Minisaw

Adam Geller

June 4, 2018

1 Introduction

Cassius is a formal specification of CSS webpage layout. It has taken years of effort and 1000 lines of code. It is difficult to debug large webpages because many boxes influence the bugs. We want to take a large webpage, and produce a smaller webpage with a similar error. Minisaw automatically minimizes a webpage where Cassius exhibits a bug. This automates a tedious and repetitive task.

The goal of Minisaw is to produce the smallest subset of the original HTML (i.e., obtained by deleting parts of the HTML) where Cassius exhibits a bug. Minisaw may produce a webpage that exhibits a different Cassius bug than the original webpage did.

2 Algorithm Description

Minisaw works by running Cassius in a loop. It repeatedly shrinks a page until the page is minimized (see section 3 for more on shrinking pages). Specifically, it shrinks the page, checks that the new page stil exhibits a bug, and recurses on the new page if it does. If Minisaw cannot figure out a way to shrink the page, it considers the page to be minimized.

Minisaw backtracks by trying to shrink the page in a different way if the new page does not exhibit a bug. Minisaw currently only implements "shallow" backtracking. That is, Minisaw only backtracks when no bug is exhibited, it does not backtrack out of a local minimum.

3 Shrinking Pages

This section describes how Minisaw shrinks webpages.

Minisaw shrinks a page by choosing an HTML element and removing it from the webpage. When Cassius exhibits a bug (its layout constraints are inconsistent with Firefox's actual rendering), it usually produces information about the failure. In this case, Minisaw chooses the HTML element to remove using a choice algorithm (described in section 3.1).

Otherwise, Minisaw could use exhaustive search: trying to remove each HTML element in turn. However, it does not yet implement exhaustive search because we have not found it to be necessary.

3.1 Choice Algorithm

Minisaw's choice algorithm attempts to select an HTML element not marked unremovable, preferably unrelated the bug, and with as many descendants as possible.

HTML elements are marked unremovable when no bug is exhibited by Cassius after the element was removed. Minisaw will never try to remove such HTML elements. This guarentees we will never make the same mistake twice.

When Cassius's constraints are inconsistent with Firefox's rendering, Cassius usually produces an UNSAT core. The UNSAT core identifies all of the boxes (a box being a container or content object involved in the rendered webpage) whose properties where somehow involved in the buggy execution. Every HTML element that is associated with a box identified by the UNSAT core, or has a descendant associated with such a box, is considered to be related to the bug.

Minisaw keeps track of an element tree, which describes the HTML elements and the hierarchy between them. The choice algorithm finds up to two candidates for every HTML element associated with a box identified in the UNSAT core. Based on a starting HTML element, Minisaw searches down the tree from the root towards the starting element until it finds an ancestor of the starting element with a sibling. If the sibling is marked unremovable, it is disregard and the search continued. If the sibling is related to the bug, Minisaw will continue searching for an unrelated element, but will add the sibling to the candidate list if it cannot find an unrelated element in its search.

Once Minisaw has computed the candidate list, it returns the candidate with the most descendants. If Minisaw cannot find any candidates to remove, the webpage is considered minimized.

4 Implementation

Minisaw is implemented using Racket, Python, and JavaScript. The Python code is responsible for getting Firefox's rendering of the shrunken webpage, running Cassius, and then running Minisaw. The UNSAT core is cached to avoid re-running Cassius after backtracking. The choice algorithm is written in Racket so it can reuse some of Cassius' code to handle the element tree and unsat core. Finally, the actual shrinking of the webpage is handled via DOM manipulation in JavaScript. Elements to be removed are identified outside of Cassius via their HTML tag and index with the HTML document (e.g., the 5th div), as are backtracked elements.

5 Results

We ran Minisaw on eight failing test webpages from a test suite we use to test Cassius. The number of boxes after minimization were consistently much smaller than the number before, as shown in fig. 1. The runtimes of Cassius on the minimized versions were all under 10 seconds, whereas the fastest uniminimized webpage took more than 20 seconds, as shown in fig. 2.

Both sets of results are statistically significant ($p < 0.01$). However, given the small test size, it is possible there are other webpages where the minimizations are no better than the initial.

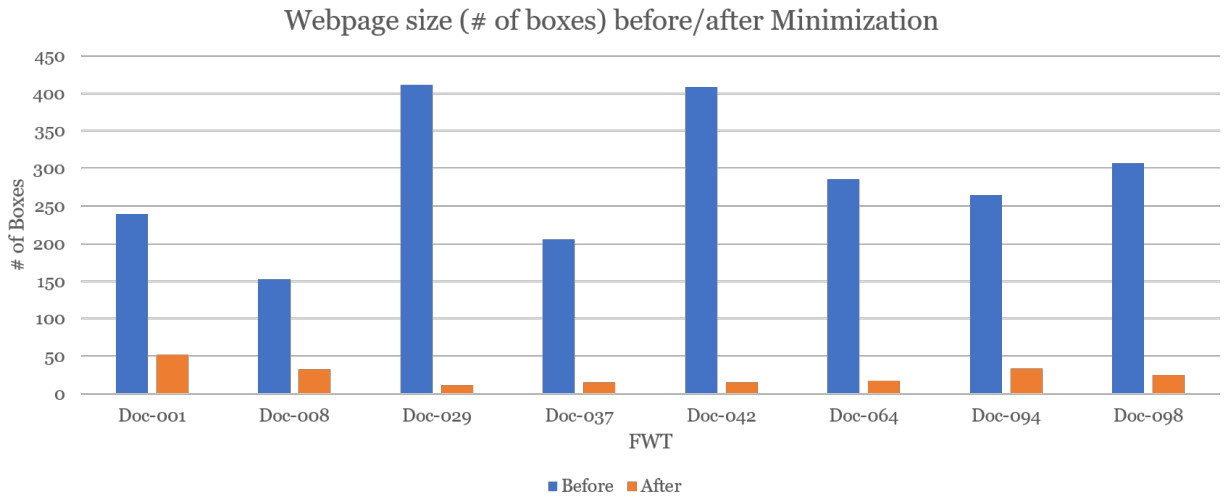

Figure 1: Sizes of test webpages before and after minimization.

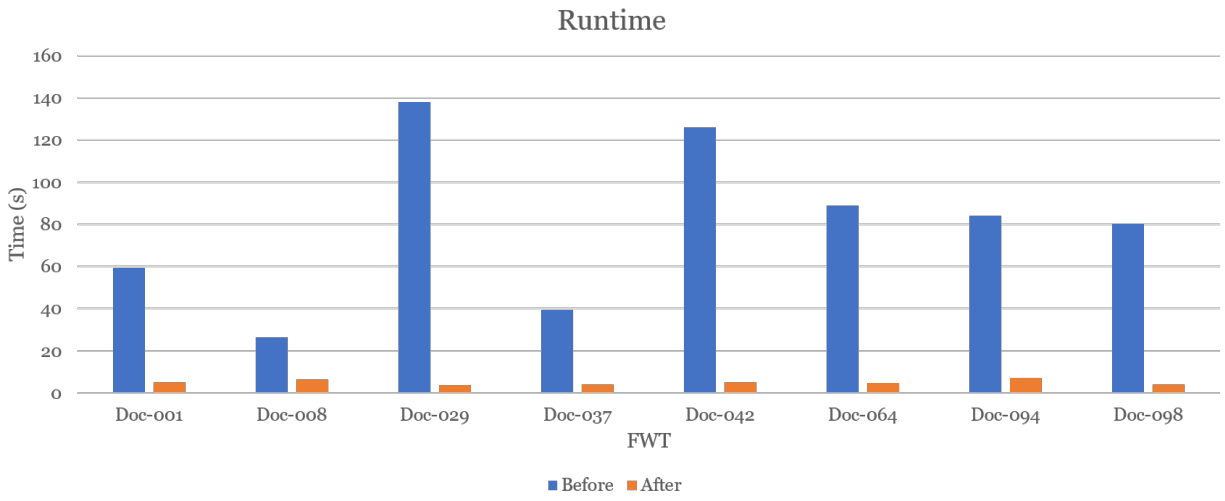

Figure 2: Cassius runtime of test webpages before and after minimization.

6 Future Work

We are currently working on adding support for minimizing VizAssert assertions to Minisaw. VizAssert is a tool that uses Cassius' specification of webpage layout to allow webpage developers to check that assertions about a webpage's layout hold across a range of browser dimensions and default font sizes. If an assertion does not hold, VizAssert provides a concrete counterexample where the assertion fails.

Adding support for minimizing VizAssert assertions to Minisaw would allow webpage developers to quickly diagnose the cause of an error, determine if it is a false positive, and test potential fixes. While we can reuse much of the codebase, we will have to make changes to accommodate the dissimilar goal of minimizing assertions. When minimizing a Cassius bug, the goal is to produce the smallest webpage that exhibits some bug. We do not care about the bug itself. Conversely, when minimizing an assertion counterexample, preserving the exact counterexample semantics is the top priority. Ensuring the counterexample remains the same is difficult because a change in the element tree could cause many changes in the box tree.

One benefit of minimizing on assertion counterexamples is that we have more information then when minimizing a Cassius bug. Specifically, instead of an UNSAT core, we have a concrete counterexample detailing exactly which boxes are at fault, and exactly what dimensions and default font size the counterexample occurs at. We can assert on these concrete values instead of over a range to achieve some speed-up while running VizAssert, but also to ensure that the bug is the same.

7 Conclusion

Minisaw is a tool that minimizes websites where Cassius exhibits a bug. It does this using a choice algorithm based on information from Cassius and knowledge of the element tree and Cassius' UNSAT core. Minisaw assists development by reducing the time it takes to find bugs within Cassius and eliminating the need for a developer to do a tedious and repetitive task by hand.