PPGI
UNIRIO

Local optimization of JavaScript code

FÁBIO FARZAT

MÁRCIO BARROS

GUILHERME TRAVASSOS

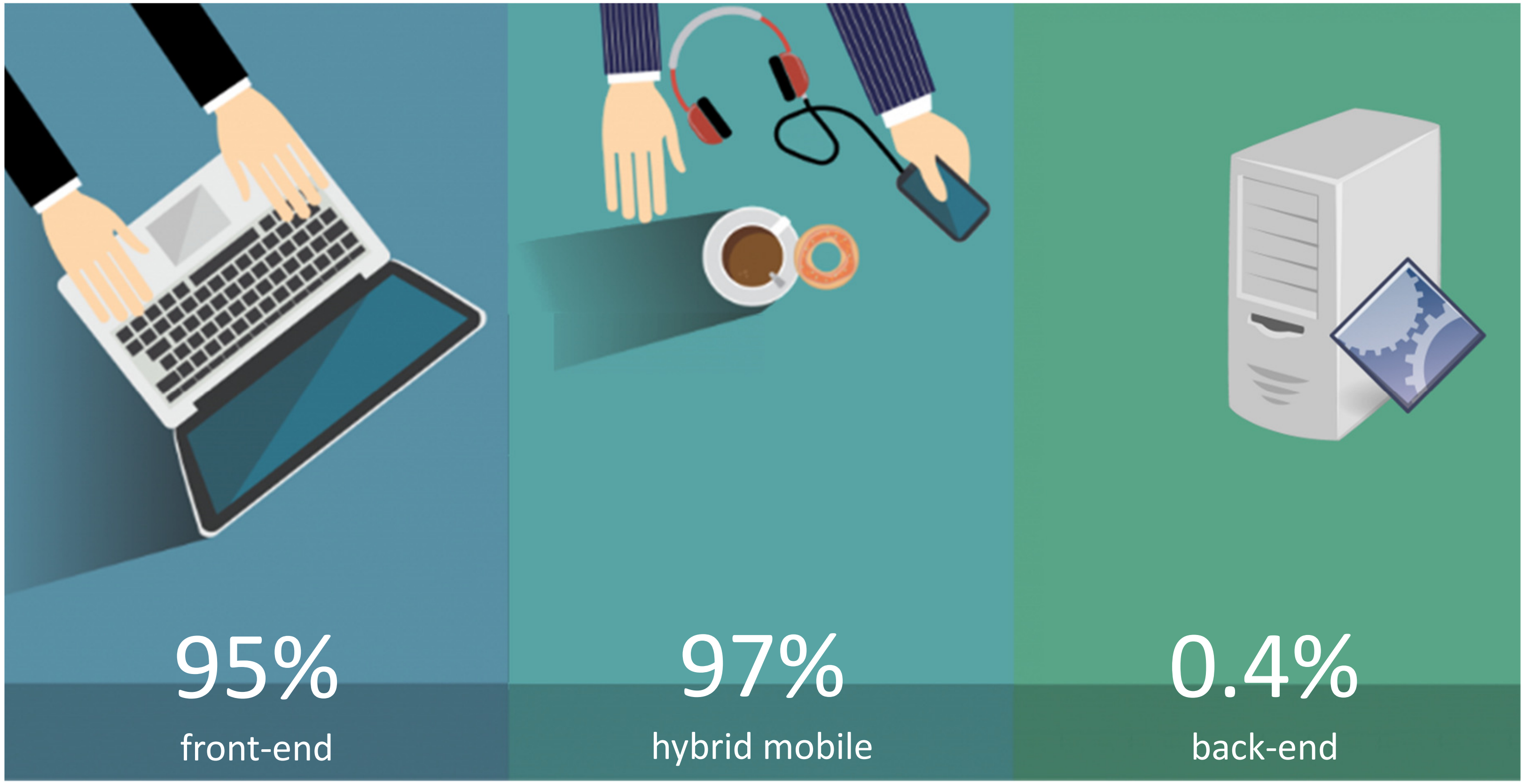
Emphasis

- Break things instead of repairing them
- Syntax tree level manipulation instead of source code lines
- Local search instead of genetic algorithms

Why should one care about JavaScript?

- The world seems to be committed to JavaScript
- The first version of the programming language was developed in a couple of weeks to allow “non-programmers” handle the structure of a Web page in a browser
- It was designed to provide a better interaction model between the front- and the back-end of Web apps
- Now, JavaScript is used everywhere …

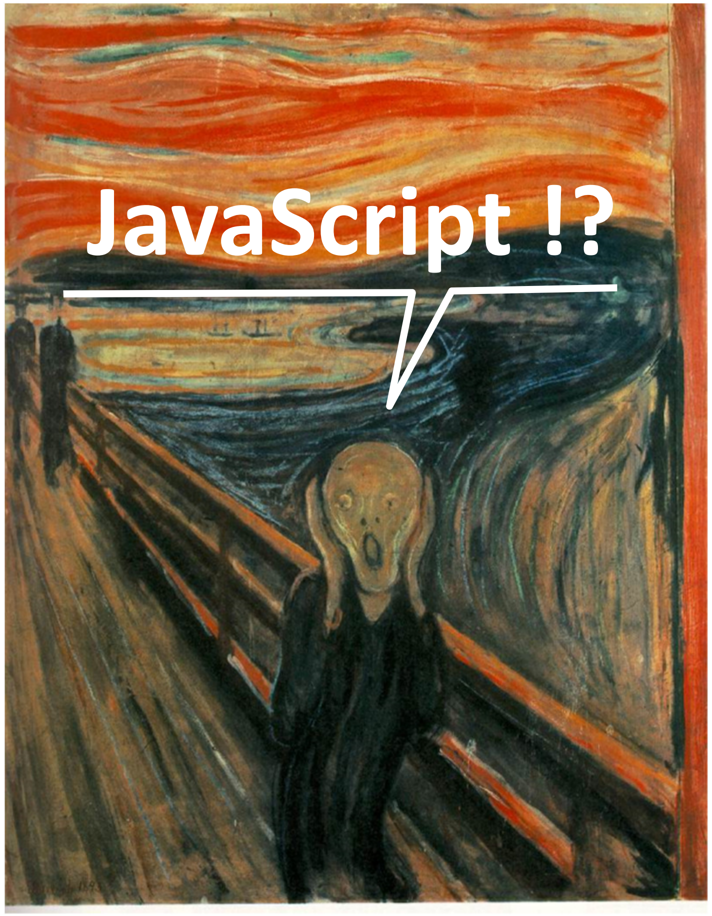
JavaScript is everywhere ...



https://goo.gl/kWbgsU https://ionicframework.com/survey/2017#trends https://www.similartech.com/technologies/nodejs

The dangers of JavaScript

- At the same time, a future that depends so much on JavaScript is worrisome
- JavaScript shows peculiar behavior if a developer goes beyond the bounds of "normal" programming



The dangers of JavaScript

Command Prompt - node

```
> var ts
undefined
> ts
undefined
> ts * 1
NaN
> ts |1
1
> 42.toFixed(2)
SyntaxError: Invalid or unexpected token
> 42 .toFixed(2)
'42.00'
```

https://www.youtube.com/watch?v=2pL28CcEijU
https://www.destroyallsoftware.com/talks/wat

Objectives

The main objective of our research is to find variants of a target JavaScript program which are smaller and functionally-equivalent to the target program.

Objectives

The main objective of our research is to find variants of a target JavaScript program which are **smaller** and functionally-equivalent to the target program.

Reducing the size of the source code (minified version) will reduce load and processing times.

Objectives

The main objective of our research is to find variants of a target JavaScript program which are smaller and **functionally-equivalent** to the target program.

Equivalence as attested by the test suite of the target program, which acts as our (limited) oracle.

Important notice

This is an ongoing work!

At the moment, we are interpreting the results collected from a second round of experiments.

But it all started with an opportunity …

Opportunity strikes!

- The university installs a supercomputer and needs someone to test it!

- Lobo Carneiro
  - Cluster-based supercomputer
  - 252 processing nodes
  - Each node has 24 cores running HT
  - 16 Tb of RAM memory
  - 720 Tb of disk

Opportunity strikes!

- We examined the fitness landscape for JavaScript source code improvement
- We executed a genetic algorithm and a random search over 13 target programs
  - Mutation operator that removes nodes from the AST
  - 5,000 fitness evaluations/round, 60 rounds for each program
- At top usage, we occupied 2,880 cores and 500 Gb of RAM

The selected JavaScript programs

Heavily-used JavaScript libraries

>= 90% statement coverage

Distinct sizes, from small to large

Researchers had some experience

Program	LOC	# Tests	% Cov	Usage	Version
Browserify	757	570	99%	1.9	14.3.0
Exectimer	195	37	91%	0.4	n/a
Express-ifttt	160	29	93%	0.2	n/a
Gulp-cccr	525	17	90%	0.1	n/a
jQuery	7,607	937	91%	4.8	3.2.0
Lodash	10,795	2,077	94%	33.8	3.10.1
Minimist	193	140	92%	25.8	1.2.0
Plivo-node	609	26	91%	10.6	n/a
Pug	400	240	98%	356	4.0.0
Tleaf	133	131	96%	0.2	n/a
Underscore	1,481	198	95%	8,710	1.8.3
UUID	209	21	91%	11,497	n/a
XML2JS	526	83	93%	3,783	0.4.16

Findings

- Surprisingly, random search outperformed genetic algorithms for all instances
  - GA failed to find improved versions in more than 50% of its runs for all programs
  - RD fails less frequently and found variants representing from 0.2% to 22% reduction!
- Patches are small and clustered in independent parts of the source code
  - The distance between patches is moderately and inversely correlated to program size
  - Patch size is strongly and inversely correlated to program size
  - The median is always smaller than the average for both measures (a few large values)
  - The best variants found by random search had many patches (37% rounds found 5+ patches)

Findings: an example

```
UUIDjs.getTimeFieldValues = function (time) {
  var ts = time - Date.UTC(1582, 9, 15);
  var hm = ts / 4294967296 * 10000 & 268435455;
  return {
    low: (ts & 268435455) * 10000 % 4294967296,
    mid: hm & 65535,
    hi: hm >>> 16,
    timestamp: ts
  };
};
```

Listing 1. function getTimeFieldValues (original version of the source code)

```
UUIDjs.getTimeFieldValues = function () {
  var ts;
  var hm;
  return {
    low: (ts & 268435455) * 10000 % 4294967296,
    mid: hm & 65535
  };
};
```

Listing 2. function getTimeFieldValues (optimized version of the source code)

UUID library

Findings: patch distribution

These findings imply that basic genetic algorithms are not effective for JavaScript source code reduction because the chances that recombination merges independent mutations are very small.

Findings: patch distribution

```
UUIDjs.getTimeFieldValues = function(time) {
  var ts = time - Date.UTC(1582, 9, 15);
  var hm;
  return {
    low: (ts & 268435455) * 10000 % 4294967296,
    mid: hm & 65535,
    hi: hm >>> 16,
    timestamp: ts
  };
};
```

Individual 1 (subjected to one mutation)

```
UUIDjs.getTimeFieldValues = function(time) {
  var ts;
  var hm = ts / 4294967296 * 10000 & 268435455;
  return {
    low: (ts & 268435455) * 10000 % 4294967296,
    mid: hm & 65535,
    hi: hm >>> 16,
    timestamp: ts
  };
};
```

Individual 2 (subjected to a second mutation)

What are the chances of a one-point crossover that keeps both building blocks?

Findings: patch distribution

```
UUIDjs.getTimeFieldValues = function(time) {
  var ts = time - Date.UTC(1582, 9, 15);
  var hm;
  return {
    low: (ts & 268435455) * 10000 % 4294967296,
    mid: hm & 65535,
    hi: hm >>> 16,
    timestamp: ts
  };
};
```

```
UUIDjs.getTimeFieldValues = function(time) {
  var ts;
  var hm;
  return {
    low: (ts & 268435455) * 10000 % 4294967296,
    mid: hm & 65535,
    hi: hm >>> 16,
    timestamp: ts
  };
};
```

```
UUIDjs.getTimeFieldValues = function(time) {
  var ts;
  var hm = ts / 4294967296 * 10000 & 268435455;
  return {
    low: (ts & 268435455) * 10000 % 4294967296,
    mid: hm & 65535,
    hi: hm >>> 16,
    timestamp: ts
  };
};
```

Rephrasing: what are the chances of selecting these cutting points?

They are inversely proportional to the square of the number of instructions in the target program.

So, what is the alternative?

- A systematic transversal of the search space (for instance, a local search) may find better results than random search
  - Local search behaves well if departing from a good solution (the human-written program)
  - Optimization is performed by removing nodes from the AST that do not contribute to the test cases
  - The key challenge is the size of the neighborhood for any given program

JADE LANGUAGE
Node Template Engine

4,794 chars
1,294 instructions

jQuery

86,202 chars
30,601 instructions

JavaScript

ECMA-262 Syntax Trees

Which of the 53 different nodes types are worth examining?

Binding Pattern
ArrayPattern
AssignmentPattern
BindingPattern
RestElement
ObjectPattern

Expression
ThisExpression
Identifier
Literal
ArrayExpression
SpreadElement
ObjectExpression
Property
FunctionExpression
ArrowFunctionExpression
ClassExpression
ClassBody
MethodDefinition
TaggedTemplateExpression
TemplateElement
TemplateLiteral
MemberExpression
Super
Meta-Property
NewExpression
CallExpression
UpdateExpression
UnaryExpression
BinaryExpression
LogicalExpression
ConditionalExpression
YieldExpression
AssignmentExpression
SequenceExpression

Statement
BlockStatement
BreakStatement
ContinueStatement
DebuggerStatement
DoWhileStatement
EmptyStatement
ExpressionStatement
ForStatement
ForInStatement
ForOfStatement
FunctionDeclaration
IfStatement
LabeledStatement
ReturnStatement
SwitchStatement
SwitchCase
ThrowStatement
TryStatement
CatchClause
VariableDeclaration
VariableDeclarator
WhileStatement
WithStatement

Imports
ImportDeclaration
ImportSpecifier
ImportDefaultSpecifier
ImportNamespaceSpecifier
ExportAllDeclaration
ExportDefaultDeclaration
ExportNamedDeclaration

Which nodes types are worth examining?

- We determined the topmost node types in the patches found by random search
- We determined the frequency with which node types appear in JavaScript programs
  - We have performed a study using ~34,000 JavaScript programs from the NPM repository
- We have calculated a ratio favoring high-frequency nodes that appear as topmost
  - Set a minimum threshold that limits which node types are examined by the local search

JavaScript AST

18 most worth node types for JavaScript source code size reduction

Binding Pattern

ArrayPattern
AssignmentPattern
BindingPattern
RestElement
ObjectPattern

Expression

ThisExpression
Identifier
Literal
ArrayExpression
SpreadElement
ObjectExpression
Property
FunctionExpression
ArrowFunctionExpression
ClassExpression
ClassBody
MethodDefinition
TaggedTemplateExpression
TemplateElement
TemplateLiteral
MemberExpression
Super
Meta-Property
NewExpression
CallExpression
UpdateExpression
UnaryExpression
BinaryExpression
LogicalExpression
ConditionalExpression
YieldExpression
AssignmentExpression
SequenceExpression

Statement

BlockStatement
BreakStatement
ContinueStatement
DebuggerStatement
DoWhileStatement
EmptyStatement
ExpressionStatement
ForStatement
ForInStatement
ForOfStatement
FunctionDeclaration
IfStatement
LabeledStatement
ReturnStatement
SwitchStatement
SwitchCase
ThrowStatement
TryStatement
CatchClause
VariableDeclaration
VariableDeclarator
WhileStatement
WithStatement

Imports

ImportDeclaration
ImportSpecifier
ImportDefaultSpecifier
ImportNamespaceSpecifier
ExportAllDeclaration
ExportDefaultDeclaration
ExportNamedDeclaration

Which nodes types are worth examining?

- We examine all occurrences of each node type in a First-Ascent HC fashion
  - For small instances, we use all node types and reduce the search space to 89%
  - For larger ones, we discard MemberExpression and Identifier node types, reducing the space to 34%
  - This allows navigating the space several times in a reasonable time frame, even for large instances

Preliminary results: achieved reduction

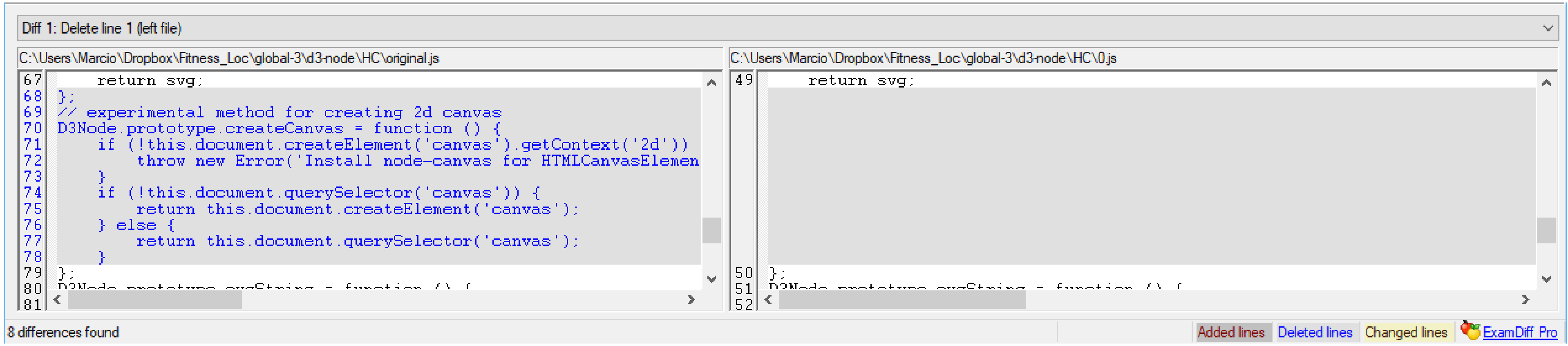
Program	RD	FAHC
browserify	0.19	25.39
exectimer	2.06	26.76
jquery	0.19	79.89
lodash	0.33	6.23
minimist	0.14	2.68
plivo-node	0.58	33.24
pug	3.16	39.17
tleaf	3.81	67.07
underscore	0.30	10.10
uuid	1.05	23.60
xml2js	0.14	-2.78

- A huge difference from former results
- Some results are within an expected range
- Other results … well, not so much!
- Some results are even curious …

But they all pass all test cases! And we have at least 90% coverage!

Is this any different to dead code removal?

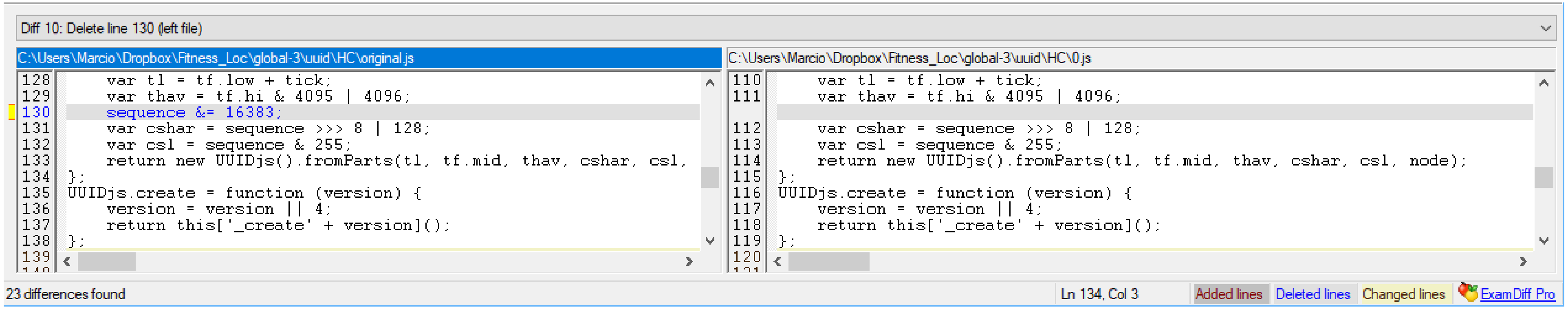
In some cases, not really.



A function from the d3-node library which is not exercised by test cases and was removed by the optimizer.

Is this any different to dead code removal?

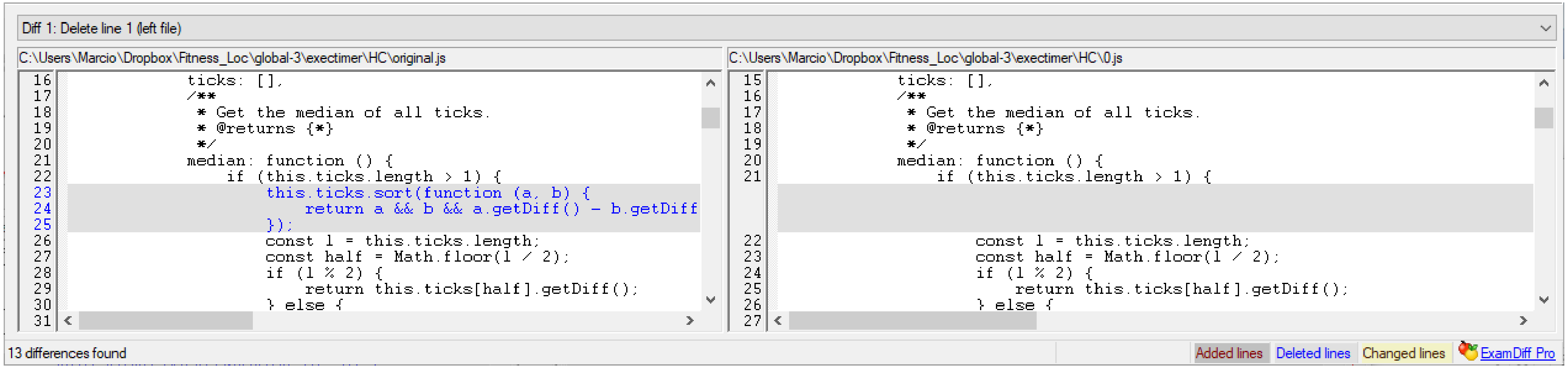
But in other cases, yes it does.



Bitwise operation from the uuid library that had no effect on test cases, despite being covered by the test suite.

Can we help to improve tests or code review?

There seems to be an opportunity to co-evolve test cases and the code.



"Summertime testing" in the exectimer library. All test cases use sorted data.

Can we help to improve tests or code review?

"A program does what the programmer commands, not necessarily what the programmer wants."

By showing what it can destroy, optimization can help developers put their assumptions into solid test suites ... and close the gap.

What is next?

- We are compiling the numbers, strengthening the arguments, and hope to have a complete version of a paper with our results soon.

1

Local optimization of JavaScript code

Fábio de A. Farzat, Márcio de O. Barros and Guilherme H. Travassos

Abstract—*Context*: JavaScript is now one of the most used languages on the Internet applications development. The number of libraries available and the complexity of the applications using these libraries brings a concern about performance. *Objective*: To apply optimization techniques (that had already shown positive results in other languages) to reduce the size of JavaScript programs and indirectly improve the performance of these programs. *Method*: Run controlled experiments involving genetic algorithms and random search to understand the solution landscape of the JavaScript source code reduction problem from the perspective of search-based techniques. *Results*: We observed that genetic algorithms were outperformed by random search due to the distribution and size of the patches that were found to reduce the programs while maintaining their functionality. Therefore, we suggest using a systematic search procedure based on Hill Climbing to find variants of a target program containing these patches. *Conclusion*: Our experiments show that a local search procedure can outperform both random search and genetic algorithms in JavaScript source code reduction.

Index Terms—SBSE, Genetic Improvement, JavaScript, Hill Climbing

1 Introduction

SINCE its debut in 1995, JavaScript has become the most used scripting language on the client-side of Web applications [7]. The need for a more efficient interaction model between the client and server sides of Internet systems drove the quick acceptance of JavaScript by software developers. On the other hand, the ability to include parts of the application logic on the client-side resulted in large programs written in a language that was designed for scripting and provided limited support for large-scale programming [8]. These programs required standardization and shared common features, leading to the creation of reusable JavaScript libraries, such as jQuery, AngularJS and React. These libraries must be transferred to the client-side before the application starts and transfer time is directly related to the size of their source code. Large libraries may cause undesired delays if the application is served over lines with limited bandwidth or to mobile devices with limited processing capabilities.

to JavaScript code.

After applying genetic algorithms and random search to improve 13 JavaScript libraries in a set of exploratory experiments designed to draw a rough picture of the JavaScript source code improvement solution landscape, we observed that typical improvements involved several small changes clustered in independent parts of the source code instead of changes that might be easily produced by recombination. Therefore, we hypothesized that a local search algorithm might perform better than a population-based approach, such as a genetic algorithm, on improving JavaScript code. Local search algorithms are usually faster and less demanding on computing resources than population-based algorithms. Thus, their usage in automated code improvement might allow coping with larger code bases and require less human intervention to select parts of the source code to be optimized and subsets of the test suite to drive the optimization.

Thank you!