Optimization of HTML Automatically Generated by WYSIWYG Programs

Jacqueline Spiesser
spiesser@cs.mu.oz.au

Les Kitchen
ljk+www2004@cs.mu.oz.au

Department of Computer Science and Software Engineering
The University of Melbourne
Parkville, Vic. 3010, Australia

ABSTRACT

Automatically generated HTML, as produced by WYSIWYG programs, typically contains much repetitive and unnecessary markup. This paper identifies aspects of such HTML that may be altered while leaving a semantically equivalent document, and proposes techniques to achieve optimizing modifications. These techniques include attribute re-arrangement via dynamic programming, the use of style classes, and dead-code removal. These techniques produce documents as small as 33% of original size. The size decreases obtained are still significant when the techniques are used in combination with conventional text-based compression.

Categories and Subject Descriptors

E.4 [**CODING AND INFORMATION THEORY**]: Data compaction and compression; H.3.5 [**INFORMATION STORAGE AND RETRIEVAL**]: Online Information Services—*Web-based services*; H.5.4 [**INFORMATION INTERFACES AND PRESENTATION**]: Hypertext/Hypermedia; D.3.4 [**PROGRAMMING LANGUAGES**]: Processors—*Optimization, Parsing*

General Terms

Performance, Algorithms

Keywords

HTML optimization, WYSIWYG, dynamic programming, Haskell

1. INTRODUCTION

Major software vendors have exploited the Internet explosion, integrating web-page creation features into their popular and commonly used products to increase their perceived relevance. Knowledge of HTML is no longer necessary to create a web page; users can create a document in Microsoft Word or Microsoft Publisher and these programs can then save the document into HTML form.

While these *What You See Is What You Get* (WYSIWYG) editors have had the benefit of opening the web up to an audience broader than computer professionals and enthusiasts, and arguably have allowed the Internet to take off to the extent that it has, such ease comes at a price: The HTML markup generated by these applications is not of the same standard as hand-coded markup. It is usually technically correct and in accordance with the Document Type Definition, but is also unnecessarily bulky and untidy. Just saving a HTML document to MSWord format then using MSWord to save it back to HTML form typically doubles the size of the file. A large proportion of the blow-out may be due to the long list of style classes included at the start of each document, many of which are never used in the document—MSWord includes every style in the normal template! The rest of the markup is littered with proprietary, vendor-specific attributes and unnecessarily repetitive attributes. For example, it is common for MSWord to put font formatting information on every cell in a table rather than factoring it up to the row level (or higher). Both MSWord and FrontPage do the same for paragraphs and, even worse, place font formatting on paragraphs that do not contain any text at all!

The consequences of such bloated HTML are unnecessarily high storage costs, transmission costs, download times, and browser rendering times, paid for in varying degrees by service providers and end users, in both money and time. The impact of HTML size on these costs in not simple, because of factors like filesystem blocking and network protocol overheads. Still, it is obvious that excessive HTML size has adverse effects across the board.

The aim of this work is to find ways to create the smallest semantically equivalent document for any given HTML document, while still preserving some good coding practice and being XHTML-compliant wherever possible. We focus here on HTML generated by WYSIWYG programs, since for such HTML the problem seems to be most acute. However, as noted in Section 11, it would be of interest to extend this work to a broader range of HTML generators. Because the HTML under consideration is automatically generated and fits the DTD, the parser need not be able to handle incorrect HTML; it can be much less robust than the parsers used by web browsers. This work investigates the effect of the following techniques in reducing HTML document size, both individually and in combination:

- general tidying-up of document, removal of proprietary tags, folding of whitespace;
- exploitation of style classes to decrease document size;
- use of dynamic programming to find the best arrangement of attributes for a document;
- removal of markup that has no effect.

WWW2004, May 17–22, 2004, New York, New York, USA.
ACM 1-58113-844-X/04/0005.

2. OTHER WORK

2.1 Existing HTML Optimization Programs

Little work has yet been done in the area of HTML transformation with optimization of WYSIWYG HTML specifically in mind. At the present moment there exist mostly programs which decrease the size of HTML via more superficial methods, such as collapsing whitespace, removing quotes around attribute values, removing non-W3C approved attributes, and removing closing tags when they are not strictly necessary. Insider Labs' Space Agent [10], and VSE Web Site Turbo [20] are typical examples.

One of the better known HTML transformation programs is HTML Tidy [17]. As the name suggests, Tidy mainly validates and corrects HTML. On the optimization front, it will remove proprietary attributes and factor in-line style out into classes, but will not convert HTML attributes to style attributes. It has an option to ruthlessly prune Word 2000 HTML, but does so at the expense of semantics. Tidy does not remove unused attributes nor optimize attribute placement.

The sparsity of work in the area of HTML transformation and optimization led us to seek ideas and concepts from other fields.

2.2 Finding Structure Through Grammar

The most bulky aspect of a WYSIWYG HTML document is often the attributes applied to each element. This is also the most open to change and re-arrangement while still preserving page semantics. By finding "structure" within sets of attributes (that is, groups often repeated on several elements) it may be possible to factor them out and replace them with an identifier for that group. HTML provides a very effective mechanism for doing this: style classes.

Dictionary-based compression, such as Ziv and Lempel's LZ78, might seem well suited for the task. LZ78 [22] constructs a dictionary online as it passes through and compresses the data, creating longer and longer dictionary entries as longer repeated sequences are found. However, this method was deemed unsuitable as HTML elements must be considered a bag rather than a sequence, and because it would provide sub-optimal performance on earlier elements as it requires time to build a satisfactory dictionary.

Grammar-based compression algorithms initially seemed a better proposition. Larsson and Moffat's RePair algorithm [11] seemed suitable, as it incrementally builds up structure by creating grammar rules that merge adjacent pairs of symbols. These are stored in a dictionary which can then be used to decode. The algorithm seemed applicable to HTML: The dictionary could be formed of style classes, and adjacent symbols in a sequence could be replaced with subsets of a set of attributes.

RePair is extremely similar to Witten and Nevill-Manning's Sequitur algorithm [14], but the RePair algorithm has the added benefit of being offline: An offline algorithm should in theory provide much better compression, as attributes earlier in the document would be compressed just as well as those occurring later. Neither algorithm would be able to act optimally, however, as each HTML element may have only one class attribute. Thus if a set of attributes contained more than one possible optimization, only one could be used.

Additionally, RePair's space efficiency relies heavily on the use of pointers, so that several data structures (a hash-table, heap and linked list) may all point to the same object in memory representing a character, or in this case an attribute, rather than duplicating it three times over. As Haskell, our implementation language (chosen for other reasons), lacks side-effects, such structure sharing would be impossible and the algorithm would lose the space efficiency that makes it attractive for text compression. These issues led us to deem the algorithm unsuitable.

2.3 Compiler Optimization

As HTML is basically code, and the field of compiler optimization of code is quite a mature one, it seemed likely that algorithms from this field might be applied to HTML. Techniques such as loop optimization quite obviously have no application to HTML optimization as there are no loops to optimize, but removal of common subexpressions and especially removal of dead code [1] could be applicable. In the end dynamic programming proved to be a far better way of performing optimizations analogous to common-subexpression removal.

2.4 Dynamic Programming

The 2001 ICFP programming competition [4] was a rich source of ideas. There the task was to optimize a very simple, HTML-like markup language as much as possible within 3 minutes. The markup language used was extremely basic compared with HTML: no document structure whatsoever, only some very simple, finite text decoration. While HTML is greatly more complicated, many of the basic ideas proposed by the contest entrants were transferable.

By far the best and most useful of these ideas was dynamic programming, a method used by many of the best entries including the winner, Haskell Carrots [19]. The Haskell Carrots group recognized that the task was essentially an optimal parsing problem: a renderer computes a display from a parse tree. So the job of an optimizer is to parse, and more so to find the best, cheapest parse tree. The group used the Cooke-Younger-Kasami (CYK) probabilistic parsing technique, mapping the cost of representing a particular section of the parse tree to a probability for use by the algorithm. Similar methods were also used by Tom Rokicki [18] and the Tycon Mismatch team [3], amongst others.

Jurafsky and Martin [9] also describe the use of the CYK algorithm, describing its applications in language processing. Aho and Ullman also describe it, focusing more on the performance aspects of the algorithm and finding it to be order n^3 time and n^2 space [2]. Amusingly, they state that it is doubtful that it will ever find practical use for these reasons!

While such time and space requirements do not, as Aho and Ullman suggest, rule the algorithm out altogether, they do mean that it must be slightly modified to be practically usable. The large sizes of the grammars and input documents often needed exacerbate these requirements. A common solution to this problem is to modify the algorithm to do a beam search, eliminating less-promising parses from each cell as it is constructed [9]. The beam-search algorithm was first implemented by Lowerre in 1968 [12], and is still in common use. The Carrots group used this beam search/thresholding and also used an approximation of global thresholding, which attempts to eliminate parses which are globally unpromising rather than the locally unpromising ones eliminated by beam thresholding. Both thresholding methods are discussed in detail by Joshua Goodman [6], who gives a good account of their pros and cons, and suggests some improvements to the general algorithms.

3. APPROACH

Our approach has three main parts:

1. Build a scaffolding on which to add later HTML optimization algorithms. The scaffolding consisted of an abstract syntax tree (AST) to represent HTML, a parser and an unparser, all implemented in Haskell (Section 4).

2. Investigate the effects of various ways of optimizing HTML. The techniques implemented can be further divided roughly as follows:
    - Superficial but often quite effective "dumb methods", such as removing whitespace (Section 6).
    - The use of style classes (Section 7).
    - The use of dynamic programming to re-arrange markup (Section 8).
    - Standard compiler optimization techniques, in this case dead-code removal (Section 9).
3. Investigate the results of combining the various optimizations, in comparison with and in addition to common text-compression techniques (Section 10).

4. BACKGROUND

4.1 HTML Structure

There are several ways that the formatting in an HTML document may be specified. It can be done by attributes, as in line 12 of Figure 1. This, however, greatly limits the options for the appearance of the document. Cascading Style Sheet (CSS) notation [21] was introduced to provide more expressiveness. CSS notation can be inserted into the document from an external style sheet, in the header of the document (as in Figure 1), or on each individual element (line 13). Header or external style can be linked to a particular element via style declarations for the element type (line 11), style classes which are referenced by the element's class attribute (lines 14 and 15), or style classes referenced by the element's id attributes (line 16).

4.2 Parsing HTML

In order to reorganize the structure of an HTML document, it must first be parsed into an abstract syntax tree. There were several options available to do this.

One of the most immediately obvious options was to take the parser from an open source browser, as such a parser should be very robust and thus capable of handling incorrect HTML. The parsing engine from the Mozilla browser [13] was considered. It immediately became clear that there was a disadvantage to this robustness: the code becomes bloated and complicated in order to handle as many possible cases of incorrect markup gracefully. We decided that, within the scope of this work, dealing with such complications would hinder our investigation of the core issues in HTML optimization.

Another possible parser was the HTML Tidy parser [16]. This provided a good, simple API, and the code is much clearer. Unfortunately, Tidy does not yet have much support for integrating style into documents, which was central to the techniques that we wished to implement.

In the end, the best option proved to be writing our own parser, which allowed us to design our own AST to best fit the ways we wished to manipulate the document. It also allowed us to integrate CSS notation much more closely into the AST, and convert between HTML attributes and CSS attributes. We chose to use Haskell monadic-style parsing owing to its very simple, clear interface.

4.3 Haskell

We chose to implement our algorithms in Haskell [15] (a polymorphically typed, lazy, purely functional language) mainly because we were drawn to the combination of power and simplicity allowed by functional languages. Haskell code has the advantage of being generally clearer, more concise and elegant than imperative code, and additionally is strongly typed (thus when a program type-checks correctly it generally runs correctly). Haskell code is more easily extensible and re-usable on account of polymorphic types and lazy evaluation [8].

Table 1: Features of the test documents.

Doc.	Generator	DTD	CSS	Nesting	Size (bytes)
1	MSWord	loose	yes	no	126759
2	MSWord	loose	yes	yes	63912
3	MSPub	strict	yes	yes	21893
4	MSPub	strict	yes	yes	21648
5	FrontPage	loose	yes	no	45014
6	Excel	loose	no	yes	330859
7	Excel	loose	yes	no	23451

The other main reason for our use of Haskell was the availability of Monadic Parser Combinators [7], which take advantage of Haskell's laziness and higher-order functions to provide an extremely simple yet powerful parser interface.

This allowed us to develop our parser and optimizer more quickly than in a more conventional programming language. However, this comes at a price: for the same task, as Haskell program will normally run much slower than a program written in say C. While our Haskell implementation is the best choice for experimentation and initial proof of concept, it would probably not be suitable for production use.

5. THE TEST DATA

In order to discuss and evaluate various HTML optimizations, there must first be some standard test data for which all the optimization effects can be compared. The test documents used to evaluate and fine-tune the optimizer were carefully chosen to represent the largest range of inefficiencies present in WYSIWYG-generated documents. They were also chosen to present a range of different document structures. However, the document types were restricted to those for which we had access to the generator program, as it is difficult to find such documents on the Internet without various additions and modifications which cause them to violate the DTD. The documents we had access to were generated mainly by Microsoft products: Word, Publisher, Excel and FrontPage. As this bias towards the use of Microsoft products closely reflects real-word usage patterns, it should not be considered too worrying. It would be interesting to see how well the optimizations suggested here can be extended to previously unseen patterns of HTML.

The most common sin committed by WYSIWYG HTML editors is unnecessary repetition of style attributes. All of the generators under consideration do this. Other bad practices observed include Publisher's use of a header style class for each ID, and Word's practice of using header style which is more often than not rendered superfluous by inline style attributes which overwrite it.

Table 1 profiles the seven test documents used throughout this paper. (In this, and in all other tables, document sizes are given in bytes.) All but the Publisher documents use the loose DTD, which allows font elements. All but Document 6 use CSS notation: Document 6 was generated by an earlier version of Excel which did not support CSS. The nesting column refers to the overall structure of the document. Nested documents have "narrow and deep" parse trees, with individual elements having fewer children, but many

```
<html>
<head>
<style type="text/css">
  P { font-size:12pt;}
  P.special {font-style:italic;}
  .bold {font-weight:bold;}
  #134 {font-size:14pt;}
</style>
</head>
<body>
<p> size 12 text </p>
<p> <font color="green"> green, size 12 text </p>
<p style="font-size:15pt; color:green;"> green, size 15 text </p>
<p class="special"> size 12, italic </p>
<p class="bold"> size 12, bold text </p>
<p id="134"> size 14 text </p>
</body>
</html>
```

Figure 1: Use of HTML attributes, and inline and header CSS style notation.

Table 2: Results of parsing and unparsing the input files.

Doc.	Original size	Parsed size	% of original
1	126759	67567	53.3
2	63912	46870	73.3
3	21893	20569	94.0
4	21648	20801	96.1
5	45014	44825	99.6
6	330859	470641	142.2
7	23451	17692	75.4

descendants. Non-nested documents are “broad and shallow”, consisting of structures such as a long series of paragraphs, or lists with many items.

While this data set is quite modest, it is sufficiently representative of this class of HTML documents for investigating the effects of our proposed optimization.

6. PARSING, UNPARSING AND PROPRIETARY ATTRIBUTES

Significant cleaning up of the document can occur during the parse and unparse phases, avoiding many extra passes over the document tree later on. The most basic but also highly effective savings are found in the removal of unnecessary whitespace within and between elements and attributes. Browsers treat these multiple white spaces as a single space, so it makes little sense to leave them in a document.

The exclusion of empty DIV and SPAN elements (i.e., those without any attributes) can also save some space. While these rarely occur in nature, they are generated by the dynamic programming optimizing algorithm (see Section 8.3). They can be removed at a block level with no effect on the semantics of the document.

Microsoft includes many proprietary tags used to encode its messy, unstructured WYSIWYG HTML. Many of these are unnecessary or redundant and can be safely removed without affecting the look of the document, e.g., mso-font-charset, mso-generic-font-family, mso-font-pitch, and mso-font-signature.

Table 2 shows the effect of parsing and unparsing of a document on document size. Parsing has a profound effect on Word documents (1 and 2), the worst offenders in terms of proprietary style attributes, which to compound the problem are displayed in a space-wasting manner. The very small difference in the size of the Publisher documents (3 and 4) is most likely due to removal of excess whitespace, as they use mostly W3C-approved markup. The first Excel document, number 6, on the other hand, underwent a significant size blow-out. This is due to the original document being coded entirely without style markup, which is generally more bulky than HTML attributes. For example, `<center>` is replaced with `<div style="text-align:center">`. This bulkiness is outweighed by the later savings that factoring style out into classes affords.

7. STYLE CLASSES

Style classes were originally designed to allow for more compact representation of style data, as commonly-used style tags can be made a class, which is then all that appears within the body of the HTML code. Unfortunately not all WYSIWYG programs use classes in this manner. Microsoft Publisher, for example, includes header style information for almost every ID in a document and, the purpose of IDs being to act as unique identifiers of an element in a document, this style data will not be reused and may as well be inline. Significant space can be saved by substituting this into the document and removing the header style.

However, the majority of HTML generating programs use classes in more or less the proper manner. Microsoft Word classes, for example, are used many times throughout the document, and thus substituting them in significantly blows out the document size.

Unfortunately, correctly optimizing the document through use of dynamic programming requires classes to be factored in as will be explained in Section 8.7. The blow-out is still so large that it offsets the gains of optimizing, however. The solution is to factor in the classes then factor out the excess again once optimization has occurred.

We initially considered doing this in quite a clever manner using grammatical compression techniques to find structure within sets of attributes, factoring out only the most commonly used attribute subsets. We came to the conclusion, however, that this was an overkill. In the end we settled for the simpler method of factoring whole attributes sets out into classes: if a set of attributes already existed in the header style as a class, substitute that class for the set of attributes, otherwise create a new class in the header style.

The majority of the classes created were used many times over,

```
<td style="height:6.75pt; padding:0cm 5.75pt 12.25pt
5.75pt; border-bottom:solid white .75pt; width:116.2pt;
vertical-align:top; width:155; " colspan="2">

<p style="margin-bottom:3.0pt; font-family:Tahoma;
font-size:11.0pt; " lang="EN-AU"> text </p>

<p style="margin-bottom:3.0pt; font-family:Tahoma;
font-size:11.0pt; " lang="EN-AU"> more text </p>

<p style="margin-bottom:3.0pt; font-family:Tahoma;
font-size:11.0pt; " lang="EN-AU"> even more text </p>

<p style="margin-bottom:3.0pt; font-family:Tahoma;
font-size:11.0pt; " lang="EN-AU"> still more text </p>
</td>
```

Figure 2: An example of repetitive attributes (from Document 2).

and even those used only once create an overhead of less than 10 characters each: add `.cnum { }` and `class="cnum"`, and remove `style=""`. As these styles cross over between all the different elements, they are a more efficient representation than those in a Word HTML or Tidy document.

When substituting style attributes for their classes, it is important to merge them with the attributes already present rather than just appending them to the beginning or end. Many browsers will ignore not only duplicate attributes, but also any attributes occurring after a duplicate.

Table 3 shows the effects of factoring classes into and out of the test documents. As can be seen, just factoring repetitive attributes out into classes reduced the size of all of the test pages by over 15% from the parsed file size. The greatest improvements are seen in the files with the most repetition: Document 5 with its sequence of paragraphs with near identical attribute sets, and Document 6 with its enormous number of similarly formatted table cells. If the formatting is more diverse across the document then the gains are not quite so large. Interestingly, Documents 3 and 4, generated by Publisher, also show improvement when present classes are factored in, reflecting the earlier-mentioned use of header style for ID elements.

8. OPTIMAL ATTRIBUTE PLACEMENT

One of the greatest sins committed by WYSIWYG HTML editors is the repetition of attributes that should be factored up to a higher level. The example in Figure 2 is typical. As can be seen, all four P elements share the exact same font formatting. It would be much more efficient if the formatting were on the TD element instead, avoiding the repetition. Achieving such a re-arrangement of attributes was found to be possible, using dynamic programming. An account of the process follows.

8.1 Heritable Attributes

The first observation that needed to be made is that some HTML attributes are inheritable by child elements while others are not. These will henceforth be referred to as heritable attributes. When present on a parent element, such an attribute also applies to all child elements unless overwritten by another copy of the attribute. That this is the case is intuitive for attributes pertaining to font style, for example. Table 4 contains a list of all of the heritable attributes.

The assumption underpinning the following dynamic programming approach is that if the value of a heritable attribute on a parent node applies to all of its child nodes, then it can also be said that if all or even most of the children of a particular parent node share the same value for a particular attribute, then this attribute value can safely be moved from the child nodes to the parent node. This assumption can be used to reduce the size of an HTML file by replacing many instances of an attribute-value pair with a single instance. The question then, is how to identify instances where it may be cheaper to lift attribute values up to a higher level in the parse tree (and, of course, how to define cheaper). In their paper explaining their response to the 2001 ICFP programming competition [19], the winning Haskell Carrots group made the observation that the optimization of markup is essentially a parsing problem, specifically a probabilistic one. They mapped probability to the cost of representing a particular fragment of the parse tree. The job of an HTML optimizer is to construct the parse tree which generates the cheapest possible (thus the smallest) HTML document.

8.2 Cost

To be able to use dynamic programming to build up the least cost parse tree, we first had to define the notion of cost, as applied to HTML elements and attributes. We chose to define cost as the number of bytes needed to represent the particular snippet of HTML under consideration. Thus a font attribute, for example, has a cost of 4, and an inline style attribute containing the style information to set the font to Ariel (`style="font:Ariel;"`) costs 19 bytes. A DIV element costs 11 bytes: `<div> </div>`.

8.3 Dynamic Programming Algorithm

With costs defined for all of the HTML structures, a chart parsing algorithm could then be applied to the parsed HTML documents. The problem lent itself best to bottom-up parsing, so a variation loosely based on the Cooke-Younger-Kasami (CYK) algorithm was used, with costs rather than probabilities to determine the best parse.

The CYK algorithm (see [19]) is an inductive one, finding successively longer and longer parses across a group of items by merging pairs of shorter parses. It operates in this mode until it has a parse spanning the whole length of the group of items. To use the algorithm a way needed to be found to merge two HTML elements into a single node, thus parsing them. This was done by making them both children of a DIV element, which could then take on any attributes lifted up to a parent level. DIV elements are extremely suitable as an empty DIV element has no effect on a document at block level and can be removed when unparsing. A DIV element with no attributes was given a cost of zero, representing the fact that it does not form a part of the final document. The final DIV, spanning all of the children of the previous parent node, could then have its attributes merged with that parent node, removing this final DIV.

8.4 Grammar

It remained to find a grammar for representing the possible, semantically equivalent arrangements of attributes which the parsing algorithm requires in order to construct the parse tree of DIVs. The problem of using parsing techniques to optimize a parse tree is quite separate from the parsing of the initial document to first create a parse tree. Thus the grammar needed is not the DTD of an HTML document but rather a way of representing equivalent arrangements of attributes.

Construction of this grammar proved to be quite a problem as the number of possible combinations of elements and attributes for HTML is enormous—infinite, for all practical purposes. There was no practical way of specifying all of the possible transitions between parents and children in a fixed, explicit grammar.

Instead, an abstract, pattern-matching style of grammar was first

Table 3: Effect of factoring style attributes out into classes (cin: classes factored in, cout: new classes factored out).

Doc.	Original	Parsed	cin	% orig.	% parsed	cin,cout	% orig.	% parsed
1	126759	67567	75761	59.8	112.1	50161	39.6	74.2
2	63912	46870	59084	92.4	126.1	33532	52.5	71.5
3	21893	20569	18770	85.7	91.3	15345	70.1	74.6
4	21648	20801	18826	87.0	90.5	15388	71.1	74.0
5	45014	44825	44851	99.6	100.1	30084	66.8	67.1
6	330859	470641	470641	142.2	100.0	295282	89.2	62.7
7	23451	17692	18051	77.0	102.0	18046	77.0	102.0

Table 4: Heritable attributes.

azimuth	font-size	list-style-type	speak-numeral
border-collapse	font-size-adjust	orphans	speak-punctuation
border-spacing	font-stretch	page	speech-rate
caption-side	font-style	page-break-inside	stress
color	font-weight	pitch	text-align
cursor	font-variant	pitch-range	text-indent
direction	letter-spacing	quotes	text-transform
elevation	line-height	richness	word-spacing
font	list-style-image	speak	lang
font-family	list-style-position	speak-header	

used to find all the possible parses for each individual attribute type for a pair of elements. These possibilities could then be combined with each of the possibilities for all of the other attributes to find all of the possible parses for a pair of elements.

The grammar for individual attributes has three distinct cases: both elements have the same value for the attribute, the elements have different values for the attribute, or only one element has the attribute.

In the case where both elements have the same value for the attribute, there are two possible parses. Consider the following example HTML markup:

```
<p style="color:red;"> red text </p>
<p style="color:red;"> red text </p>
```

The above can be parsed as either of the following:

```
<div>
<p style="color:red;"> red text </p>
<p style="color:red;"> red text </p>
</div
```

or

```
<div style="color:red;">
<p> red text </p>
<p> red text </p>
<div>
```

When the values for the attribute differ, there are three possible parses. Consider:

```
<p style="color:red;"> red text </p>
<p style="color:green;"> green text </p>
```

The above can be parsed as any of the following:

```
<div>
<p style="color:red;"> red text </p>
<p style="color:green;"> green text </p>
</div
```

Table 5: Effect of applying dynamic programming optimization.

Doc.	optimized	% orig.	% parsed
1	57836	45.6	85.6
2	41198	64.5	87.9
3	15737	71.9	76.5
4	15815	73.1	76.0
5	28111	62.4	62.7
6	164030	49.6	34.9
7	16935	72.2	95.7

or

```
<div style="color:red;">
<p> red text </p>
<p style="color:green;"> green text </p>
<div>
```

or

```
<div style="color:green;">
<p style="color:red;"> red text </p>
<p> green text </p>
</div
```

Finally, when only one element has the attribute, the attribute must remain on that element.

8.5 Effect of Applying Optimization

The results of applying such optimization over a whole document can be seen in Table 5.

Document 6, the first Excel document, realises the greatest effect, as the majority of attributes on the majority of cells are the same. The effect is smaller in relation to the original document than it is in relation to the parsed document owing to the substitution of CSS attributes for HTML, but this is what allows the optimization. There is also a large effect on the FrontPage document, as it has so many similar paragraphs. There is a good size decrease for the

Word and Publisher files, too. However, these documents are a little more diverse in formatting so the effect is not so great.

8.6 Optimizing the Optimizer

Initially the algorithm was run across all the children of an element, with a limit of ten possible parses in each cell of the parse chart. Unfortunately the algorithm took far too long to run (many minutes in some cases) and used enormous amounts of memory, which would make it impractical for use in the real world. Two different methods were used to remedy this problem.

The first was a divide-and-conquer approach to running the algorithm on each set of child elements. Instead of running the algorithm on all of the children of a particular element, we inductively applied it to groups of ten child elements: first to the actual child elements, and then to the best parse trees resulting from the application of it to the groups of child nodes. This was done repeatedly until a set of parse trees spanning all the children was found. The advantage of this approach was that many parse options could be removed from memory much earlier, needing only to be there for the parsing of their group, rather than the parsing of the whole set of child elements.

The other approach was to reduce the number of possible parses stored in each cell. Was is really necessary to keep all ten of them? It eventuated that only five, or even three need be kept, with virtually no increase in the size of the resulting parse tree. Table 6 shows the resulting run-times and document sizes for lower thresholds. (These and other run-times quoted are on a Celeron 466 processor with 256MB RAM, running RedHat 9 Linux, using the Glasgow Haskell Compiler, (`ghc`) [5].) Only a couple of parse trees were smaller for larger cell thresholds, a fact far outweighed by the dramatically reduced run-times resulting from smaller cells.

Note that these run-times are for our experimental Haskell implementation. Experience suggests that an implementation in C would run about ten times faster.

8.7 Problems and Improvements

Upon applying the algorithm, we immediately discovered that many documents and segments of documents did not display as they should. There turned out to be several reasons for this.

One factor affecting the appearance of documents was the precedence of the various types of style attributes applied to particular elements. Inline style overrides style specified by an id attribute, which in turn overrides style specified by a class attribute. WYSIWYG editors often take advantage of this by using overall style classes and then overriding them at a local level with inline style. In addition, each element may have only one class attribute; any extras are ignored. This means that it is important to factor all style classes into the document, onto their associated element, so that the correct style still applies after optimization is applied.

Another problem is that the optimizing process leaves DIV elements in places where, according to the DTD, they should not be. Most of these occur as children of TABLE or TR elements. Most browsers will simply ignore these erroneous elements and any attributes they may have, removing vital formatting information. The solution was to apply any attributes sitting on DIV elements in between back down to the child nodes, once a parse tree of any children had been constructed and merged with the parent node.

9. REMOVING DEAD MARKUP

A common form of compiler optimization of code is dead-code removal. Dead code is code that is not reachable by running the program—it will never be executed and can thus be safely removed. Removal of dead code from a program involves the construction of a directed graph representing the execution of the program, and then removing the unreachable sections.

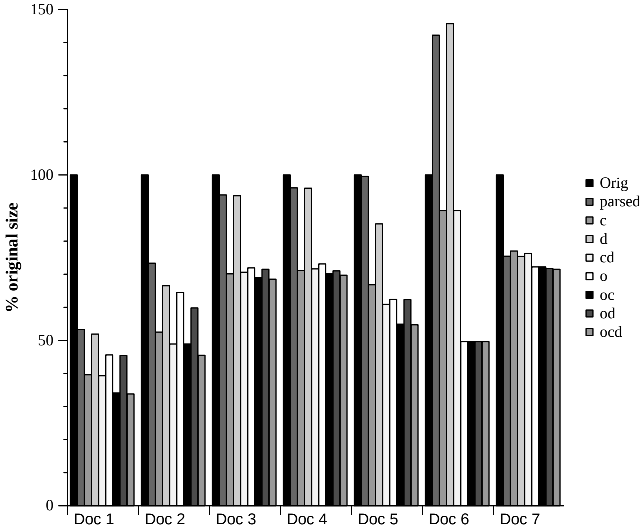

Figure 3: Summary of Effects produced by all optimizations (c: classes factored out, d: dead markup removed, o: dynamic programming). Results are given as a percentage of the original document size.

Dead code has an analogue in HTML: attributes that will never be applied to any element. Most common are font formatting attributes with no text children (see Table 7 for a list of such elements). Microsoft Word in particular does not distinguish between empty paragraphs and those with text in them. For example:

```
<p class=MsoNormal>
    <b><span lang=EN-AU></span></b></p>
<p class=MsoNormal>
    <b><span lang=EN-AU>text</span></b></p>
```

The actual removal of dead elements is trivial as there already exists a directed graph in the form of the document parse tree. It is easy to determine if a node has no text descendants, in which case its text formatting attributes can be removed.

Table 8 shows the effects of applying dead-markup removal to the documents. As can be seen, dead-markup removal has an effect only on HTML documents generated by programs which are prone to the problem, namely MSWord and FrontPage. These documents can have their size decreased by as much as 10–15% over the parsed size. Conversely, the effect on Publisher and Excel-generated documents is negligible. In fact, it has a negative effect for Publisher files in the case where classes are factored out, because it results in the need for some extra classes when some dead attributes have been removed from sets which fit into other classes.

10. DISCUSSION

10.1 Combining the Optimization Techniques

As can be seen from Table 9 and Figure 3, dynamic programming achieves the greatest decrease in document size over the original document: an average of 37.2%. This was followed by factoring classes out, with an average reduction by 33.4%, and finally dead-markup removal with an average reduction by 12.2%. These averages include both the documents for which a method worked well and the documents for which it did not, and so therefore can

Table 6: Execution times (in seconds) and resulting document sizes (in bytes) for thresholds of 3, 5 and 7.

	Threshold: 3		Threshold: 5		Threshold: 7	
Doc.	size	time	size	time	size	time
1	57840	55	57836	74	57836	101
2	41279	25	41198	34	41178	45
3	15795	10	15737	14	15737	20
4	15873	10	15815	14	15815	20
5	28111	30	28111	46	28111	70
6	163830	235	164030	727	163074	515
7	16935	10	16935	11	16935	14

Table 7: Style attributes which apply to visible text.

font	lang	text-shadow
font-family	line-height	tab-stops
font-size	font-size-adjust	text-transform
font-style	font-stretch	
font-weight	text-indent	

Table 8: Effect of dead-markup removal.

Doc.	dead	% orig.	% parsed
1	65826	51.9	97.4
2	42477	66.5	90.6
3	20522	93.7	99.8
4	20784	96.0	99.9
5	38339	85.2	85.5
6	482178	145.7	102.5
7	17692	75.4	100.0

be considered as aceptable results. For example, when only the MSWord and FrontPage documents are considered, dead-markup removal actually provides a decrease of 32.1%. These figures all include the average 10% decrease resulting from parsing the document.

When the different optimizations are combined, the aggregate result is not as effective as the sum of all of its parts, although the aggregate result exceeds any one of them individually. Thus, applying all three optimizations provides only an extra 15.6% decrease over only factoring out classes and 10.5% over dynamic programming. This is because, to some extent, the optimizations overlap in the attributes they affect: much need for dead markup removal is eliminated as dead attributes are lifted to parent nodes along with their siblings. Similarly, optimization decreases the amount of inline style, rendering the effect of switching from inline style to header style classes smaller.

10.2 Reduction in Markup

The optimizations proposed here apply only to the markup; the text content of the HTML document is unchanged. So, in the results presented so far, the effectiveness of the optimizations is to some extent masked by the constant text component. To better isolate the effect of the optimizations, the parsed documents were compared with their optimized counterparts with all text and comments stripped out of both, leaving only the markup. Table 10 shows the results. The average size of the optimized documents was a respectable 58.3% of the parsed documents. However, the results were much better for some of the documents: around 32% for the FrontPage and one of the Excel documents, and around 53% for the two MSWord documents. The overall average is heavily skewed by the result for the last document, which contains little style information to optimize. Note also that this comparison is based on the parsed documents. For most documents, the reduction in markup with respect to the original would be even greater (see Table 2). However, parsing is required in order to separate text content from markup, and so it is convenient to measure the markup reduction at this point, in terms of the parsed document. It would be possible relate this back to the original unparsed markup, but that would be more trouble that it is worth.

10.3 Comparison with Text Compression

So, is it worthwhile it to perform all of these relatively costly optimizations on HTML documents when instead they could just be compressed with a conventional text compression algorithm? Table 11 compares the results of compression alone on the original HTML with the results of compression *after* our optimization. Clearly, these documents are highly redundant, and compress very well using conventional text-based compression techniques. In fact, the size reduction from compression alone is greater than the compression obtained by our optimization techniques on their own. (See Table 9.) This is hardly surprising, since our optimization techniques leave the textual content of the document unchanged, and leave the markup (tags and attributes) spelt out in full, all of which can be compressed as text. What is of interest, however, is the combination: Compressing the result of our optimizations gives a file size significantly smaller than can be achieved by compression alone. There is an average 21.3% decrease for the optimized files for gzip and an average 16.5% decrease for bzip2.

Both techniques remove redundancy from the documents, and there is a certain overlap between the two: The optimized documents compress less well than the originals, because the optimization does remove some textual redundancy. See Table 11. Likewise, the effects of our optimizations are reduced after compression, because some of the redundancy removed by our optimization could also have been removed by compression. Compare Tables 9 and 11. However, there remains some “higher level” redundancy which can be removed by our optimizations, but which cannot be removed by mere text-based compression. Hence, combining the two techniques, optimization followed by compression, can still produce further worthwhile size reductions.

Which combination of text compression and HTML optimization would be best in a particular situation obviously depends on the interaction of a number of factors and trade-offs, for example, storage and computational load on server and client, transmission bandwidth and costs, latency, and frequency of access.

Table 12 shows the relative compression and decompression costs for gzip and bzip2. It seems that, of text-based compression techniques, gzip is likely to be more useful for the purpose of compress-

Table 9: Summary of the effects produced by all optimizations (c: classes factored out, d: dead-markup removed, o: dynamic programming). Results are given as a percentage of the original document size.

		% of original size							
Doc.	original	parsed	c	d	cd	o	oc	od	ocd
1	126759	53.3	39.6	51.9	39.3	45.6	34.1	45.4	33.8
2	63912	73.3	52.5	66.5	48.9	64.5	48.9	59.8	45.5
3	21893	94.0	70.1	93.7	70.6	71.9	68.9	71.5	68.5
4	21648	96.1	71.1	96.0	71.6	73.1	70.1	71.0	69.7
5	45014	99.6	66.8	85.2	60.9	62.4	54.9	62.3	54.7
6	330859	142.2	89.2	145.7	89.2	49.6	49.6	49.6	49.6
7	23451	75.4	77.0	75.4	76.3	72.2	72.2	71.7	71.5
Average		90.6	66.6	87.8	65.3	62.8	57.0	61.1	56.2

Table 10: Effect of optimization on the documents with text and comments removed.

Doc.	parsed	optimized	% parsed
1	53258	27638	51.9
2	40709	21939	53.9
3	20269	14707	72.6
4	20501	14785	72.1
5	29943	9738	32.5
6	446908	138695	31.0
7	14405	13512	93.8
Average			58.3

ing and decompressing HTML of servers and browsers as its compression is 8 times faster and its decompression is 3 times faster.

It should be kept in mind that, while our optimizations may be more costly than conventional compression, they produce immediately usable HTML output, and therefore entail no "decompression" cost whatsoever.

11. CONCLUSION AND FURTHER WORK

In conclusion, the optimization techniques analysed in this paper achieve a significant reduction in the size of WYSIWYG HTML documents: the markup in the test documents was reduced to an average of 58% of its original size. The results were even better for some document types—with all text removed, the markup alone was reduced by 68% to 32% of its original size. This translated to an average reduction of whole HTML documents to 56% of their original size.

In this work we succeeded in our aims of investigating and identifying the aspects of HTML mark-up that are able to be changed while still leaving a semantically equivalent document. We developed techniques to improve the HTML aspects identified, including the removal of whitespace and proprietary attributes, dead-markup removal, the use of header style classes and dynamic programming. For dynamic programming, we extended ideas presented by entries in the 2001 ICFP programming competition to a real-world markup language and dealt with all the pitfalls of this more complicated language.

We conducted quantitative experiments on the performance of the various techniques, both individually and in combination, and compared the performance of our techniques to simple, text-based compression. We optimized the algorithm to make it run as fast as possible so that it may eventually be practically useful, as this is a project with a very practical aim.

Much can still be done to extend and improve on the work presented in this paper. Some possible extensions include:

- Perform thresholding on dynamic programming parse chart cells based on "goodness" of a particular parse rather than on a strict cell quota. This would make the thresholding method closer to traditional beam thresholding.

- Attempt to get closer to the semantic meaning of a page, and create HTML markup from scratch rather than re-arranging what is already there. This would require each element in the parse tree to know all formatting applied to it, including that inherited from parent elements.

- In relation to the above, extend the dynamic programming grammar for the case where only one of a pair of elements has a particular attribute so that the attribute may be lifted to a higher level, and then overridden on the element to which it does not apply. This would require the optimizer to have a greater understanding of document semantics in order for it to understand how to undo changes applied by an attribute.

- Improve the parser and general algorithm to handle the peculiarities of HTML documents from a broader range of WYSIWYG editors, and indeed other HTML generators, including perhaps non-conforming or errorful HTML.

- If appropriate, re-implement these optimizations in a more efficient language, both for ultimate practical use, and also to make larger-scale experiments (see next item) more feasible.

- Enlarge the test set to include a larger number of documents, from a broader range of sources.

- Investigate in more detail the trade-offs of these various optimizations vis-a-vis conventional compression, their effect on storage costs, transmission times and latencies, etc.

- Investigate where these optimizations might fit into web delivery: For example, could they be effectively integrated into caching servers, or into the back-end of HTML generators?

12. ACKNOWLEDGEMENTS

Thanks to Harald Søndergaard for pointing us towards the ICFP programming competition and dynamic programming. Also, thanks to James, Andrew, Tim, Scott, Charlotte and everyone in 2.19 for company and helpful discussion, and to family and friends for being understanding and supportive, especially Kwan and Marcie.

13. REFERENCES

[1] Alfred V. Aho, Ravi Sethi, and Jeffrey D. Ullman.
Compilers: Principles, Techniques and Tools.
Addison-Wesley Publishing Company, 1988.

Table 11: Effect of zipping original and optimized files

		gzip			bzip2		
Doc.	original size	gzip alone	gzip after optimization	% difference	bzip2 alone	bzip2 after optimization	% difference
1	126759	9887 (7.8%)	6769 (5.3%)	31.5	8902 (7.0%)	6119 (4.8%)	31.3
2	63912	5901 (9.2%)	4809 (7.5%)	18.5	5728 (9.0%)	4669 (7.3%)	18.0
3	21893	4608 (21.0%)	4063 (18.6%)	11.8	4503 (20.6%)	4034 (18.4%)	10.4
4	21648	4784 (22.1%)	4061 (18.8%)	15.1	4711 (21.8%)	4049 (18.7%)	14.1
5	45014	8321 (18.5%)	7921 (17.6%)	4.8	7197 (16.0%)	7073 (15.7%)	1.7
6	330859	10568 (3.2%)	8474 (2.6%)	19.8	4915 (1.5%)	5129 (1.6%)	–4.4
7	23451	4495 (19.2%)	2348 (10.0%)	47.8	4113 (17.5%)	2277 (9.7%)	44.6
Average				21.3			16.5

Table 12: Time to compress or uncompress Document 1, 1000 times

Compression Tool	Time
bzip2	381s
bunzip2	79s
gzip	46s
gunzip	23s

[2] Alfred V. Aho and Jeffrey D. Ullman. *Principles of Parsing, Translation and Compiling*. Englewood Cliffs, N.J., Prentice-Hall, 1972.

[3] Perry Cheng, Tom Murphy, Charlie Smart, Dave Swasey, and Joe Vanderwaart. *Tycon Mismatch 2001 ICFP Programming Competition Entry*, 2001. `http://www-2.cs.cmu.edu/~tom7/icfp2001/`.

[4] Damien Doligez, Luc Maranget, and Pierre Weis. *2001 ICFP Programming Competition 2001*, 2001. `http://cristal.inria.fr/ICFP2001/prog-contest/`.

[5] Glasgow Haskell compiler. `http://www.haskell.org/ghc/`.

[6] Joshua Goodman. Global thresholding and multiple-pass parsing. In *Proceedings of the 2nd Conference on Empirical Methods in Natural Language Processing*, pages 11–25, 1997.

[7] Graham Hutton and Erik Meijer. Monadic parser combinators. Technical report, University of Nottingham, NOTTCS-TR-96-4, 1996.

[8] Simon Peyton Jones. *Why use Haskell*. haskell.org, 2001. `http://www.haskell.org/aboutHaskell.html`.

[9] Daniel Jurafsky and James H. Martin. *Speech and Language Processing*. Prentice-Hall Inc., 2000.

[10] Insider Labs. *Space Agent: Web Site Optimization*, 2003. `http://www.insiderlabs.com/spaceagent/`.

[11] N. Jesper Larsson and Alistair Moffat. Off-line dictionary-based compression. *Proceedings of the IEEE*, 88(11):1772–1732, 2000.

[12] B.T. Lowerre. *The Harpy Speech Recognition System*. PhD thesis, Carnegie Mellon University, Pittsburgh, PA, 1968.

[13] mozilla.org. *Mozilla Layout Engine*, 2003. `http://www.mozilla.org/newlayout/`.

[14] Craig G. Nevill-Manning and Ian H. Witten. Identifying hierarchical structure in sequences: A linear-time approach. *Journal of Artificial Intelligence Research*, 7:67–82, 1997.

[15] John Peterson and Olaf Chitil. *Haskell: A purely functional language*. haskell.org, 2003. `http://www.haskell.org`.

[16] Dave Raggett. *HTML Tidy Documentation*. W3C, 2002. `http://tidy.sourceforge.net/docs/api/`.

[17] Dave Raggett. *Clean up your Web pages with HTML TIDY*. W3C, 2003. `http://www.w3.org/People/Raggett/tidy/`.

[18] Thomas Rokicki. *Tom Rokicki's 2001 ICFP Programming Competition Entry*, 2001. `http://thomas.rokicki.com/icfp2001/`.

[19] Chung-Chieh Shan and Dylan Thurston. Markup optimization by probabilistic parsing. `http://www.digitas.harvard.edu/~ken/icfp2001/d2-a4.pdf`, 2001.

[20] VSE. *Optimize your Web Page with VSE Web Site Turbo*. vse-online.com, 2003. `http://vse-online.com/web-site-turbo/`.

[21] W3C. *Cascading Style Sheets, Level 2 CSS2 Specification*, 1998. `http://www.w3.org/TR/REC-CSS2/`.

[22] J. Ziv and A. Lempel. Compression of individual sequences via variable rate encoding. *IEEE T. Information Theory IT*, 24 (5):530–536, 1978.