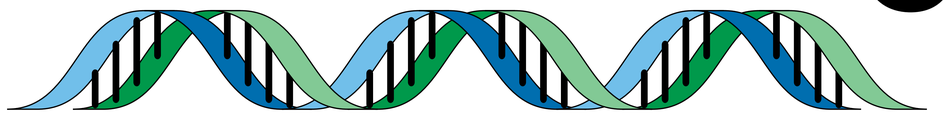
CSS and Genetic Algorithms

Evolving Optimal CSS with Python

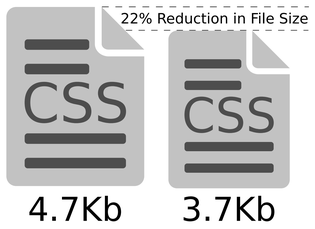
Ryan Scott Brown



Save bandwidth, beat standard minification techniques by 10%*

Why bother?

Web developers everywhere have heard of minifying Javascript and CSS to decrease asset sizes and make web pages load faster. With Python we can go beyond minification using genetic algorithms to compress CSS. Using the tinycss library we can apply a bipartite algorithm to any stylesheet to beat standard minification techniques by 10%.



How does this apply to CSS?

CSS is just a series of rules, and we can look at each rule as if it were a biclique. That's a fancy way of saying that we can assume both sets of rules on the right are valid.

We must assume that the rule with the highest specificity is the one that the user wants to win.

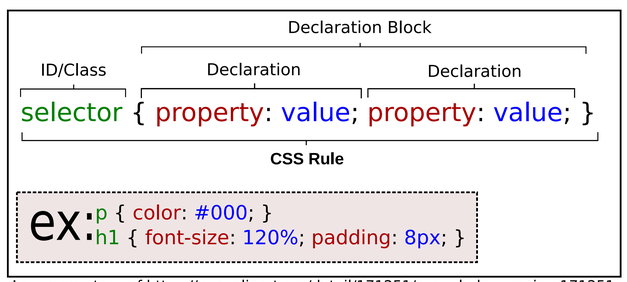


Image courtesy of https://openclipart.org/detail/171251/css-rule-by-xsapien-171251

Links:
http://github.com/ryansb/genetic-css

* As compared to removing all whitespace from the CSS file, a common practice in web development.

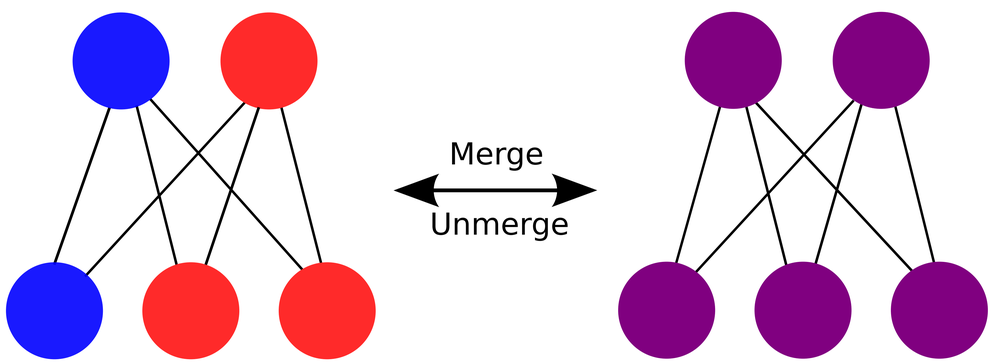
What is a genetic algoritm?

1) Randomly mutate members of the population

2) Test against a fitness function

3) Select and copy the fittest member

4) Repeat




```
div {
    color: blue;
    margin: 5em;
}

p {
    color: blue;
    margin: 5em;
}
```

```
div , p {
margin: 5em;
color: blue;
}
```

What is a bipartite graph?

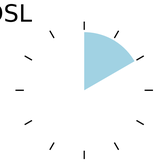
A bipartite graph is a graph that can be split such that there are no connections within a partition.

We can assume this because every CSS rule can stand on its own and be operated on as if it were its own element in the bipartite graph.

How much faster is it?

1-2 seconds faster on most home DSL connections

0.5 seconds faster on higher-end home DSL connections



How long does it take?

It takes as little as 60 seconds to optimize a 4-5KB stylesheet.

For longer stylesheets it can take 10 to 15 minutes, but can be made faster by decreasing the number of generations before considering a covering to be "stable".

The more generations required before "stability" the more optimal the result.

How do you know it's done?

The CSS must work the same way but use fewer bits. To do this, we need to cover the same graph (rule set) but find the best way to do it.

Because compressing a graph in this way is NP-hard, we need to figure out an end condition. If there is no improvement for 10 generations we say "good enough" and end the program.