Applying Formal Concept Analysis to Cascading Style Sheets

David Federman
federman@cs.utexas.edu
Advisor: William Cook
Department of Computer Science
University of Texas at Austin

Abstract. Cascading Style Sheets (CSS) are used in the HyperText Markup Language (HTML) to describe the style, size, color, and position of elements in a document. While simple styles are easy to specify, a style sheet for a complex site can become many thousands of lines long. One problem in style sheets is that as they grow there is a tendency for increasing duplication of styles and properties, especially when the style sheet is used for many pages created by multiple authors. Formal Concept Analysis (FCA) is a technique for eliminating redundancy while identifying common concepts in a complex space of property definitions. In this work we use FCA to optimize style sheets to reduce redundancy, merge several rules together, and group the selectors with their declarations to express general formatting concepts. Two problems we solved include converting complex style sheets into a form on which FCA can be applied, and interpreting the resulting concept lattice to avoid introducing too many styles for small concepts. We evaluate the effectiveness of the solution on several style sheets.

1 Introduction

Typically, *Cascading Style Sheets* (CSS), which are used to stylize *Hypertext Markup Language* (HTML) documents, are touched by multiple authors and evolve over a long period of time. This leads to messy or inefficient code such as duplicate styles and style which should partially grouped together. This inefficiency can lead to poor website performance and a bad end-user experience, as well as some difficulty for future web developers who may add on to the CSS style sheet. We have written a program which will run a particular technique called Formal Concept Analysis (FCA) upon the CSS to make it more optimum with respect to file size and other metrics.

1.1 Why Optimize CSS?

Because CSS is generally linked as an external file, the contents of this file can be cached in the browser[1], so why is there such an emphasis on optimizing a file which will typically only need to be loaded once or less on average per site visit? One large reason is because web crawlers do weight some sites based on page load[2]. Now page load can be a subjective definition, but we are loosely defining it as from the time when the browser makes the GET request to the server until most of the document's content is loaded and rendered. The reason why we say "most" is due to asynchronous JavaScript (sometimes known as AJAX) requests which may end up changing the content of the page repeatedly and indefinitely, but we consider the page loaded when all but these AJAX calls are complete if the AJAX calls do not greatly change the HTML document.

The aforementioned web crawlers typically will not have cached copies of the externally linked files and must load them for the first time when they gather page load data. Also, the initial page load time may determine whether a user will stay on the page or find another web site to browse[3]. Of course, with this initial load comes the load of the CSS file. Thus optimizing the CSS helps with the optimization of the initial page load[4] and therefore may affect the user-experience and reduce what is know as the "bounce rate"[5]. Additionally, our approach will not only decrease file size, but also decrease the total number of rules and selectors. This can be very important to performance[6].

Figures 1 and 2 show a non-optimized CSS file with its optimized counterpart. They both specify the same styles on the same object, but the optimized version is more concise.

Figure 1 An example of unoptimized CSS

```
a
{
  color: black;
  font-weight: bold;
}

b
{
  color: black;
}

c, d
{
  color: black;
  padding: 0;
}

e
{
  padding 0:
  font-weight: bold;
}
```

2 Background

2.1 Cascading Style Sheets

CSS is a style sheet language[7] used in conjunction with and to supplement HTML as an example of the presentation layer of the OSI Model[8] applied to web browsing. Typically and ideally, the idea is that HTML should provide structural information about the web document along with the document's contents, while CSS should provide stylistic and purely presentational aspects of the web page[9, 10, 11]. For instance, HTML may specify that there are a certain number of containers on the page with certain textual headers and content inside, while the CSS may specify the font face, font color, container padding, background colors, and other such cosmetic properties for the document. Although CSS may be embedded in HTML in various

Figure 2 An example of optimized CSS

```
a, b, c, d
{
  color: black;
}

c, d, e
{
  padding: 0;
}

a, e
{
  font-weight: bold;
}
```

ways[1], generally CSS is linked as an external file and this is the type of CSS we will be discussing.

A CSS style sheet is comprised, for the most part, by a list of CSS rules, much like a C program is basically a list of functions. A *rule* is defined as a set of selectors which have a set of declarations applied to them. A CSS *selector* is a string in the CSS document which specifies, or "selects", which HTML tags, classes, or IDs have the associated style. *Declarations* are a property-value pairs such as `color: black` or `border: 0px`. A *simple selector* selects just one group of HTML elements, while *complex selectors* are comma-delimited lists of simple selectors. Figure 3 is a small example of a complete CSS style sheet with two rules, one with complex selector and one with a simple selector, each with two declarations.

Figure 3 A basic CSS style sheet

```
/* A rule specifying HTML tags 'a' and 'b' become black and bold */
a, b /* A complex selector containing two simple selectors */
{
  color: black; /* A declaration */
  font-weight: bold;
}

/* HTML elements with class='c' become normal weight and have no border */
.c /* One simple selector */
{
  font-weight: normal;
  border: 0;
}
```

2.2 Formal Concept Analysis

Formal Concept Analysis is an algorithm which takes a relation of objects and properties and creates a lattice of nodes, called *concepts*[12, 13]. Each concept consists of a pair of sets: a set of objects and a set of attributes. The objects in the concept have all the attributes in the concept. The lattice of concepts is organized in a way that the attribute set of upper nodes are a subset of its lower neighbors, and the object set of lower nodes are a subset of its upper neighbors. Thus the top node always contains all objects and the intersection of all the attributes they have, which is typically the null set. Likewise, the bottom node always contains all attributes and the objects which have all properties in the system, also typically the null set.

Figure 4 shows a concept lattice where the set of object is the numbers one through ten and the set of attributes are {`square, prime, composite, even, odd`}. Using this lattice, certain information is readily available by looking at particular concepts like the set of even and prime objects, which properties the numbers one and nine share, what properties the number 4 has, etc.

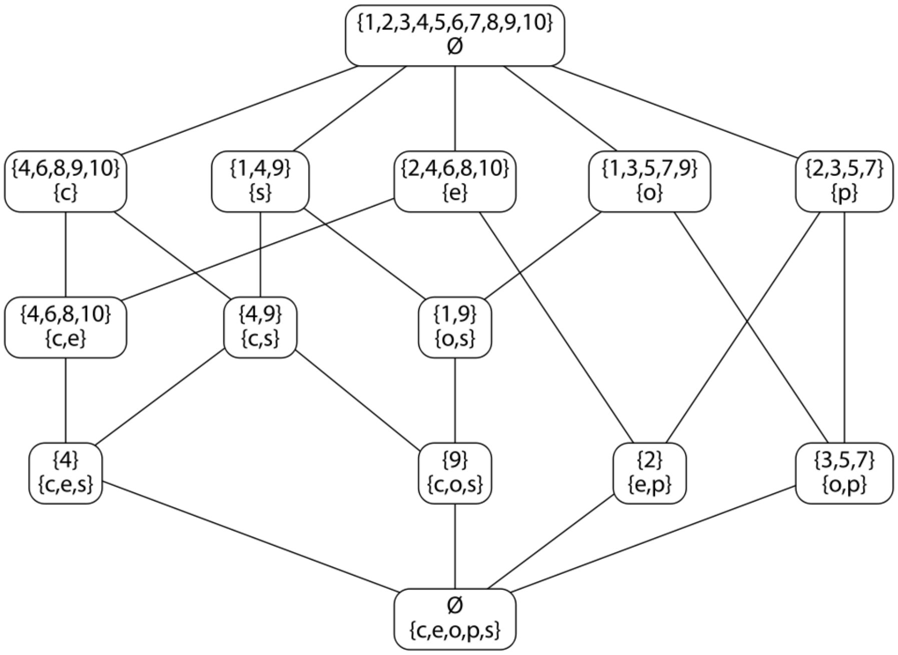
3 Preparing CSS for Formal Concept Analysis

As is typical with any optimization technique, the input file needs to be parsed and converted into an internal representation useful to the program. We use various classes we created to for what is called an abstract syntax tree, which are objects that represent the structure of the input file and can be manipulated by the program rather than being just a long input string. Also, the CSS specification does not perfectly map to the input that FCA requires. Because of this, we need to do some pre-processing on the abstract syntax tree to prepare the CSS for FCA.

3.1 Parsing

It is very difficult to make useful changes to documents containing code without first parsing the document and converting it into to some sort of internal representations like an abstract syntax tree in our case. This is an internal representation which makes working with the document much easier than just string

Figure 4 A concept lattice, with the numbers 1 to 10 as objects and attributes square, prime, composite, even, and odd (shown as initials)[14]



manipulation. We used a Java-based parser generator called Rats![15] to create our CSS parser. The grammar we used was based on the CSS 2.1 specification[16]. From there we created the abstract syntax tree Java classes and were then able to continue the pre-processing of the CSS.

3.2 Simplifying Complex Selectors

The original input file may contain rules which have complex selectors. To convert the CSS into a format which we can more easily use FCA upon, we decided to simplify all complex selectors. To do this, we iterated through all rules with complex selectors and broke them up into multiple rules with simple selectors and identical declaration bodies. This results more-or-less in a list of full selectors each with a list of properties they have. Duplicate selectors may exist at this point, as well as duplicate declarations, which will all be reduced by FCA.

3.3 Mapping CSS to Concepts

CSS almost maps directly to FCA, but there are a few irregularities which need to be accounted for. We used simple selectors as the object input to FCA, and declarations as the property input to FCA. Because of this, the relation between selectors and declarations remain intact, but CSS inheritance comes into play and needs to be handled. Also, selectors are defined to potentially select zero or more HTML elements rather than exactly one. This is another aspect of the mapping issue we addressed.

3.3.1 Classes of Object vs Particular Objects

Because CSS selectors do not specify particular HTML elements, but rather classes of elements, multiple rules can apply to one elements, and one elements may have several rules which apply to it. FCA requires particular objects to be provided, so this was a point of concern. However, we did not want to introduce particular HTML elements into the problem due to a loss of generality. However, because FCA takes particular objects to produce concepts containing particular objects, we concluded that it could also map classes of objects to concepts containing classes of objects. Because of this we decided that the fact that CSS specified classes of objects was inconsequential to the problem and that FCA would still work as intended and produce correct results.

3.3.2 Handling CSS Inheritance

One of the biggest challenges we faced was the fact that FCA only maps the relation of objects and their properties and not the relationship between different objects, like when one object implicitly inherits from another. Because of this, CSS selector inheritance was a major point of concern for us. An example of CSS selector inheritance is that all declarations that are defined on the `a` selector are implicitly defined on the selector `a.b` except for the declarations which may be overridden by `a.b` as defined by the CSS specification.

To handle this, we decided to iterate through all selector pairs and if one of the selectors would inherit from the other, we would copy all declarations from the more general rule to the inheriting rule. This explicitly defines all declarations which would be inherited to the selectors which would inherit them. We had to also check to see if the inheriting rule was overriding the particular declaration because, if so, we would not want to copy that declaration. As with simplifying complex selectors, this makes the style sheet larger and more verbose, but more simpler for FCA to use. After FCA is applied, this increase in verbosity will reduce and be cleaned up.

3.4 Aliasing

The FCA library we used, colibri-java, does not work well with different but equal objects; objects where the `.equals` method returns true but the objects do not reside at the same memory location. Also, the parser generator and abstract syntax tree we developed creates new objects for each selector and declaration, even if they already exist in another rule. This was done for generality, as we did not want to have unwanted side-effects if we, for instance, wanted to change a selector to a different selector. This would be less of a problem if we decided to make our abstract syntax tree classes immutable.

Because of these two design decisions, we needed to get rid of these extra objects and alias the references to the same object if they were equal. How we accomplish this is we iterate through all rules, their selectors, and their declarations, and if any are equal to each other, but not the same object, we reduce them by aliasing one to the other and allow the now-unreferenced object to be garbage collected.

An alternative which we decided not to explore is to have the parser create objects using a factory. If an equal object already exists at the time, return a reference to it, but if not then create the object and return it. This method could use less memory to parse, but would loose the flexibility of being able to use mutable abstract syntax tree objects.

4 Applying Formal Concept Analysis

After pre-processing, we had our abstract syntax tree in a format which was ready for FCA. After complete though, we still needed to traverse the lattice that FCA created and do some post-processing to re-introduce the CSS inheritance that we extracted in the pre-processing.

4.1 Creating the Lattice

Prior to this point, we were treating property-value pairs in the intuitive way. When considering these things for FCA, we needed to consider the property-value pair as just one property since objects in FCA either have a property or they do not; there is no notion of a property having a value. For example, when handling `color: black` and `color: white` prior to this part of the program, they had the same property but different values and were treated as such in regards to overriding CSS inheritance. When passing these same two pairs to the FCA library, we considered them as two completely separate properties. Thus some objects could have `color: black`, some `color: white`, some neither, and theoretically some could have both (although we put a restriction on this prior to this point).

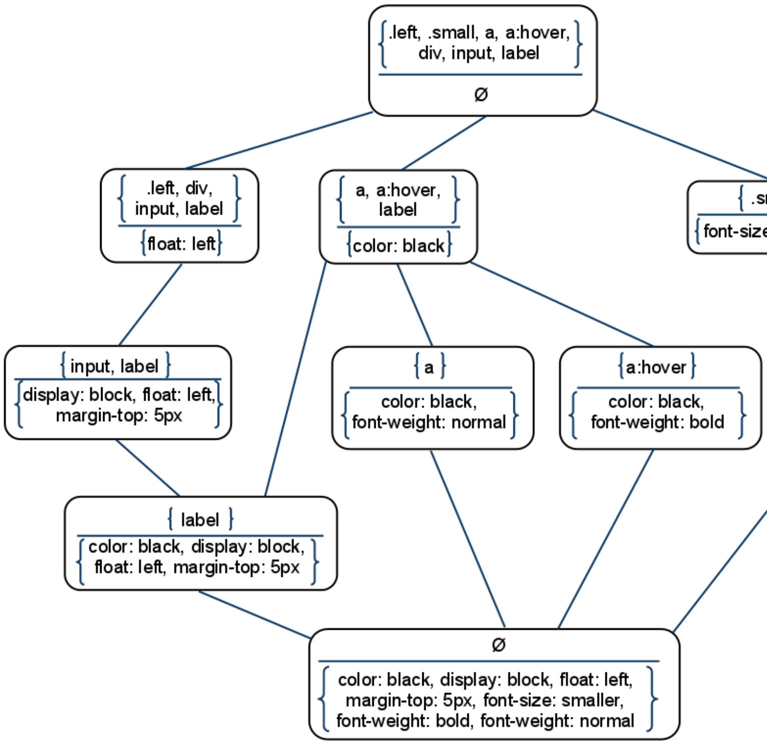
We used an open-source java library called colibri-java[17] to perform FCA and create the lattice. We iterated through each rule and added each object-property pair to the relation used in the library. The library then created the lattice of concepts and returned it to us for us to traverse. Once the lattice was created, we needed to traverse and interpret the lattice.

4.2 Traversing the Lattice

While traversing the lattice, we added the visited concept as a rule, where the set of objects became the complex selector for the rule, and the set of properties became the declaration block. Because all properties of upper concepts are duplicated in lower concepts, we removed these duplicated properties from the rules formed by the lower concepts. Similarly, the lower neighbors for a particular concept have a subset of the concept's objects, these repeated objects were not included. Now there were multiple ways to traverse the lattice and each produced different results.

Figure 5 shows a full concept lattice with CSS selectors as the concept objects and CSS declarations as the concept attributes.

Figure 5 Example CSS concept lattice


{.left, .small, a, a:hover, div, input, label}
Ø
{.left, div, input, label}
{float: left}
{a, a:hover, label}
{color: black}
{.small}
{font-size: smaller}
{input, label}
{display: block, float: left, margin-top: 5px}
{a}
{color: black, font-weight: normal}
{a:hover}
{color: black, font-weight: bold}
{label}
{color: black, display: block, float: left, margin-top: 5px}
Ø
{color: black, display: block, float: left, margin-top: 5px, font-size: smaller, font-weight: bold, font-weight: normal}

4.2.1 Top-Down Traversals

Top-down traversals are the most intuitive approach to traversing the lattice. These traversals begin from the top of the lattice, or the concept whose set of objects contains all the objects in the relation and whose set of attributes contains the properties shared by all objects, which is typically the empty set. This traversal tends to favor object-groupings in that it produces rules with a large complex selector and small declaration block. Because of this, top-down traversals go hand-in-hand with object count traversals.

4.2.2 Object Count Traversals

Object count traversals traverse the lattice by visiting the concepts with the largest object sets before visiting those with smaller object sets. This goes well with top-down traversals and in our final program, our top-down traversal uses an object count traversal of the concept's lower children in that it will traverse from top down, but visit its lower neighbors in descending order of their object count. A side-effect is that each unique declaration usually appears in the final CSS only once.

4.2.3 Bottom-Up Traversals

Bottom-up traversals begin from the bottom of the lattice, which is the concept with the set of attributes containing all declarations in the sheet and the object set containing the objects that have all attributes in the sheet, which is typically the empty set. This traversal goes very well with attribute count traversals, as it favors attribute-groupings. Rules produces by this traversal tend to have a large declaration body and relatively small and simple selectors.

4.2.4 Attribute Count Traversals

Attribute count traversals visit concepts in the lattice which have a larger property set than those with smaller property sets. Because of this, it goes hand-in-hand with bottom-up traversals. In fact, our bottom-up traversal visits upper-neighbors in descending order of their property set sizes. The results of this kind of traversal typically only contain each selector once, with the possible exception of when a particular selector has a majority of its declarations shared with another particular selector.

4.2.5 Skip Singleton Traversal

The skip singleton traversal is interesting. It visits nodes in the top-down or bottom-up pattern, but when it comes across a concept which only contains one property, it holds off on adding that rule. This way, it gives other concepts a chance to express that particular relationship. If one of these concepts is not found, the rule is added in at the end. This traversal can produce interesting results, as it forces concepts to contain multiple shared properties instead of just one if at all possible. This traversal does not typically produce better results than the others in terms of file size or other objectively measurable metrics, but can group objects and attributes in ways that may make more semantic sense. For instance, it may produce a rule with five or so selectors which all share several declarations such as border width, border color, background color, padding, font-style, etc. Perhaps these all look like a button and so the web developer may decide to just create a separate class for these groupings of declarations and thus be able to remove the declarations from the set of selectors in the concept. Of course, the groupings may not make more semantic sense, but this is very subjective and up to the discretion of the web developer.

Figures 6 and 7 show an example of CSS optimized using some traversal and CSS optimized using the skip singleton traversal, respectively. If, perhaps the fact that the classes `a1`, `b1`, and `c1` all are black, bold, and have no padding semantically makes sense and is a “natural” grouping, so one thing the author could do is create a new class called `d` and give it those shared declarations, then add that new class to all the HTML elements which have the `a1`, `b1`, or `c1` classes. If all declarations are shared, the classes could even be replaced on those HTML elements.

4.2.6 Other Traversals?

There are many more traversal methods which we did not explore which could be explored and quantified. For instance, instead of skipping just singleton selectors, perhaps skip sets of two or three. Some other ideas could be to traverse the lattice from the left or right corner or even a random traversal. These traversals which we did not explore all would produce different and potentially interesting results.

4.3 Reintroducing Inheritance

Because FCA knows nothing about CSS inheritance, it may group selectors together in which one always inherits all declarations from the other. This is especially true because of the pre-processing we did where we explicitly duplicated all inherited declarations. An example is that a rule may be produced in which the set of selectors contains both `a` and `a.b`, but the `a.b` selector is not needed since all declarations that apply to `a` also apply to `a.b` always. To remove these extraneous selectors, we iterate through all rules and check each selector pair within the rule to check if one selector inherits from the other. If so, the inheriting selector is removed from the rule.

Figure 6 CSS which could use the skip-singleton traversal

```
.a1, .a2, .a3, .a4, .a5, .b1, .c1
{
  color: black;
}

.b1, .b2, .b3, .b4, .b5, .a1, .c1
{
  font-weight: bold;
}

.c1, .c2, .c3, .c4, .c5, .a1, .b1
{
  padding: 0;
}
```

4.4 Merging Rules with Duplicate Selectors

Due to reintroducing inheritance and removing some selectors from rules, sometimes there will be two rules with identical selectors. For example, if a rule contained only selectors `a` and `a.b`, the `a.b` selector would be removed, leaving the whole selector for the rules as `a`, which may already exist in the sheet. To correct this, we go through all rule pairs and if selectors are the same, we merge the declaration blocks into one of the rules and remove the other rule from the sheet, allowing it to be garbage collected.

At this point the CSS style sheet has been optimized by FCA, converted back into the abstract syntax tree classes we created, and a bit of post-processing cleanup has been performed. We then are free to continue performing other optimizations, analyses, or just print the sheet out for the user to see the optimized CSS style sheet.

Figure 7 An example of the skip-singleton traversal

```
/* This rule could be consolidated into one class called .d */
.a1, .b1, .c1
{
  color: black;
  font-weight: bold;
  padding: 0;
}

.a2, .a3, .a4, .a5
{
  color: black;
}

.b2, .b3, .b4, .b5
{
  font-weight: bold;
}

.c2, .c3, .c4, .c5
{
  padding: 0;
}
```

5 Discussion

5.1 What is Optimal?

One topic we discussed was what makes a style sheet optimal? Some options are file size, the total number of rules, the total number of selectors, the total number of declarations, or even semantically optimized in that it is easier for humans to read and maintain and if the groupings make sense. Because of this and also because different traversals produce not only different results, but are inconsistent on the quality of results they do produce, it is difficult to have a singular black-box program that is just fed an input and out pops the best result. In fact, some traversals even increase the file size for particular style sheets!

In general, we are looking at file size to be the main metric for quantifying how optimal a style sheet is. The reason we decided upon this is because if we had based it on the number of rules, selectors, or declarations, we would be making assumptions of how different browsers render the HTML on the page when applying CSS. For example, one browser may make significant improvements when reducing the number of rules while the number of selectors may make little to no effect, while another browser's rendering may be more tied to the number of selectors and not the number of rules. Since we did not want to make these assumptions and to keep our analysis as general as possible, we are choosing file size as the most important metric to determine optimization. Nevertheless, we did look at some of these other measurements and analyzed how certain traversal methods affected them.

5.2 Do we Have Optimal Results?

Another issue we discussed is that although FCA creates a lattice of all the possible groupings, and although we work around the problem of CSS inheritance, do we really come up with the best results? In fact, is there even a "best" result in the general case? Because we get different results with different traversals, and because FCA does not have any notion of any relation between objects, we concluded that the results may not be optimal when applying FCA alone, even if an optimal solution existed. However, the data shows that the optimization technique does significantly reduce various measurable metrics, and something can be said about that. Just as optimizations used in compilers do not provide the best imaginable result, they still provide very significant and useful results. This conclusion also leads to future research to provide better and better solutions.

5.3 Non-exact Duplicates

A point of improvement we could make is to recognize and identify non-exact duplicates. Our program only sees the property-value pairs simply as a pair of strings, which causes the two declarations `color: black`, `color: #000000`, and `color: #000` to be unequal and thus their rules not be considered for being grouped together by that declaration when they should be. Similar to this problem are properties which really specify several properties. For example, `border: 1px solid black` should be broken up into the three declarations `border-width: 1px; border-style: solid; border-color: black`. Because of these issues, results may not be as optimal as they could be.

5.4 Specificity

Because the optimization technique potentially re-groups and re-orders the CSS rules, this can obviously alter the specificity[18] of certain rules. However, different orders of specificity should stay intact; for example, if a style sheet relies on IDs to have priority over classes and classes to have higher priority than regular tags, then these priorities will remain in the optimized style sheet. Due to this, CSS run through this optimizer should not heavily rely on fine-grained specificity.

An example of this problem is exemplified in Figure 8. In this example, the third rule would be eliminated by our program which would result in the element `<a id='b'>` to have color white instead of black like it should. Most CSS we have seen does not rely on specificity very heavily in this way, so we decided that this was a perfectly fine trade-off.

Figure 8 An example of specificity problems

```
a   { color: black; } /* Rule 1 */
 #b { color: white; } /* Rule 2 */
a#b { color: black; } /* Rule 3 */
```

6 Evaluation

Table 1 shows the results of several CSS files run through our FCA program. It compares several different files' original file size with the two traversals we found to work the best most of the time: the top-down and bottom-up traversals with object-count and attribute-count affinities respectively, as discussed earlier. We compare the number of characters (which is also a measure of file size), the total number of rules, the total number of simple selectors, and the total number of declarations. We also compare these quantities as percentages of the original file. Formatting was consistent across all files both before and after the optimization and does not affect the results except for the possibility of making them slightly more or less dramatic across the board.

Table 1: Comparing Results to Original Files

CSS File		**Characters**	**Rules**	**Selectors**	**Declarations**
numbers.css	Original	523	10	10	22
	Top Down	223	5	22	5
	Change	-57%	-50%	+120%	-77%
	Bottom Up	238	4	6	10
	Change	-54%	-60%	-40%	-55%
mod.css	Original	6781	112	176	208
	Top Down	6600	98	297	114
	Change	-3%	-13%	+69%	-45%
	Bottom Up	5842	75	125	180
	Change	-14%	-33%	-29%	-13%
test.css	Original	4178	66	69	149
	Top Down	3675	56	106	94
	Change	-12%	-15%	+54%	-37%
	Bottom Up	3359	37	44	128
	Change	-20%	-44%	-36%	-14%
test2.css	Original	2318	32	38	96
	Top Down	2210	36	83	58
	Change	-5%	+13%	+118%	-40%
	Bottom Up	2914	23	27	114
	Change	+26%	-28%	-29%	+19%
large_test.css	Original	169968	1898	2137	4250
	Top Down	187466	1342	4356	1587
	Change	+10%	-29%	+104%	-63%
	Bottom Up	133004	1175	1278	3883
	Change	-22%	-38%	-40%	-9%

Based on these numbers, we feel that under most circumstances, applying FCA as an optimization technique for CSS produces a significantly smaller CSS file. However, the traversal method that is best is highly dependent on the style sheet itself and a web developer should always look at the results for particular style sheet to pick the best technique. Our data suggests that the bottom up traversal may be slightly better than the top down traversal most of the time, but by no means all the time.

7 Related Work

Many CSS optimizers, sometimes called "minifiers," exist on the internet, but unfortunately, by definition, they only remove whitespace, line-breaks, comments, and other syntactic "fluff"[19, 20]. While this method is optimizing can be efficient, easy, and useful, it really is a well-known and quite trivial task. The main benefit of using these minifiers is that they run very quickly and are easy to use. Our program can take a non-trivial amount of time and the solution sometimes needs to be analyzed to ensure that it is in fact better and that specificity has not been broken. A large benefit of our work, however, is that it is mutually exclusive with minification, and so can be run independently or in sequence with minification. Other than these minifiers, we were not able to find much related work in the field of developing automation tools for optimizing CSS style sheets.

8 Conclusions

While applying FCA may not provide the best solution in all cases, a best solution may not even always exist and if so, this technique still produces solutions which are typically significantly closer to optimum than the original input. This proves by example that optimizing CSS structurally is possible which is known to improve web site performance and ultimately the end-user experience. We have also seen that we can map systems which may not exactly fit perfectly with FCA into forms which can be accepted by the algorithm and have it produce very useful and worthwhile solutions. Further research can and should be done to improve upon our methods and produce better and better solutions.

References

[1] Dave Raggett, Arnaud Le Hors, and Ian Jacobs, "Html 4.01 specification: Style sheets", http://www.w3.org/TR/REC-html40/present/styles.html, 1999.

[2] Amit Singhal and Matt Cutts, "Using site speed in web search ranking", http://googlewebmastercentral.blogspot.com/2010/04/using-site-speed-in-web-search-ranking.html, 2010.

[3] "How to improve seo by reducing your page load time", http://www.blogseoexpert.com/blog-seo-tips-tricks/how-the-page-load-time-of-your-blog-or-website-impacts-seo/, 2009.

[4] "15 quick ways to shrink page load times", http://webjackalope.com/fast-page-load-time/, 2008.

[5] Wikipedia, "Bounce rate", http://en.wikipedia.org/wiki/Bounce_rate, 2010.

[6] Steve Souders, "Performance impact of css selectors", http://www.stevesouders.com/blog/2009/03/10/performance-impact-of-css-selectors/, 2009.

[7] Bert Bos, "Cascading style sheets", http://www.w3.org/Style/CSS/, 2010.

[8] "Information technology - open systems interconnection - basic reference model: The basic model", *ITU-T Recommendation X.200*, Jul 1994.

[9] "Multipurpose web publishing using html, xml, and css", *Communications of the ACM*, vol. 42, no. 10, pp. 95–101, 1999.

[10] "Separate content and structure from presentation", *Web Content Accessibility Guidelines 2.0*, pp. 14–15, 2001.

[11] Jim Thatcher, Paul Bohman, Michael Burks, Shawn Lawton Henry, Bob Regan, Sarah Swierenga, Mark D. Urban, and Cynthia D. Waddell, *Constructing Accessible Web Sites*, chapter 9, pp. 231–266, Glasshaus, 2002.

[12] Bernhard Ganter and Rudolf Wille, *Formal Concept Analysis: Mathematical Foundations*, Springer-Verlag, 1998.

[13] BA Davey and HA Priestley, *Introduction to lattices and order*, chapter 3, University of Cambridge, 2002.

[14] David Eppstein, “Concept lattice”, `http://en.wikipedia.org/wiki/File:Concept_lattice.svg`, 2006.

[15] Robert Grimm, “Rats! - an easily extensible parser generator”, `http://cs.nyu.edu/rgrimm/xtc/rats.html`, 2009.

[16] Bert Bos, Tantek Çelik, Ian Hickson, and Håkon Wium Lie, “Grammar of css 2.1”, `http://www.w3.org/TR/CSS2/grammar.html`, 2009.

[17] Christian Lindig, “colibri-java: Formal concept analysis implemented in java”, `http://code.google.com/p/colibri-java/`, 2007.

[18] Bert Bos, Tantek Çelik, Ian Hickson, and Håkon Wium Lie, “Assigning property values, cascading, and inheritance”, `http://www.w3.org/TR/CSS2/cascade.html`, 2010.

[19] “Minify javascript and css”, `http://developer.yahoo.com/performance/rules.html#minify`, 2010.

[20] Steve Clay, “minify”, `http://code.google.com/p/minify/`, 2009.