

Code Analysis for Test Case Reduction

Weber State University

CS 6780

Parker Adams, Heberto Rodriguez

<https://github.com/parkeradams/Project-6720>

Abstract

The goal of this project is to shed light on how code analysis can affect test case reduction, specifically with delta debugging. Given static code analysis on a method or test case, how can we use that information to improve delta debugging?

Background

Delta Debugging, specifically the ddmin algorithm, is an automatic debugging method that utilizes a binary search to reduce a delta (statements, lines of code, or characters) to its minimum passing code. In our case, the minimum code is the least amount of code possible to continue to reproduce an error.

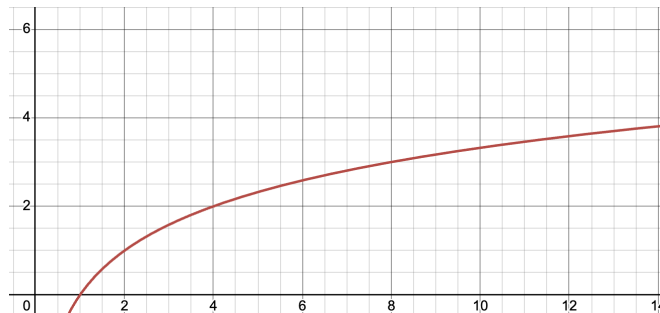
Static analysis involves examining the syntax, potential errors, spelling errors, and overall structure of the code. It is the analysis of a program before it runs.

Experiment

1. Static analysis

A test case with 16 lines of code was used to test the iterations and wall time of the ddmin algorithm. The reduced test case found by ddmin was lines 0 and 14, and it took 15 iterations of ddmin and 227 seconds to execute. Using static analysis, we found that three lines were already being tested in a previous passing test case, therefore they could be removed. Without the lines, the test reduction took 15 iterations and 228 seconds. Removing an additional line of code before ddmin reduced the algorithm to 13 iterations and about 30 seconds.

Mathematically this result makes sense because DDMIN is a binary search algorithm and has a time complexity of $O(\log_2 n)$. A strong argument could be made that using a static analysis to reduce a test case before delta debugging does not save much time and may not be worth the hassle.







Ddmin $O(\log_2 n)$ time complexity

2. Dynamic analysis

In our dynamic analysis, we used Jacoco to evaluate the test coverage of the isValid method in the UrlValidator class. The report provided insights into several key aspects of test coverage:

Apache Commons Validator > org.apache.commons.validator.routines

org.apache.commons.validator.routines

Element	Missed Instructions	Cov.	Missed Branches	Cov.	Missed	Cxty	Missed	Lines	Missed	Methods	Missed	Classes
UrlValidator		96%		90%	10	66	9	129	1	19	0	1
isValid(String)		94%		86%	3	12	2	23	0			1

- **Missed Instructions (Cov.):**
 - **71, 94%:** Out of all instructions in the isValid method, 94% are covered by the tests, leaving 6% uncovered (71 missed instructions).
- **Missed Branches (Cov.):**
 - **19, 86%:** Out of all branches (decision points like if-else statements) in the isValid method, 86% are covered by the tests, leaving 14% uncovered (19 missed branches).
- **Missed Cxty (Cyclomatic Complexity):**
 - **3, 12:** This indicates that the isValid method has a cyclomatic complexity of 12, and 3 complexity points were not covered by the tests.
- **Missed Lines:**
 - **23, 0:** Out of the total lines in the isValid method, 23 are uncovered.
- **Missed Methods:** This shows how many methods within the UrlValidator class were not executed.

These metrics from Jacoco provide valuable insights into the effectiveness and thoroughness of the isValid method in the UrlValidator class. By further investigating the highlighted sections of the code, which Jacoco marks in red, yellow, and green, you can visually identify where the test cases have been effective (green), partially effective (yellow), or ineffective (red). This visual feedback helps in determining whether additional test cases need to be created or if some existing ones can be optimized or removed. Achieving 100% test coverage may require adding tests to cover uncovered areas, and in some cases, it might reveal parts of the code that are redundant or unnecessary, suggesting they could be refactored

or removed.

```
public boolean isValid(final String value) {
    if (value == null) {
        return false;
    }

    URI uri; // ensure value is a valid URI
    try {
        uri = new URI(value);
    } catch (final URISyntaxException e) {
        return false;
    }
    // OK, perform additional validation

    final String scheme = uri.getScheme();
    if (!isValidScheme(scheme)) {
        return false;
    }

    final String authority = uri.getRawAuthority();
    if ("file".equals(scheme) && GenericValidator.isBlankOrNull(authority)) { // Special case - file: allows an empty authority
        return true; // this is a local file - nothing more to do here
    }
    if ("file".equals(scheme) && authority != null && authority.contains(":")) {
        return false;
    }
    // Validate the authority
    if (!isValidAuthority(authority)) {
        return false;
    }

    if (!isValidPath(uri.getRawPath())) {
        return false;
    }

    if (!isValidQuery(uri.getRawQuery())) {
        return false;
    }

    if (!isValidFragment(uri.getRawFragment())) {
        return false;
    }

    return true;
}
```

Making a minor change to the code or removing tests that aren't covering the code can improve the percentage.

● isValid(String)	<div><div></div></div>	97%	<div><div></div></div>	90%	2	11	1	21	0	1
-------------------	------------------------	-----	------------------------	-----	---	----	---	----	---	---

Even though Jacoco is not typically used for test reduction it has its advantages in looking at the bigger picture of your code and the coverage your test has. This can help tremendously in knowing where to add or remove test cases. If test reduction is the goal then we suggest the ddmin algorithm is the best way to go for test reduction.

Future Work

Unless static analysis can pre-reduce the test case by a considerable amount, it doesn't have too much of an effect on the runtime and iterations of the ddmin algorithm. However, future work on the topic could include using a smaller delta, such as the characters, and using static analysis to verify that a program is valid before compilation. Assuming static analysis is faster than compiling the code, speed-ups could be achieved. The exploration of combining code coverage tools like Jacoco with advanced test case minimization techniques such as the ddmin algorithm could also be further investigated. This approach could help in not only optimizing the test suite for coverage but also reducing the complexity of individual test cases, leading to more efficient and manageable testing processes.