this section we present our two algorithms: Semi-Dynamic HTML Transform (SDHT) and Static HTML Transform (SHT). We introduce subsequent parts of our algorithms step by step.

3.1 End tag encoding

In the previous section we have described structure of HTML documents. In a well-formed HTML document, every end tag must match a corresponding start tag. This can hardly be exploited by general-purpose compression algorithms, as they maintain a linear, not stack-alike data model. The compression ratio can then be increased by replacing every matching end tag with merely an element closing flag.

Our transform puts elements on a stack when a start tag has appeared. The last inserted element is removed from a stack when an end tag has appeared. The problem with HTML is that not all elements must have a closing tag. It can be solved by ignoring elements that allow an end tag omission. The second problem with HTML is that some tags (e.g. <p>) should have corresponding end tags, but human editors skip these closing tags. Moreover, web browsers do not report errors on documents of this kind. Therefore our transform allows

non-valid HTML documents. The above-mentioned problems do not occur in XHTML.

3.2 Quotes modeling

Attributes of HTML elements usually contain neighboring *equal* and *quotation mark* characters (e.g. `attribute="value"`). Sometimes attributes are encoded using *equal* and *apostrophe* characters (e.g. `attribute='value'`). We have found that replacing these two characters with a flag improves compression performance. We made the same with *quotation mark* and *angle right bracket* (*greater*) characters that closing start tags with attribute(s) (e.g. `<element-name attribute="value">`).

3.3 Spaces modeling

Layout of an HTML document (e.g., trailing spaces, tabulators, end of line symbols) is not relevant for web browsers, but it may be useful for human editors of a document. This kind of redundancy, typical to HTML documents created with editors caring about the output format, cannot be well exploited by general-purpose compression algorithms.

Our transform makes use of structural indentation by efficiently encoding the leading spaces in lines. For every line our transform counts number of occurrences for leading spaces with length from 1 up to 256 symbols. If number of occurrences for the certain length is higher than a predefined threshold our transform assigns a special codeword for leading spaces of this length.

3.4 Number encoding

Numbers appear very often in HTML documents. We found that storing numbers as text is ineffective. Numbers can be encoded more efficiently using a numerical system with base higher than 10.

In our transform every decimal integer number n is replaced with a single byte whose value is $\lceil \log_{256}(n+1) \rceil + 48$. The actual value of n is then encoded as a base-256 number. A special case is made for sequences of zeroes preceding a number – these are left intact.

Our transform encodes in a special way also other numerical data that represent specific information types. Currently our transform recognizes the following formats:

- dates between 1977-01-01 and 2153-02-26 in YYYY-MM-DD (e.g. “2007-03-31”, Y for year, M for month, D for day) and DD-MMM-YYYY (e.g. “31-MAR-2007”) formats;
- years from 1900 to 2155 (e.g. “1999”, “2008”)
- times in 24-hour (e.g., “22:15”) and 12-hour (e.g., “10:15pm”) formats;
- value ranges (e.g., “115-132”);
- decimal fractional numbers with one (e.g., “1.2”) or two (e.g., “1.22”) digits after decimal point.

Dates are replaced with a flag and encoded as a two bytes long integer whose value is the difference in days from 1977-01-01. To simplify the calculations we assume each month to have 31 days. If the difference with the previous date is smaller than 256, another flag is used and the date is encoded as a single byte whose value is the difference in days from the previous date.

Years are replaced with a sequence of two bytes representing respectively: the year flag and the difference between the actual year and 1900.

Times are replaced with a sequence of three bytes representing respectively: a flag signaling a time pattern (conforming to the presented notation), the hour in 24-hour convention, and minutes.

Value ranges in the format x–y where $x < 65536$ and $0 < y - x < 256$ are encoded in four bytes: one for the range flag, two for the value of x, and one for the difference $y - x$.

Decimal fractional numbers with one digit after decimal point and value from 0.0 to 24.9 are replaced by two bytes: a flag and their value stored as fixed point integer. In case of those with two digits after decimal point, only their suffix, starting from the decimal point, is replaced with two bytes: a flag and the number’s fractional part stored as an integer.

3.5 Semi-dynamic dictionary

The backbone of the proposed transform is to replace the most frequent words with references to a dictionary. A semi-dynamic version of our transform constructs a separate dictionary for every processed document, but, once constructed, the dictionary is not changed during an HTML transform. The transform works in two passes. The dictionary is obtained in a preliminary pass over the data, and contains sequences of length at least $l_{\min}$ characters that appear at least $f_{\min}$ times in the document. The dictionary is sorted by word frequency and stored within the compressed file, thus making the reverse operation faster. Dictionary references are encoded using a byte-oriented prefix code, where the more frequent words have assigned shorter codewords (length varies from one to four bytes). The prefix code and the variables $f_{\min}$ and $l_{\min}$ depend on a back-end compression algorithm described in the next section. We have also tried a fully dynamic (one-pass) transform variant, but it gives much worse compression ratio as the same word can have assigned different codewords.

In the second pass of a semi-dynamic transform, the parsed data items are encoded in a byte-oriented manner (words, spaces and flags with a prefix code; numbers, dates, years, times, value ranges, and decimal fractional numbers with respective coding schemes), and then compressed with a compression algorithm and written to disk. We chose four algorithms of this kind: LZ77-based, LZMA/BWT-based, PPM-based, and PAQ-based, which are described in detail in the following section.

Our notion of a “word” is broader than its common meaning. Namely, semi-dynamic dictionary contains items from the following classes:

- ordinary words – sequences of lowercase and uppercase letters (a-z, A-Z) and 128-255 (which

supports, e.g., all languages with a Latin-based alphabet);
- start tags – sequences of characters that start with <, contain letters, digits, underscores, colons, dashes, or dots, and end with >. Start tags can also include one or more preceding spaces as HTML documents sometimes have regular arrangements of the lines in which individual tags very often begin in the same column, preceded with the same number of spaces,
- URL address prefixes – sequences 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,
- words in form `"&data;"`, where data is any combination of letters, representing HTML entities.

3.6 Matching shorter words

Our transform uses separate output alphabets for original words (not replaced with a reference to a dictionary) and codewords. Therefore it is easy to encode a part of a word, if the prefix matches some word in a dictionary but the whole word does not. Still, the gain we achieve in this way is insubstantial. Our algorithm with all above-mentioned ideas is called Semi-Dynamic HTML Transform (SDHT).

3.7 Static dictionary

Static HTML Transform (SHT) is similar to Semi-Dynamic HTML Transform (SDHT). The main difference is that a semi-dynamic dictionary is replaced with a static dictionary, which is embedded in the compressor and the decompressor.

There are two advantages of a static dictionary over a semi-dynamic dictionary: there is no need to make the first pass over the input data to create the semi-dynamic dictionary and there is no need to store the semi-dynamic dictionary within processed data to make decompression possible.

On the other hand a static dictionary is limited to some class of documents e.g. English language. The dictionary must be spread with the compressor and the decompressor. Moreover, a semi-dynamic dictionary contains words that are actually frequent in the document, not words that could potentially be frequent, as it is in the case of a static dictionary. Nevertheless for HTML documents a static English dictionary usually gives a better compression ratio than a semi-dynamic dictionary.

4 Back-end compression

Succinct word encoding appears to be the most important idea in Static HTML Transform (SHT) and Semi-Dynamic HTML Transform (SDHT). The dictionary references are encoded using symbols which are not existent in the input HTML document. If, however, one of reserved symbols occurs in the document, and is not a part of an encoded word, the coder prepends it with a special escape symbol.

There are four modes of encoding, chosen depending on the attached back-end compression algorithm: LZ77-based [23], LZMA/BWT-based [3], PPM-based [5], and PAQ-based [11]. The encoding scheme, however, is the same for SHT and SDHT. In all cases, dictionary references are encoded using a byte-oriented prefix code, where the length varies from one to four bytes. Although it produces slightly longer output than, for instance, Huffman coding [8], the resulting data can be easily compressed further, which is not the case with the latter. Obviously, more frequent words have assigned shorter codewords.

4.1 LZ77-based compression

The LZ77 algorithm [23] finds duplicated sequences of bytes in the input data. The next occurrences of a sequence are replaced by a pointer to the previous occurrence. The pointer is encoded as a distance to the previous occurrence in a limited past buffer and a match length. Literals are encoded directly. Most LZ77 variants use Huffman coding for literals, match offsets, and match lengths. LZ77-based methods are the most widely-used compression algorithms. They are known for fast compression and very fast decompression, but limited effectiveness.

Gzip is a common LZ77-based compression algorithm. It uses a buffer (*sliding window*) for finding matches that has only 32 KB, which is mostly responsible for both very high compression speed and mediocre compression ratios. When a sequence of bytes does not occur anywhere in the previous 32 KB, it is emitted as a sequence of literal bytes. Match lengths are limited to 258 bytes. We used gzip 1.2.4 with default values in our experiments.

4.2 LZ77 optimized transform

In comparison to modern algorithms LZ77-based compressors are not complicated. For example, they do not predict characters on the basis of their context. The strength of LZ77 lies in succinct encoding of long matching sequences. In consequence, a transform optimized for LZ77 compression should attempt to:
- reduce the number of characters to encode;
- decrease the offset (in bytes) of the matching sequences;
- decrease the length (in bytes) of the matching sequence;
- virtually increase the sliding window, i.e., the past buffer in which matching sequences are looked for.

It appears that in case of LZ77 (but not necessarily LZMA, BWT, PPM, or PAQ), shortening the output of the transform improves the compression ratio. In accordance with this observation, we chose the biggest possible alphabet for codewords: byte values from 128 up to 255 and most values in range 0–31, plus a few more. These symbols are very rarely used in most HTML documents. If, however, one of these symbols occurs in

the document, and is not part of an encoded word, the coder marks it with a special escape symbol.

HTML elements contain usually textual content. Another important idea in LZ77 optimized transform is elimination of most spaces between words. Since usually a word is preceded by a single space, only exceptions from this rule require special treatment: only the positions where spaces should not be inserted are marked with a respective flag. Such an assumption is known as the spaceless word model [12].

Still, without spaces between words, there must be a way to detect codeword boundaries. In a LZ77 optimized transform dictionary references are encoded using a byte-oriented prefix code, where the length varies from one to three bytes. The first byte of the codeword can belong to one of three disjoint ranges:

- C_1 if it is a one-byte long codeword; there are $|C_1|$ such codewords available,
- C_2 if it is a prefix of two-bytes long codeword, followed by a single byte in the full possible value range; there are $|C_2| * 256$ such codewords available,
- C_3 if it is a prefix of three-bytes long codeword, followed by two bytes in the full possible value range; there are $|C_3| * 256 * 256$ such codewords available.

In this way, we obtain $|C_1| + |C_2| * 256 + |C_3| * 256 * 256$ codewords in total. As this is a kind of prefix code, all the codewords are immediately decodeable. The size of ranges C_1, C_2, and C_3 are set according to the size of the document to compress and the resulting dictionary size.

For SDHT a semi-dynamic dictionary contains sequences of length at least $l_{min} = 2$ characters that appear at least $f_{min} = 12$ times in the document. These values gave good results for most files used in the experiments.

4.3 LZMA-based compression

LZMA is a modern compression algorithm based on ideas from a LZ77 compression family. It also finds duplicated sequences of bytes in the input data, but it contains many improvements. Some of the major features of LZMA are sophisticated match parsing, working with large buffers (up to 1 GB), and low order contextual encoding of literals.

LZMA significantly improves compression ratio in comparison to LZ77-based algorithms at the cost of much slower compression (decompression speed is not much affected). LZMA is implemented in the well-known 7-zip [14] compression utility. We used LZMA 4.43 with default 8 MB dictionary in our experiments.

4.4 LZMA and BWT optimized transform

We found experimentally that a transform optimized for LZMA and BWT-based [3] (e.g. bzip2) compression should have similar characteristic and there is no need to create separate versions.

In the LZMA/BWT optimized transform the codeword alphabet consists of fewer symbols than LZ77 optimized transform. It uses only 128 symbols with byte values from 128 up to 255.

In the LZMA/BWT optimized transform dictionary references are encoded using a less dense variant of a byte-oriented prefix code, with non-intersecting ranges for different codeword bytes. We use only two disjoint ranges of bytes, C_1 and C_2, but the codeword lengths still span from 1 to 3 bytes. Any codeword byte from the range C_1 is unambiguously recognized as the suffix byte. In this way, we have $|C_1|$ one-byte codewords, $|C_2| * |C_1|$ two-byte codewords, and $|C_2| * |C_2| * |C_1|$ three-byte codewords. Such a reversed byte order was found to improve a compression ratio.

As well as the LZ77 optimized transform the LZMA/BWT optimized transform uses spaceless word model. For SDHT a semi-dynamic dictionary contains sequences of length at least $l_{min} = 2$ characters that appear at least $f_{min} = 12$ times in the document. These values were found experimentally.

4.5 PPM-based compression

PPM [5] is an adaptive statistical compression method. A statistical model accumulates counts of symbols (usually 1-byte characters) seen so far in a given context. Thanks to that, an encoder can predict probability distribution for new symbols from the input data. The more skewed the probability distribution in contexts, the higher compression will result. Increasing the context length is beneficial for encoding symbols known in a given context, but amplifies the problem of efficient encoding of the symbols yet unseen in a given context (generally speaking, they are handled via an escape to a lower order model, but how to estimate the escape probability is a gross research topic).

HTML data might contain long repeated strings. These data are compressed with most PPM variants in a way far from optimal, as the highest order used by e.g. Shkarin's PPMd [16] is only 16. Skibiński and Grabowski [17] presented the PPMVC algorithm (PPM with variable-length contexts), a variant of PPM* [6] adapted to cooperate with modern PPM mechanisms. PPMVC extends the character-based PPM with string matching similar to the one used by the LZ77 algorithm.

The PPMVC mechanism works on maximum order contexts only; in shorter contexts the current symbol is encoded with an ordinary PPM model (namely, Sharkin's PPMd model was used).

In PPMVC (called PPMVC2 in [17]) each maximum order context holds a pointer to reference context (the previous occurrence of the context) and the minimum left match length. The left match length (LML) is the length of the common part of the active context and the reference context. LML, by definition, is always at least as large as the maximum PPM order. The right match length (RML) is defined as the length of the matching sequence between symbols to encode and symbols followed by the reference context.

When a character is encoded from the maximum order context, the longest LML is evaluated, using the last context's appearance. If it is below the minimal left match length (*minLML*), then the encoder uses ordinary PPM encoding (without emitting any escape symbol). In

the other case, the encoder uses this context to find the RML (zero or more) and encodes it using an additional global RML model.

There are two more ideas in PPMVC that improve the compression effectiveness. First is the minimum right match length (*minRML*). If the current right match length is below the *minRML* threshold, then PPMVC sets RML to 0. This assures that short matches are not used.

The second idea is to encode sequences of length being a multiple of the parameter d. For example, if there is a match of length 14, and d is 3, then only the first 12 characters of the match are encoded (the truncated characters might however be part of the next RML). In this way, matches are somewhat shorter than they could be, but their lengths are cheaper to encode. In the original PPMVC [17], RML was bounded by a constant, while in the current variant the maximum RML is automatically increased if very long matches are encountered.

PPMVC offers compression ratio higher than LZMA, and faster compression time. The PPMVC's drawback is that its decompression time is very close to its compression time, which means it is several times longer than gzip's or LZMA's decompression times. In our experiments we used PPMVC 1.2 with prediction model order 8 and 64 MB of model size.

4.6 PPM optimized transform

In the PPM optimized transform the codeword alphabet consists of the biggest possible alphabet for codewords: byte values from 128 up to 255 and most values in range 0–31, plus a few more.

In the PPM-friendly mode dictionary references are encoded using a prefix code, where the length varies from one to four bytes. The four disjoint ranges are of size $|C_1|$, $|C_2|$, $|C_3|$ and $|C_4|$, respectively. Namely, we have $|C_1|$ one-byte codewords, $|C_2| * |C_1|$ two-byte codewords, $|C_3| * |C_2| * |C_1|$ three-byte codewords, and $|C_4| * |C_3| * |C_2| * |C_1|$ four-byte codewords. The first byte of a codeword unambiguously defines its length. For instance, when encoding a two byte long codeword, a byte from the range of size $|C_2|$ will be followed by a byte from the range of size $|C_1|$. The parameters C_1, C_2, C_3, C_4 are selected according to the size of the created dictionary, with the principle of maximizing the number of short codewords.

The PPM optimized transform does not use spaceless word model. For SDHT a semi-dynamic dictionary contains sequences of length at least $l_{\min} = 2$ characters that appear at least $f_{\min} = 64$ times in the document. These values gave good results for most files used in the experiments.

4.7 PAQ-based compression

PAQ [11] is a family of compressors, originally developed by Matthew Mahoney, based on context modeling. As opposed to most PPM variants, which use a character-based alphabet PAQ works on the bit level. In PPM a new symbol in a context must be encoded in lower orders using an escape mechanism. PAQ does not use the escape symbol at all as in each step it must encode only 0 or 1.

The binary alphabet allows a new character in a context to be distinguished after first unseen bit, what is not possible in the case of PPM. This is the next improvement to the PPM algorithm. In the PAQ's coding stage a binary symbol is encoded with a predicted probability by an arithmetic encoder, like in the PPM algorithm.

Bit level coding in PAQ allows easy introduction of additional predicting models. PAQ8 uses several predicting models e.g., order-n models (n to 16), similar to the one used in PPM; a string matching model, similar to one used the LZ77 algorithm; and a number of text, multimedia, tabular, or binary data oriented models (e.g., for x86 executables or BMP images).

Mixing the prediction of individual models in PAQ8 is performed with several neural networks. The outputs of these networks are combined using a second-level neural network. Before submitted to an arithmetic coder, the outputs go through two stages of adaptive probability maps (APM). The APM mechanism is related to the secondary symbol probability estimation (SSE), known from the PPMII algorithm [16]. It updates the probability considering previous experience and the current context.

The main disadvantage of the PAQ8 algorithm are high memory requirements and low compression speed. It makes this algorithm unattractive from a practical point of view. This is why we prepared FastPAQ, stripped-down version of PAQ8, intended to improve compression and decompression speed. From PAQ8 we have left only the order-n models, and we have also simplified APM stages, in overall making it more similar to PPM. FastPAQ is still much slower than fast PPM variants, but achieves better compression ratios.

In our experiments we used FastPAQ8 with model size 140 MB.

4.8 PAQ optimized transform

In the PAQ optimized transform the codeword alphabet consists of fewer symbols than PPM optimized transform. It uses only 128 symbols with byte values from 128 up to 255.

In the PAQ-friendly mode dictionary references are encoded using the same prefix code as in the PPM optimized transform, where the length varies from one to four bytes. The four disjoint ranges are of size $|C_1|$, $|C_2|$, $|C_3|$ and $|C_4|$, respectively. Namely, we have $|C_1|$ one-byte codewords, $|C_2| * |C_1|$ two-byte codewords, $|C_3| * |C_2| * |C_1|$ three-byte codewords, and $|C_4| * |C_3| * |C_2| * |C_1|$ four-byte codewords. In the PAQ optimized transform, however, the parameters C_1, C_2, C_3, C_4 are fixed and equal 64, 32, 16, and 16, respectively, what makes them better suitable for PAQ's bit-level predictors.

The PAQ optimized transform does not use spaceless word model. For SDHT a semi-dynamic dictionary contains sequences of length at least $l_{\min} = 2$ characters that appear at least $f_{\min} = 64$ times in the document. These values were found experimentally.

5 Experimental results

This section presents implementation details of the SDHT and SHT algorithms. It also contains description of files used for experiments and discussion on experimental results of the SDHT and SHT algorithms with four different back-end compression methods.

5.1 Implementation details

The SDHT and SHT implementation contains a fast and simple HTML parser built as a finite state automaton (FSA), which accepts proper words and numerical (including date and time) expressions. The parser does not build any trees, but treats an input HTML document as one-dimensional data. It has small memory requirements, as it only uses a stack to trace opening and closing tags. The parser supports the HTML 4.01 specification (e.g. allowed an end tag omission for some tags).

The SHT implementation uses a static English dictionary with about 80.000 words. In this dictionary, words are sorted with the relation to their frequency in a training corpus of more than 3 GB English text taken from the Project Gutenberg library. The words are stored in lower case as SHT implements the capital conversion method to convert the capital letter starting a word to its lowercase equivalent and denote the change with a flag. Additionally, SHT uses another flag to mark a conversion of a full uppercase word to its lowercase form.

SHT requires only one pass over the input data while SDHT works in two passes over the input data. In the first pass, a dictionary is formed and the frequency of each of its items is computed. For the semi-dynamic dictionary, we allocate 8 MB of memory. If the dictionary reaches that limit, it is frozen, i.e., the counters of already included words can be incremented but no new word can be added. Still, in practice we rarely get close to the assumed limit (which can also be changed with a program switch). The complete dictionary is stored within the compressed file, so this pass is unnecessary during decompression, making the reverse operation faster. The words selected for the dictionary are written explicitly, with separators, at the beginning of the output file. In the second pass, the actual transform takes place, data are parsed into proper words and numerical expressions and respectively encoded.

The crucial operation in the encoding is dictionary search. In SDHT a search function is called twice for each word in the document: first time during the semi-dynamic dictionary buildup, second time during the actual parsing and word encoding. The choice of a dictionary data structure can seriously affect the overall transform performance. We have decided to use a fixed-size (4 MB) array with chained hashing for search, which we previously tested in our work on a text transform [18]. Its advantages are simplicity, moderate memory usage, and $O(1)$ search time (assuming that a single word is read in constant time).

The reverse SDHT and SHT are simpler. Again we use an FSA, which now recognizes flags and codewords, and transforms them to the original form. Obviously, there is no real search in the dictionary, only lookups in $O(1)$ time per codeword.

Our implementation of SDHT and SHT has embedded four back-end compression algorithms: gzip, LZMA, PPMVC, and FastPAQ8. Of these, gzip is the fastest, but provides the lowest compression ratio. FastPAQ8 is the slowest, but gives the best compression effectiveness.

SDHT and SHT are truly lossless, i.e., they do not ignore the document layout (e.g., trailing spaces) and the decoded file is an exact copy of the encoded one. The transforms can handle any HTML documents with 8-bit (ISO-8859 and UTF-8) or 16-bit (Unicode) encodings. SDHT and SHT was implemented in C++ and compiled with MS Visual C++ 2008.

5.2 HTML corpus

In compression benchmarking, proper selection of documents used in experiments is essential. To the best of our knowledge, there is no publicly available and widely respected HTML corpus to this date. Therefore, we have based our test suite on entire common Internet web sites downloaded (without images, etc.) using WinHTTrack Website Copier. The resulting corpus represents a wide range of real-world HTML documents.

Detailed information for each group of the documents is presented in Table 1; it includes: URL address, number of files and total size of files. The size of a single file spans from 1 up to 296 KB.

Name	*URL address*	*no. files*	*Total size*
Hillman	hillmanwonders.com	781	34421 KB
Informatica	www.informatica.si	12	122 KB
Mahoney	www.cs.fit.edu/~mmahoney/	11	596 KB
MaxComp	maximumcompression.com	61	2557 KB
STL	www.sgi.com/tech/stl/	237	2551 KB
TightVNC	tightvnc.com	21	289 KB
Tortoise	tortoisesvn.net	393	5342 KB
Travel	travelindependent.info	69	3841 KB

Table 1: Basic characteristics for the HTML corpus used in the experiments

5.3 Compression ratio

The primary objective of experiments was to measure the performance of our implementation of the SDHT and SHT algorithms. For comparison purposes, we included in the tests general-purpose compression tools: gzip 1.2.4, LZMA 4.43, PPMVC 1.2, and FastPAQ8,

employing the same algorithms at the final stage of SDHT and SHT, to demonstrate the improvement from applying the HTML transform.

As we are not aware of any specialized algorithms for HTML compression we have compared our algorithms to well-know word-based text compression techniques: StarNT [20], WRT [18], HufSyl [9], LZWL [9], mPPM [1] and StarWE [22]. StarWE is based on WRT and gives almost identical results therefore its results were omitted. We have also tried to use XMLPPM [4] and SCMPPM [2], which work well with XHTML files, but it do not support HTML files. These algorithms are described in details in Section 2.

The first part of Table 2 contains results of word-based text compression algorithms. For each program and group of HTML documents a bitrate is given in output bits per input character, hence the smaller the values, the better. The last but one column includes an average bitrate computed for all the eight groups of documents. The last column presents the average improvement of preprocessors for all documents compared to the general purpose algorithms result.

The next parts of Table 2 contain compression results of the introduced HTML corpus using gzip, LZMA, PPMVC, FastPAQ, and our implementation of the SDHT and SHT algorithms combined with gzip, LZMA, PPMVC, and FastPAQ.

SHT with gzip achieves compression results better than all word-based text compression algorithms, including a PPM-based mPPM. Compared to the general-purpose compression tools, SDHT improves compression of the introduced HTML corpus on average by 4% in case of gzip, 0% for LZMA, almost 2% in case of PPMVC and about 1% for FastPAQ. SDHT is very fast and almost does not influence on compression and decompression speed of general-purpose compression algorithms. Moreover, it speeds up FastPAQ, because preprocessed data is smaller than original.

In the first section we were wondering if HTML is more similar to XML or to texts. Our experiments show that HTML is more similar to texts as Static HTML Transform (SHT) with a fixed English dictionary gives much better results than SDHT. SHT improves compression of the introduced HTML corpus on average by about 15% in case of gzip, 12% for LZMA, almost 8% in case of PPMVC and 10% for FastPAQ. Compression and decompression speed in comparison to SDHT is a little bit lower as there is a need to read a fixed English dictionary. SHT, however, allows to read the dictionary only once and processes all HTML documents in one run.

Concluding, SHT with gzip gives 15% improvement over gzip achieving comparable processing speed. Moreover, SHT with FastPAQ gives the best compression effectiveness, which is 28% better than gzip without any transform.

To ease the comparison, Figure 1 shows size of compressed HTML corpus with all tested transforms and back-end compression algorithms.

6 Conclusions

HTML has many advantages, but its main disadvantage is verbosity, which can be coped with by applying data compression. HTML is usually used in combination with gzip compression, but gzip is a general-purpose compression algorithm and much better results can be achieved with a compression algorithm specialized for dealing with HTML documents.

In this paper we have presented the SDHT and SHT transform aiming to improve lossless HTML compression in combination with existing general purpose compressors. The main components of our algorithms are: a static dictionary or a semi-static dictionary of frequent alphanumerical phrases (not limited to "words" in a conventional sense), and binary encoding of popular patterns, like numbers and dates.

We have developed two versions of our transform: semi-dynamic (SDHT) and static (SHT). Both algorithms have some disadvantages. SDHT does not support streams as input (offline compression) as it requires two passes over an input file. SHT uses a fixed English dictionary required for compression and decompression. It might be the biggest obstacle for SHT to become standard.

Thanks to the SHT transform, however, compression ratio of the introduced HTML corpus was improved by as much as 15% in case of gzip, 12% for LZMA, 8% in case of PPMVC and almost 10% for FastPAQ.

SHT and SDHT have many nice practical properties. The transforms are completely reversible, i.e. the decoded document is an accurate copy of the input document. Moreover, SHT and SDHT are implemented as a stand-alone program, requiring no external compression utility, no HTML parser, thus avoiding any compatibility issues.

There is a way likely to increase the HTML compression further. Layout of an HTML document (e.g., trailing spaces, tabulators, end of line symbols) is not relevant for web browsers and can be transformed to a more compressible form. We expect that a lossy version of the SHT transform could produce a few percent better results for the price of further complication of the transform.

Acknowledgement

The author would like to thank Szymon Grabowski and Jakub Swacha for suggestions of possible improvements.

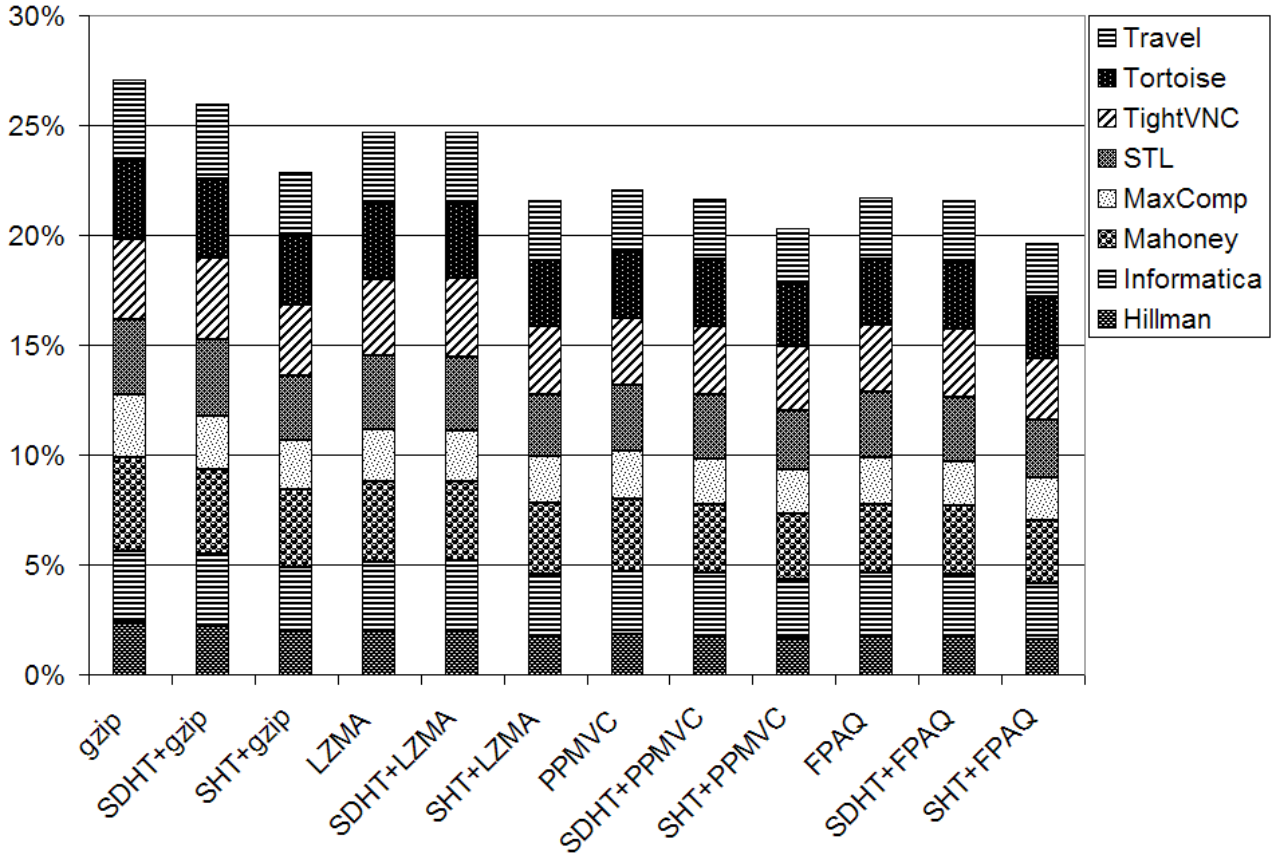

Figure 1: Size of compressed HTML corpus with different back-end compression algorithms

	Hillman	Informatica	Mahoney	MaxComp	STL	TightVNC	Tortoise	Travel	Average	Improvement
HufSyl	2.95	3.53	3.31	3.03	3.48	3.44	3.37	2.88	3.249	
LZWL	2.13	3.18	3.23	2.39	3.22	3.26	3.13	2.72	2.908	
gzip	1.51	2.08	2.72	1.86	2.19	2.34	2.27	2.34	2.164	
StarNT+gzip	1.42	1.94	2.54	1.79	1.97	2.17	2.08	2.06	1.996	7.74%
WRT+gzip	1.44	1.99	2.49	1.80	1.95	2.13	2.06	1.97	1.979	8.55%
mPPM	1.34	2.16	2.30	1.55	2.31	2.24	2.23	1.95	2.010	
gzip	1.51	2.08	2.72	1.86	2.19	2.34	2.27	2.34	2.164	
SDHT+gzip	1.40	2.13	2.45	1.56	2.20	2.39	2.31	2.19	2.079	3.93%
SHT+gzip	1.23	1.88	2.26	1.47	1.85	2.09	2.02	1.84	1.830	15.42%
LZMA	1.29	1.99	2.35	1.53	2.13	2.23	2.17	2.13	1.978	
SDHT+LZMA	1.29	2.04	2.30	1.46	2.16	2.29	2.21	2.07	1.978	0.00%
SHT+LZMA	1.13	1.78	2.08	1.38	1.79	1.99	1.92	1.74	1.726	12.71%
PPMVC	1.19	1.83	2.09	1.41	1.91	1.96	1.93	1.79	1.764	
SDHT+PPMVC	1.15	1.80	2.02	1.33	1.87	1.97	1.94	1.77	1.731	1.84%
SHT+PPMVC	1.06	1.71	1.92	1.30	1.71	1.86	1.83	1.60	1.624	7.94%
FPAQ	1.14	1.81	2.01	1.36	1.90	1.96	1.92	1.79	1.736	
SDHT+FPAQ	1.13	1.80	1.99	1.28	1.89	1.99	1.94	1.79	1.726	0.58%
SHT+FPAQ	1.01	1.65	1.83	1.24	1.67	1.82	1.77	1.56	1.569	9.65%

Table 2: Compression results for HTML datasets in output bits per input character.

References

[1] Adiego, J., and de la Fuente, P.: *Mapping Words into Codewords on PPM*. String Processing and Information Retrieval, SPIRE, (2006), LNCS 4209, pp. 181–192.

[2] Adiego, J., de la Fuente, P., and Navarro, G.: *Using Structural Contexts to Compress Semistructured Text Collections*. Information Processing and Management 43, 3 (May), (2007), pp. 769–790.

[3] Burrows, M., Wheeler, D. J.: *A block-sorting data compression algorithm*. SRC Research Report 124. Digital Equipment Corporation, Palo Alto, CA, USA, (1994).

[4] Cheney, J.: *Compressing XML with multiplexed hierarchical PPM models*. Proceedings of the IEEE Data Compression Conference, Snowbird, UT, USA, (2001), pp. 163–172.

[5] Cleary, J. G., and Witten, I. H.: *Data compression using adaptive coding and partial string matching*. IEEE Trans. on Comm. 32, 4 (April), (1984), pp. 396–402.

[6] Cleary, J. G., Teahan, W. J., and Witten, I. H.: *Unbounded Length Contexts for PPM*. Proceedings of the IEEE Data Compression Conference, Snowbird, UT, USA, (1995), pp. 52–61.

[7] Deutsch, P.: *DEFLATE Compressed Data Format Specification version 1.3*. RFC1951, (1996), http://www.ietf.org/rfc/rfc1951.txt.

[8] Huffman, D. A.: *A Method for the Construction of Minimum-Redundancy Codes*. Proc. IRE 40.9 (Sept.), (1952), pp. 1098-1101.

[9] Lánský, J., Žemlička, M.: *Text Compression: Syllables*. Proceedings of the Dateso 2005 Annual International Workshop on DAtabases, TExts, Specifications and Objects. CEUR-WS, Vol. 129, pp. 32–45.

[10] Mahoney, M.: *About the Test Data*, 2006, http://cs.fit.edu/~mmahoney/compression/textdata.html

[11] Mahoney, M.: *Adaptive Weighing of Context Models for Lossless Data Compression*. Technical Report TR-CS-2005-16, Florida Tech., USA, 2005.

[12] Moura E.S., Navarro G., Ziviani N.: *Indexing Compressed Text*. In Baeza-Yates R, editor, *Proceedings of the 4th South American Workshop on String Processing (WSP'97)*, Valparaiso, Carleton University Press, 1997; 95–111.

[13] Nielsen H.F.: *HTTP Performance Overview*, 2003, http://www.w3.org/Protocols/HTTP/Performance/

[14] Pavlov I.: *7-zip* compression utility. http://www.7-zip.org.

[15] Radhakrishnan S.: *Speed Web delivery with HTTP compression*, 2003, http://www-128.ibm.com/developerworks/web/library/wa-httpcomp/

[16] Shkarin, D.: *PPM: One Step to Practicality*. Proceedings of the IEEE Data Compression Conference, Snowbird, UT, USA, (2002), pp. 202–211.

[17] Skibiński, P., and Grabowski, Sz.: *Variable-length contexts for PPM*. Proceedings of the IEEE Data Compression Conference, Snowbird, UT, USA, (2004), pp. 409–418.

[18] Skibiński, P., Grabowski, Sz., and Deorowicz, S.: *Revisiting dictionary-based compression*. Software – Practice and Experience, 35(15), (2005), pp. 1455–1476.

[19] Skibiński, P., Grabowski, Sz., and Swacha, J.: *Effective asymmetric XML compression*, Software – Practice and Experience, 38 (10), (2008), pp. 1027–1047.

[20] Sun, W., Zhang, N., Mukherjee, A.: *Dictionary-based fast transform for text compression*. Proceedings of international conference on Information Technology: Coding and Computing, ITCC, (2003), pp. 176–182.

[21] Wan, R.: *Browsing and Searching Compressed Documents*. PhD dissertation, University of Melbourne, 2003, http://www.bic.kyoto-u.ac.jp/proteome/rwan/docs/wan_phd_new.pdf

[22] Yang, J., Savari, S.A.: *Dictionary-based English text compression using word endings*. Proceedings of the IEEE Data Compression Conference, Snowbird, UT, USA, (2007), pp. 410.

[23] Ziv, J., and Lempel, A.: *A Universal Algorithm for Sequential Data Compression*. IEEE Trans. Inform. Theory 23, 3 (May), (1977), pp. 337–343.