Jakub Swacha*, Szymon Grabowski**

Self-Extracting Compressed HTML Documents

1. Introduction

Data compression, whether in its lossless or lossy form, is ubiquitous in modern information systems. The aims of data compression are multiple and comprise:

- reducing the storage of data, in order to e.g. fit the capacity of popular media formats, like DVD;
- speeding-up data transmission over a network;
- speeding-up sequential access to data, e.g. via compact representation of sparse matrices;
- speeding-up random access to data, thanks to storing a larger portion of data either in RAM (vs. disk) or in cache memory (vs. RAM);
- searching directly in the compressed form, which can sometimes be faster than searching in raw data (again, thanks to e.g. better caching).

To make those techniques really useful, several conditions should be met. In particular, it is desirable that the used data compression algorithms and formats are:

- transparent for users;
- appropriate for the expected data types (e.g., text compression techniques are rather useless for multimedia data);
- fast enough, especially in the decompression (since reading data is typically performed more often than writing);
- aware of the hardware constraints (CPU power and its cache sizes and hierarchy, network bandwidth, available RAM).

Although the majority of data that users deal with today are multimedia (esp. video), one can observe still large (and perhaps growing) amount of various textual data, especially on the Internet. Those include HTML pages (and other related files, e.g., CSS and

* Institute of Information Technology in Management, University of Szczecin, Poland, jakubs@uoo.univ.szczecin.pl
** Computer Engineering Department, Technical University of Lodz, Poland, sgrabow@kis.p.lodz.pl

JavaScript), XML datasets, mail messages (transmitted and then archived), plain-text newsgroup archives, web server logs and more. Human-readable textual data are easy to analyze (e.g., in order to track bugs in serialized objects or detect suspicious user behavior in a web traffic analyzer or an OS activity log), edit, and extract snippets from. An interesting feature of "texts" of the mentioned kinds, however, is their redundancy, typically much greater than the redundancy of natural language texts, e.g., fiction books with no markup. Redundancy not only increases the costs of data transmission and storage, but can also slow down query handling. Another issue concerning redundant data are increased memory requirements, which may pose trouble in the notoriously multitasking and multi-user systems.

Once we agree that data compression is a natural means to overcome some issues with textual data, we should decide what particular data type we want to compress, what the main purpose of this process is, and which of the aforementioned desirable features of compression we can attain.

In this paper we present a compression algorithm dedicated to HTML pages. While not a new research goal, the novelty of our solution is that the decompression process will be transparent to a user visiting a given page with a standard web browser capable of running JavaScript.

2. Related work

The HTTP/1.1 protocol [2], widely used for transferring data on the World Wide Web, gives possibility to deliver compressed content, i.e., all modern web browsers are capable to decode it automatically (and transparently) at the client side. Two compression formats are provided, Deflate and gzip (the underlying compression algorithms are identical).

No other compression algorithms are widely supported on the WWW. For example, the Lighttpd web server [3] supports the stronger bzip2 compression, but it is handled by few niche browsers (e.g., Lynx) and the market share for the Lighttpd server is below 1% in May 2010, according to the well-known Netcraft survey [4]. We note that bzip2 is a general-purpose compressor, not aware of the nested (X)HTML structure.

Opera Turbo™ is a solution available only for the Opera browsers (Opera Desktop, Opera Mobile and perhaps others) in which the requested web content is cached (on a server), compressed and served upon a further request to other Opera users. It was reported [5] that Opera Turbo compresses web traffic by 63.6% on average (November 2009), but it is hard to find information on the details of the techniques used (at least part of the gain is from lossy compression of images on the web pages). According to [6], in April 2010 there were 5.7 million users of this technology, out of 52 million total users of Opera Desktop.

A specialized, and not handled by web browsers, compression algorithm was presented by Skibiński [7, 8], which is based on earlier XML compression research from his team [9, 10]. The algorithm performs a lossless (or visually lossless, i.e., the exact original file cannot be recovered but its rendered appearance in a browser remains unchanged) prepro-

cessing of the input HTML document and then submits it to a backend coder; in case of the Deflate algorithm, the average improvement compared to pure Deflate is about 14%. The ideas used include denoting closing tags with a flag, dense encoding of numbers, semi-dynamic encoding of frequently used words (which notions comprises also URLs, emails and HTML entities) and removal of excessive blank characters (in the visually lossless variant). The problem with that approach is, of course, the need to manually decompress the HTML document, i.e., this can reduce storage of web archives rather than make faster the transfer of a requested page.

Finally, we note that there are a plethora of HTML "compressors" that minimize the size of the input document via removing comments, unnecessary whitespace characters (spaces, tab characters, EOLs), unnecessary quotation marks, optionally removing META tags etc. Typically the HTML document gets shrunk by at most 10%. Those techniques are visually lossless, but note that e.g. removing some META tags can affect the rank of a given page in web searchers, and eliminating quotation marks is allowed in HTML but not in well-formed XHTML. An example of such application is Absolute HTML Compressor [1].

3. The compression algorithm

The key concept behind our approach was to make the compressed HTML document self-extracting, that is to enable its proper decompression in a web browser without a need to install any client-side software (neither stand-alone nor browser plug-in). In order to accomplish this task, we only require the web browser to be capable of processing JavaScript code and have this feature enabled.

The compressor is a stand-alone program that processes a HTML document, replacing the whole contents of its BODY element with a single SCRIPT element containing JavaScript code that is capable of fully recreating the original content. The actual compressed contents of the document BODY are stored server-side in another file read by the script using an XMLHttpRequest object. Notice that the compressor does not change HEAD element to avoid conflicts with original scripts and make the header easily accessible for robots and search engines.

The actual compression is done by substitution of frequent character sequences with short identifiers. The compression procedure consists of the following steps:

1. Extract the contents of the BODY element from the compressed document.
2. Obtain the set of characters which are unused within the BODY element. These originally unused characters will be utilized to represent identifiers of the substituted sequences. Only the characters from the 14–126 ASCII range are considered for this purpose in order to avoid problems with control codes and non-compatible encoding for non-ASCII characters.

3. Split the contents of the BODY element into a list (A) of non-overlapping character sequences. The splitting is done using a regular expression which tries to expose the character sequences that often tend to appear in the document many times, including:
   a. plain words and numbers (e.g., ‘good’, ‘cat’, ‘128’);
   b. opening and closing tags (e.g., ‘<tr>’, ‘<img’, </td>);
   c. attributes and entities (e.g., ‘width=”’, ‘title=’, ‘ ’);
   d. protocol, server and file names (e.g., ‘http://’, ‘wneiz.pl’, ‘index.html’);
   e. directory name sections (e.g., ‘~user’, ‘service/’).
   Notice that the compressor has no a priori knowledge of what to look for apart from the regular expression defining the split positions, be it dictionary of a specific natural language or a set of HTML tags. It means that it works fine with any version of HTML and any natural language encoded using an ASCII-derived character set.
4. Create a list (B) of unique character sequences found in the document body.
5. Count the occurrences of each character sequence in the document body.
6. Sort the character sequence list (B) depending on the occurrence counter of each character sequence.
7. Leave only those character sequences on the list (B) whose occurrence counter passes the minimum frequency threshold (by default – 2).
8. Assign identifiers to each character sequence of the list (B). There are two types of identifiers:
   a. one character long, for the most frequent sequences;
   b. two character long, for the remaining sequences.
   The set of characters used as one character long identifiers and the set of characters used as the first half (prefix) of two character long identifiers are non-overlapping; the second half (suffix) of two character long identifiers can belong to any of these sets. The number of two character long identifiers is the minimum required to assign every sequence with a unique identifier. If there are more sequences on the list (B) than possible identifiers (even if all characters are reserved for the two character long identifiers, and none for the one character long identifiers), the least frequent items are removed from the list (B) so that every sequence can be assigned a unique identifier.
9. Process the list (A) replacing every occurrence of a character sequence existing in the list (B) with its identifier.
10. Create the compressed file, consisting of:
    a. file type identifier (‘!’, ASCII 33);
    b. the number of one character long identifiers;
    c. the number of characters used as prefixes of two character long identifiers;
    d. the characters used as one character long identifiers;
    e. the characters used as prefixes of two character long identifiers;
    f. the list of substituted character sequences (B), separated with spaces;
    g. a single space (ASCII 32);
    h. the joined elements of the character sequence list (A).

Notice that the actual implementation performs multiple steps of this procedure in single runs, e.g., steps 2, 3, 4, and 5 are done in a single loop.

4. The decompression algorithm

The decompressor works in the following steps:

1. Retrieve the compressed content.
2. Recreate the set of identifiers (from the fields described in items b-e of the point 10 in the previous section).
3. Recreate the list of substituted character sequences (B).
4. Decode the list (A) into the decompressed document replacing identifiers with appropriate character sequences.

The actual JavaScript stub performing this procedure is given below (presented in less compact form than actually used, for clarity):

```
var r = new XMLHttpRequest();
r.open("GET", "filename.gshc", false);
r.send(null);
t = r.responseText;
if (t.charCodeAt(0) == 33) {
    f = t.charCodeAt(1) - 14;
    s = t.charCodeAt(2) - 14;
    var c = new Array(256);
    for (i = 0; i < 256; ++i) c[i] = 0;
    for (i = 0; i < f; ++i) c[t.charCodeAt(3+i)] = i + 1;
    for (; i < f + s; ++i) c[t.charCodeAt(3+i)] = -i;
    i += 3;
    i0 = i;
    i = t.indexOf(" ", i);
    j = 0;
    var d = new Array(f + s * 113);
    while (i - i0 > 1) {
        d[j++] = t.slice(i0, i++);
        i0 = i;
        i = t.indexOf(" ", i);
    }
    t2 = "";
    while (++i < t.length) {
        a = t.charCodeAt(i);
        if (c[a] > 0) t2 += d[c[a] - 1];
        else
        if (c[a] < 0) t2 += d[f - 113 *
            (f + c[a]) + t.charCodeAt(++i) - ${minrng}];
        else t2 += String.fromCharCode(a);
    }
    document.write(t2);
}
```

5. Experimental results

For empirical evaluation of the proposed algorithm, a set of six different HTML documents was selected. They are listed (ordered by document size) in Table 1; their source URL's are given below the table.

In the main experiment, our compressor has been compared to two competitive solutions: Absolute HTML Compressor v. 1.14 [1] and XWRT v. 3.3 [8] running in +h –0 mode (i.e., with second-stage compression disabled). As described in Section 2, the first compressor performs HTML-specific visually lossless compression, whereas the second one performs word substitution in similar manner to our algorithm, and also makes use of some HTML-specific optimizations (mode +h), however it does not produce self-extracting output, so a decompressor is needed to view the compressed page in a browser.

Table 2 lists compression gains obtained for the test documents respectively by the algorithm proposed in this paper (denoted as SXHC for 'self-extracting HTML compressor'), Absolute HTML Compressor, the combination of Absolute HTML Compressor and SXHC, and XWRT mode +h –0. The higher values denote better compression. Notice that the values in both SXHC columns include the negative impact of the size of the decompressor.

As one can observe, the technique of word-based substitution yields significantly better results than those of Absolute HTML Compressor. Apart from the first file, SXHC produced results which were only slightly worse, or sometimes even better, than the off-line XWRT compressor, which is not self-extracting. The case of the first file can be easily explained by its huge HEAD element which is left uncompressed by SXHC.

Table 1
Datasets used in the experiments

No.	File name	Original size (B)
1	Amazon: Getting Things Done	418 576
2	DBLP page of Miroslaw Kutylowski	219 021
3	The Short Second Life of Bree Tanner	201 540
4	Lessons From Central Europe	95 122
5	Pre-history Africa & the Badarian Culture	78 334
6	Python Errors and Exceptions	43 444

Source URL's:

1) http://www.amazon.com/Getting-Things-Done-Stress-Free-Productivity/dp/0142000280;
2) http://www.informatik.uni-trier.de/~ley/db/indices/a-tree/k/Kutylowski:Miroslaw.html;
3) http://www.guardian.co.uk/books/2010/jun/05/stephenie-meyer-second-life-bree;
4) http://polandeconomy.blogspot.com/2009/08/from-original-sin-to-eternal-triangle.html;
5) http://wysinger.homestead.com/badarians.html;
6) http://docs.python.org/tutorial/errors.html#exceptions.

Table 2
Compression results

No.	File name	SXHC	Absolute	Absolute+SXHC	XWRT
1	Amazon…	22.31%	11.74%	33.05%	44.25%
2	DBLP…	58.45%	4.30%	63.85%	57.01%
3	The Short…	36.60%	14.97%	47.43%	40.39%
4	Lessons…	36.82%	8.00%	44.58%	38.72%
5	Pre-history…	39.18%	4.71%	43.37%	38.31%
6	Python…	42.47%	4.37%	46.69%	49.90%
	Average	37.60%	6.47%	51.58%	41.31%

Interestingly, SXHC and Absolute HTML Compressor work together very well. The gains from the two compressors are almost additive.

The aim of the second experiment was to find how well would fare the compressors if combined with Deflate as the final-stage compressor. Figure 1 shows the improvement (positive value) or worsening (negative value) of Deflate compression if applied after the combination of Absolute HTML Compressor and SXHC or XWRT.

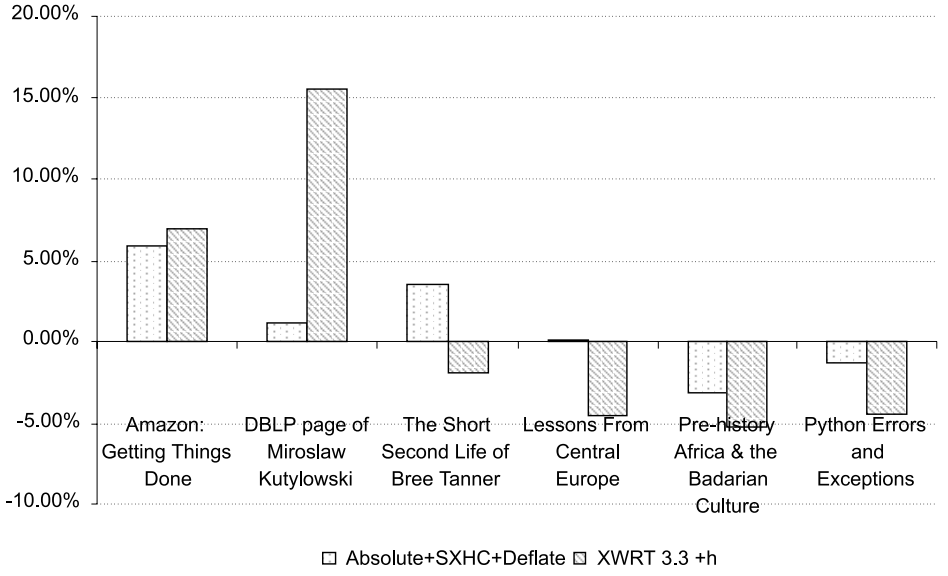

Fig. 1. Impact of prior specialized HTML compression on Deflate compression

As the way of operation of both the proposed algorithm and XWRT to some extent resembles Deflate, the results of combining the former with the latter is not always positive. A gain of several percents was measured for SXHC for three of the tested files, for one of them the result of applying SXHC was almost unnoticeable, for two of them the compression became slightly worse.

As one of the main purposes of compressing HTML documents is reducing time the user waits for them to be displayed in a browser, Figure 2 shows comparison of page load times for original document, SXHC-compressed document, original document with Deflate compression enabled on the server and SXHC-compressed document with Deflate compression enabled on the server. Only two documents were included in the comparison: "Amazon: Getting Things Done" (which, according to Tab. 2 was the least susceptible for SXHC compression), and "DBLP page of Miroslaw Kutylowski" (on which SXHC produced the best result).

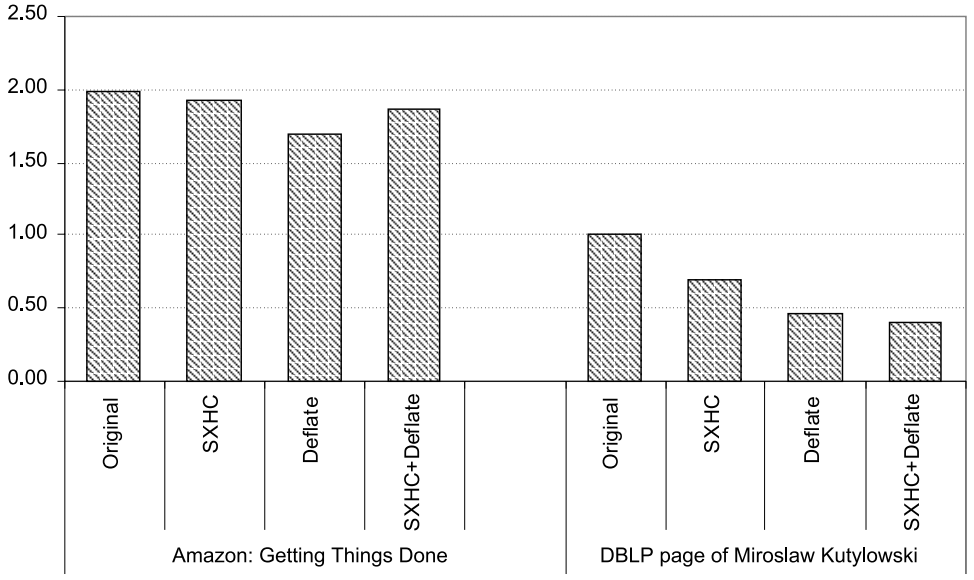

Fig. 2. Page load time (s)

Obviously, the slower the connection, the stronger the positive effect of applying data compression should be; the presented results were measured on client connected via a 6 Mb/s ADSL link (TP-NET), whereas the test files were hosted at one of the University of Szczecin's servers (uoo.univ.szczecin.pl, 1 Gb/s link). The presented times are averages of five subsequent page loads (interleaved with browser cache flush) and were measured on a Google Chrome browser, version 6.0.472.55.

As one can observe, applying SXHC leads to reduction of page load time, though the size of the reduction depends on the actual document content. Although the measured acceleration is smaller than the one obtained by enabling Deflate compression on the server, SXHC can be combined with Deflate as well, to produce even better results, at least for some documents.

6. Conclusions

We presented a compression technique of converting HTML documents into their compact self-extracting equivalents. The algorithm, following a long tradition of text compression, substitutes frequent sequences in the documents with their short (1- or 2-byte) indexes to a semi-dynamic dictionary. Self-extraction is achieved thanks to a JavaScript decompressor added to the resulting document, which runs on page load. In this way, the solution is transparent to a user visiting a given compressed page, and works in any web browser with JavaScript handling turned on.

Experiments on six diverse HTML documents, ranging in size from 43 KB to 418 KB, show that on average more than one third of the original document size can be saved. Moreover, combining our algorithm, called SXHC, with a standard HTML compressor (removing comments, whitespaces, etc.) increases the reduced part to over 50%, while mere removal of whitespaces and other non-relevant parts of the document gives only 6.5% reduction. Finally, we note that applying the Deflate compression algorithm after SXHC results in more or less the same compression as running Deflate alone, which is quite pessimistic as web servers are capable of providing Deflate-compressed content in HTTP/1.1 protocol. On the other hand, our solution shifts the compression burden from the server (compressing the given file on demand) to the webmaster uploading the web page.

There is also a possible problem with clients that do not execute JavaScript, search robots being prime examples. An obvious work-around is to use a technique known as web page cloaking, that is configuring the server to produce different responses for script-enabled and other clients.

In future work we hope to refine our compression algorithm, and in particular devise a more Deflate-friendly variant.

References

[1] Absolute HTML Compressor, v. 1.14, http://www.alentum.com/ahc/ [accessed June 2010].
[2] Fielding R. *et al.*, *RFC 2616, Hypertext Transfer Protocol — HTTP/1.1*. June 1999. Available at http://www.http-compression.com/rfc2616.txt.
[3] Lighttpd, home page. http://www.lighttpd.net/ [accessed June 2010].
[4] Netcraft May 2010 Web Server Survey. http://news.netcraft.com/archives/category/web-server-survey/ [accessed June 2010].
[5] Opera Turbo. http://www.opera.com/business/solutions/turbo/ [accessed June 2010].

[6] Opera Turbo Report, March and April 2010. http://www.opera.com/otr/ [accessed June 2010].
[7] Skibiński P., *Visually Lossless HTML Compression*. 10th Int. Conference on Web Information Systems Engineering (WISE), LNCS 5802, Springer 2009, 195–202.
[8] Skibiński P., *Improving HTML Compression*. Informatica (Slovenia) 33(3), 2009, 363–373.
[9] Skibiński P., Grabowski Sz., Swacha J., *Effective asymmetric XML compression*. Software–Practice and Experience 38(10), 2008, 1027–1047.
[10] Skibiński P., Swacha J., Grabowski Sz., *A Highly Efficient XML Compression Scheme for the Web*. 34th Conference on Current Trends in Theory and Practice of Computer Science (SOFSEM), LNCS 4910, Springer 2008, 766–777.