This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TSE.2019.2928293, IEEE Transactions on Software Engineering

Evolving JavaScript code to reduce load time

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

Abstract—JavaScript is one of the most used programming languages for front-end development of Web applications. The increase in complexity of front-end features brings concerns about performance, especially the load and execution time of JavaScript code. In this paper, we propose an evolutionary program improvement technique to reduce the size of JavaScript programs and, therefore, the time required to load and execute them in Web applications. To guide the development of this technique, we performed an experimental study to characterize the patches applied to JavaScript programs to reduce their size while keeping the functionality required to pass all test cases in their test suites. We applied this technique to 19 JavaScript programs varying from 92 to 15,602 LOC and observed reductions from 0.2% to 73.8% of the original code, as well as a relationship between the quality of a program's test suite and the ability to reduce the size of its source code.

Index Terms—JavaScript, source code improvement, genetic programming, local search

1 Introduction

Since its debut in 1995, JavaScript has become the most used scripting language in the client-side of Web applications [5]. The need for an efficient interaction model between the client and server sides of Web applications drove the quick acceptance of JavaScript by software developers. However, the ability to include parts of the application logic in the client-side resulted in large programs written in a language that was designed for scripting and provides limited support for large-scale programming [6]. These programs require standardization, and they share 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 every execution of the application and the total load and execution time depends on the size of their source code. Large libraries may cause undesired delays if the application is served over lines with tight bandwidth or to mobile devices with limited processing capabilities.

Automated source code improvement is a set of techniques that automatically manipulate software that is written by human beings to improve it according to a set of quality criteria while preserving its functionality [3]. We applied genetic and local search algorithms to reduce the size (and, consequently, the load and execution time) of eleven JavaScript programs varying from 133 to 10,795 LOC in an exploratory experiment designed to reveal the characteristics of the solution landscape for JavaScript source code size reduction. This experiment showed that reduced variants of the target programs resulted from several small changes clustered in independent parts of the source code that might not be easily produced through classic crossover operators. We, therefore, hypothesized that a local search algorithm might be more effective to reduce the size of JavaScript source code than a population-based approach (such as genetic programming), besides being faster and less demanding on computing resources.

- *Fábio Farzat and Guilherme H. Travassos are with Systems Engineering and Computer Science Program, COPPE/UFRJ, Cx Postal 68501, Cidade Universitária, Rio de Janeiro, RJ, Brazil.*
- *Márcio Barros is with the Post-Graduate Information Systems Department, PPGI/UNIRIO, Av. Pasteur 458, Urca, Rio de Janeiro, RJ, Brazil*

In this paper, we propose a technique to reduce the size of JavaScript source code. We report the results of an experiment that applied this technique to 19 heavily-used JavaScript programs with sizes ranging from 92 to 15,602 LOC. We observed reductions varying from 0.2% to 73.8% of the code, moderately and negatively correlated to program size, test suite size, and statement coverage. Results also show that adding stochastic behavior to the local search used in our approach does not increase its performance significantly.

We observed a relationship between the quality of a program's test suite and the amount and validity of the changes produced by optimization: significant changes were made in the target programs, most of them disruptive to their expected behavior, due to the way they were coded or their tests were designed. For instance, statements uncovered by test cases were removed by the search and programs with limited test suites may be subjected to overspecialization.

We posit that automated code improvement techniques can help to drive the improvement of a JavaScript program and its test suite, serving as a compass for software engineers to find defects in the code or lack of coverage in the test suite. They can also be used to tailor a program's implementation to a particular use that depends only on the part of the program's behavior (for instance, in the front-end of Web and mobile applications). The contributions of this research are:

- A technique to reduce the size of JavaScript programs and, consequently, their load and execution time;
- A description of the JavaScript source code reduction problem's solution landscape;
- A comparison between genetic programming and local search to reduce the size of JavaScript programs;
- A quantitative analysis of the potential reductions in JavaScript programs and a qualitative discussion of the most frequent changes performed by the optimizer.

Besides this introduction, Section 2 provides background information on JavaScript and the implications of improving source code written in this language. Section 3 describes the strategy we followed to build the neighborhood operator for the proposed local search code reduction technique, which is

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TSE.2019.2928293, IEEE Transactions on Software Engineering

presented in Section 4. Section 5 describes the design, execution, and analysis of an experiment that evaluated our search algorithm. Section 6 shows conclusions and future work.

2 Background

2.1 JavaScript

JavaScript is an interpreted programming language that is available on multiple platforms. It uses object prototypes to represent classes and inheritance [6]. Objects are comprised of a set of properties, which are represented as string mappings to values (as in a hashed dictionary). Features can be added and removed from these objects during execution, and their names can be dynamically calculated. Furthermore, variable values are freely converted from one type to another with few exceptions for which automatic conversion does not apply.

JavaScript was initially developed as a programming language to write client-side scripts for Web applications requiring part of their logic to reside on the client side for improved communication with the server-side and to produce quicker responses to the user. It usually involved updating part of the information shown on a web page without refreshing the whole page. Recently, JavaScript programs came to be used on mobile devices and in the server-side of Web applications. On mobile devices, software written in JavaScript uses platform-specific browsers to change their appearance, allows interaction with device-specific features and simulates an application that is written and compiled in the platform's native language [8]. On web servers, a JavaScript runtime (such as NodeJS) runs as an HTTP server and allows serving pages and running scripts that access databases and other server-side resources to provide information for the client-side.

As a significant part of the software written in JavaScript still executes in the client-side of Web applications and needs to be transferred from the server to the client for execution, an essential aspect of JavaScript programs is their physical size on disk. The larger the code, the longer the transfer time will be. In this sense, techniques that reduce the size of JavaScript source code, such as minification [7], were created. These techniques apply lexical transformations to the program, reducing the names of variables, methods, and objects, and removing excess of white spaces and comments. Minification preserves functionality while reducing the size of a JavaScript program, but it is limited to the aforementioned lexical changes. Google Closure Compiler[1] adds static analysis features to minification, applying code compression transformations (e.g., isolating string literals), *inlining* code, and removing dead code. Nevertheless, the tool is limited to a set of preconceived changes and does not attempt to understand the source code to determine whether it could be improved by other means. It is the realm of source code improvement.

2.2 Source code improvement

In 1992, John Koza used genetic algorithms to automatically generate programs for accomplishing specific tasks, such as solving mathematical expressions. The author baptized the method as Genetic Programming (GP) [4]. In GP, programs are represented as syntax trees containing *function* or *terminal* nodes. Functions can be arithmetic, Boolean, conditional, iteration-related, or domain-specific functions for the problem in question. Terminals are variables or constants.

GP searches for a syntax tree that improves a feature of the program (its size, for instance) while preserving its ability to fulfill the task for which it was created. GP handles a set of program variants (a *population*) over a sequence of iterations (or *generations*). At each iteration, genetic operators are applied to the current population to build the next generation. The *crossover* operator takes a pair of trees (*parents*) and transplants part of the first into the second and vice-versa, aiming to carry on to the next generation syntax trees having good properties of both parents. The *mutation* operation randomly affects a given syntax tree by adding or removing one or more of its nodes. Therefore, GP is a specialization of genetic algorithms for manipulating computer programs.

Genetic Improvement (GI) typically uses meta-heuristics, such as genetic programming, to improve existing software (a *target program*) according to a set of quality criteria. Petke et al. [1] prepared an extensive survey describing several applications of GI to Software Engineering. Much of the present work focus on improving the target programs execution time [1]. However, GI can be applied to improve other non-functional requirements. According to LeGoues et al. [9], research conducted using GI is divided into four areas: defects correction, runtime improvement, migration and transplantation, and dynamic adaptation. Our research focuses on the second group, aiming to reduce the transfer and execution time of a JavaScript program by using a local search algorithm to find a smaller and functionally equivalent variant of this program.

GI has been previously used to remove unused or under-utilized parts of software systems. Landsborough et al. [23] trimmed the test suite of software to remove test cases for unwanted features and executed a genetic algorithm to remove instructions (from the binary code) that were unnecessary to fulfill the remaining test cases. They report 6% to 17% size reductions for small programs. Yeboah-Antwi and Baudry [24] present a GI platform that evolves Java programs in runtime, removing, merging, and duplicating the byte-code within methods and shortening loops. The authors tested the approach on synthetic software with 98% of unused methods and found out a 40% reduction in the number of methods with 0.02% of false-positive removals.

Langdon et al. [26] used GI to improve the implementation of software libraries while leaving their API unchanged. They tested the approach on an image processing library amounting to 319 LOC and improved its runtime by 13.2% while accepting a small processing error (one in a million pixels wrong in the worst case). Distinctly, we have tested our approach on more extensive target programs, but we are limited to *perfect oracles* (in the form of test cases) that do not require an approximation function.

Petke et al. [31] applied GI in a system called MiniSAT, an open source project written in C++ to solve Boolean satisfaction problems. According to the authors, finding improvements to MiniSAT through GI was a significant challenge, since human programmers evolved its code over the years. The researchers wanted to see if it was possible to improve the execution time of the best human solution. The optimization process was applied only to the main class of the system (*solver.cc*), and the observed improvement was 1% in the best case, attained by removing assertions from the source code.

1. https://developers.google.com/closure/compiler

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TSE.2019.2928293, IEEE Transactions on Software Engineering

Automated program repair is also a domain on which GI is frequently applied. Weimer et al. [10] introduce GenProg, a program repair approach that uses GP to evolve a program to pass failing test cases by removing, duplicating, and replacing potential faulty lines of code with other lines of code already in the program. While GP typically represents variants of a target program as syntax trees during their evolution, GenProg introduces the *patch representation* [10] [20]: a variant of a target program is represented as a sequence of changes to its original syntax tree, instead of the tree itself. The patch representation reduces the memory required to represent variants of a target program and allows the crossover operator to focus on combining changes, instead of unmodified parts of the program.

2.3 Local search algorithms

Heuristic search algorithms can be broadly divided into two groups: *population-based* and *point-based* algorithms.

Population-based algorithms favor exploration instead of exploitation, investing computer-processing power to evaluate the quality and evolve a set of individuals dispersed throughout the space of potential solutions for the problem at hand. Genetic algorithms, as well as GP and GI, are typical examples of population-based algorithms, which are useful for problems whose fitness functions can be quickly calculated and where recombination tends to lead to improved individuals. However, these characteristics are not easily recognizable *a priori* and result from problem characteristics, solution representation, and the use of customized operators.

Local search is a class of point-based algorithms that use heuristics to traverse the neighborhood of a given individual systematically [11] searching for a better neighbor, if one can be found. A neighbor from a given individual A results from the application of a simple operation upon A, such as changing an individual characteristic. The neighborhood of an individual A is defined by the set of distinct neighbors that can be generated by applying the selected operation upon A. Once a better neighbor is found, the search procedure is repeated to examine its neighborhood. Local search algorithms favor exploitation instead of exploration, concentrating the investigation in the neighboring region of a given individual taken as an initial solution for the problem at hand and neglecting the remaining parts of the search space.

Hill Climbing is a local search algorithm frequently used in Software Engineering [11]. First-ascent Hill Climbing (FAHC) is a greedy algorithm in which a neighbor presenting better fitness than the current individual always replaces the latter as the search focus throughout the optimization process. The policy for selecting the next neighbor to be examined is configurable, ranging from a completely random selection to the traversal of a list of selected neighbors. As it always proceeds with the search from the first neighbor that outperforms the current solution, the algorithm may be stuck to local optimum whose neighborhood is worse than the present individual. To overcome this limitation, FAHC is frequently equipped with different strategies to jump to a new region of the search space once it finds a local optimum.

Local search algorithms are computationally simple methods whose execution cost is usually a fraction of a population-based algorithm's cost. Due to favoring exploitation, these algorithms tend to be efficient and effective if they start from a good solution, that is one that is close to the optimum. We expect this to be the case for source code improvement, meaning that the human-written program presented for optimization is already a good program in the light of the objective pursued by the search algorithm. Thus, instead of exploring the search space at large, the optimization algorithm should commit more resources to investigate the already large target program's neighborhood.

Local search has already been used in source code improvement. In the first work that applied source code improvement to JavaScript programs, Cody-Kenny et al. [27] used two mutation operators as part of a greedy search to improve the load time of web pages. The operators removed calls to methods contained in the web page or its imported files and optimized loops written in JavaScript. The authors used page load time, the number of browser events, and the depth of the event tree as performance measures and found that their algorithm could improve all these measures in a custom-made web page. Our approach follows the trend of improving JavaScript code, but it is not restricted to call statements and loop instructions. We also report on experiments based on actively used, real-world JavaScript libraries.

Harman et al. [3] improve C++ code using a GP whose recombination and mutation operators are constrained by specific BNF notation that avoids changing the control flow structure of the source code while allowing changes in expressions and non-control flow statements. Afterward, a local search is applied to reduce the changes proposed by the GP to the minimum set that enables the resulting variant to show the observed improvements while passing the test cases. While local search plays a major role in our proposed technique, we borrow the idea of selecting parts of a program's syntax to change during program evolution.

Brownlee et al. [28] and White [29] present GIN, a Java toolbox to implement GI for Java programs. GIN started with local search algorithms, though the toolbox can be extended to include other search procedures. GIN is integrated with profiling, testing, and test case generation tools to accelerate the usage of GI in the Java ecosystem. An et al. [30] present PyGGI, a GI framework developed for Python. PyGGI is designed to support multiple search algorithms (including local search), different solution representations and operators (such as "line of code" and syntax tree based mutation). In synergy with our approach, GIN and PyGGI contribute in bringing source code optimization to a different context than C and C++, on which such techniques were often applied.

3 Characterizing the solution landscape for source code improvement in JavaScript

The design decisions leading to the source code improvement technique described in this paper were made according to observations collected after running an exploratory experiment using a GP algorithm and a local search to reduce the size of JavaScript programs. This experiment was designed to draw a rough picture of the JavaScript solution landscape of source code improvement. It was performed on a supercomputer installed at COPPE/UFRJ, Brazil. Lobo Carneiro is a cluster-based supercomputer comprised of 252 processing nodes, each having 24 cores that can run 48 threads in parallel due to

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TSE.2019.2928293, IEEE Transactions on Software Engineering

HyperThreading. The nodes can address 16 Tb of RAM and 720 Tb of disk space. At full capacity, Lobo Carneiro can perform more than 226 trillion mathematical operations per second. Such computing power was welcome because running the optimization and (thousands of times) the test suites of the target programs demanded fast processing units for parallelization and a tremendous amount of memory.

The Stochastic Hill Climbing (SHC) algorithm used here [18], [19] is a FAHC that randomly picks the next solution to be examined from the set of all potential neighbors of the current solution. The starting solution for the search is the abstract syntax tree (AST) representation of a target program. SHC randomly selects a node to be removed from the AST. If the resulting program variant passes all test cases designed for the target program, it replaces the current solution, and the optimization continues from its AST. If the program variant fails to pass all test cases, the removed node is restored to the AST, and another node is randomly selected for removal. This procedure was repeated 5,000 times for each target program. SHC was selected because it is an unbiased local search algorithm and has the advantage of being embarrassingly parallel.

The GP was configured with a population of 100 individuals, each representing the complete AST of a target program variant. The initial population was created by applying a single mutation to the target program. The population was evolved over 50 generations, leading to 5,000 test suite evaluations upon program variants. A single-point crossover operator (affecting two individuals selected by the tournament with 100% probability) and a mutation operator that randomly selects and removes a node from an individual's AST (with 75% probability) drive the evolutionary process. Crossover is applied to the whole AST instead of a representation of the changes already made to a particular individual.

Both algorithms were executed over eleven heavily-used JavaScript programs whose sizes range from small to large and whose test suites guarantee at least 90% statement coverage. From the set of programs attending to these restrictions, the choices for the exploratory experiment were selected by convenience, some of them due to the researchers' practical usage experience. Such experience allowed researchers to play the role of users and assess the effectiveness of the changes proposed by different algorithms. The selected programs are listed below, and Table 1 shows their characteristics.

- **Browserify** adds NodeJS methods in the browser, allowing to write multi-platform JavaScript code;
- **Exectimer** tracks execution time of a code snippet or function with a resolution of up to nanoseconds;
- **jQuery** helps DOM manipulation and simplifies JavaScript scripts in the browser;
- **Lodash** adds utility functions and complements the behavior of functions from the Underscore library;
- **Minimist** handles command line arguments passed to JavaScript programs;
- **Plivo-node** allows creating voice and SMS applications based on the Plivo API;
- **Jade (Pug)** is a template engine to render client-side scripts for NodeJS and browsers;

Table 1
Characteristics of the programs used in the exploratory experiment. The first column shows the number of lines of code in each program; next, the number of test cases in their suite, the percentile of statements covered by the test suite, the number of times each program was downloaded in April 2018 (divided by 1,000) and the version selected for improvement (n/a in the lack of version numbers).

Program	LOC	# Tests	% Cov	Usage	Version
Browserify	755	570	99%	1,900	14.3.0
Exectimer	195	37	91%	0.4	n/a
jQuery	7,414	937	91%	48,000	3.2.0
Lodash	10,795	2,077	94%	33,800	3.10.1
Minimist	193	140	99%	25.8	1.2.0
Plivo-node	928	26	91%	10.6	n/a
Pug	317	240	98%	356	4.0.0
Tleaf	282	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

- **Tleaf** automates the generation of test cases for AngularJS controllers;
- **Underscore** adds functional programming helpers to JavaScript without extending built-in objects;
- **UUID** generates and parses Global Unique Identifiers (GUID) [2];
- **XML2JS** converts strings in the XML format to JavaScript objects and vice-versa;

Sixty independent optimization rounds were performed for each target program and algorithm. At top usage, 2,880 processing nodes and 500 Gb RAM of the supercomputer were reserved for these tasks. Next, we outline the significant findings of the exploratory experiment. Detailed data produced by this experiment and others reported here can be found at https://github.com/unirio/js-size-optimization.

3.1 Comparing local search to genetic programming

Table 2 presents the average (μ), standard deviation (σ), and median values (Med) of the results produced by the GP and the local search while traversing the solution space of the JavaScript source code improvement problem in search of smaller variants for each target program passing all test cases. Results are shown as percentile reductions upon the target program minified size (using UglifyJS to *minify* the code), measured in characters.

The size of programs can be measured in many units, including lines of code, the number of instructions, and the number of characters. JavaScript programs are usually bundled and minified before their deployment to production. Minification removes excess of white spaces and line breaks and, therefore, the number of lines of code is not a reasonable size measure for minified programs. On the other hand, the number of instructions in a program is only partially related to its size in bytes stored in the server and transferred through the Web, as different instructions may require a distinct number of bytes to be expressed. The number of characters is a more fine-grained measure and is also strongly related to the number of bytes occupied by the source code. We, therefore, measure the size of JavaScript programs as the number of characters in the minified version of their source code.

The mean results for the local search in all target programs are better than the mean results produced by GP. Eight out

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TSE.2019.2928293, IEEE Transactions on Software Engineering

Table 2
Comparing the results produced by genetic programming (GP) and the stochastic hill climber (SHC). Results are presented as a percentile of the number of characters reduced in the minified version of the best solution found by each algorithm on each round of optimization, in respect to the number of characters in the target program.

	GP		SHC	
Program	$\mu \pm \sigma$	**Med**	$\mu \pm \sigma$	**Med**
Browserify	0.00% ± 0.03%	0.00%	0.19% ± 0.42%	0.00%
Exectimer	0.00% ± 1.01%	0.00%	1.48% ± 2.08%	0.51%
jQuery	0.00% ± 0.00%	0.00%	0.26% ± 0.54%	0.11%
Lodash	0.00% ± 0.01%	0.00%	0.14% ± 0.21%	0.06%
Minimist	0.00% ± 0.04%	0.00%	0.12% ± 0.26%	0.00%
Plivo-node	0.02% ± 0.06%	0.00%	0.60% ± 0.48%	0.51%
Pug	0.09% ± 0.33%	0.00%	1.59% ± 1.80%	0.96%
Tleaf	0.01% ± 0.26%	0.00%	3.02% ± 5.58%	1.11%
Underscore	0.04% ± 0.22%	0.00%	0.15% ± 0.24%	0.09%
UUID	0.17% ± 0.91%	0.00%	0.90% ± 1.53%	0.30%
Xml2JS	0.01% ± 0.09%	0.00%	0.15% ± 0.34%	0.00%

of eleven medians for SHC are greater than zero, while all GP medians are equal to zero. Those results show that while GP eventually succeeds in finding an improved variant, more than 50% of its runs fail in this attempt. Such failures seem irrespective of program size, as the three cases for which SHC failed most of its attempts are related to small programs (*Browserify*, *minimist* and *xml2js*). The large values for standard deviation also show that any independent execution of the local search cannot be trusted to find a good solution.

To understand the poor performance of the GP, in the next section we examine the distribution of the patches produced by the local search.

3.2 Spatial distribution of patches

The variants produced by each round of the local search were compared to the target program. This comparison produced a sequence of patches proposed by the algorithm to reduce the size of each target program. These patches were represented both as textual *diffs* (such as those produced by the file comparison algorithm of a version control system [17]) and AST *diffs*, represented as JSON objects.

Table 3 shows the average and median distance between two consecutive patches found by SHC, measured in percentiles of program size in lines of code. Next, it shows the average and median size of these patches in the same unit. The last column shows the mean of the size of all patches found in a given optimization round. For the sake of an example, the average distance between consecutive patches in Lodash is 2.21% of the source code size (or 238 LOC), the mean size of its patches is 0.05% (or 5 LOC), and the mean size of all patches found by an optimization round is 1.37% (or 148 LOC).

Lines of code were used in this analysis because textual *diffs* work on a line-of-code basis. Furthermore, the files should be readable by developers to allow the investigation of the kinds of changes performed by the local search. A line of code is part of a patch if one or more of its instructions were removed by the local search in a given optimization round. If instructions from consecutive lines of the source code were removed, these lines are considered parts of the same patch. Thus, patches have one or more lines of code.

Both patch distance and size show a negative correlation to program size (Spearman rank-order correlation): average distance between patches (r_s=-0.76) and median distance between patches (r_s=-0.65) present negative and moderate

Table 3
The distance between consecutive patches proposed by the local search and the size of these patches, both measured as percentiles of program size in LOC. LOC is used because this analysis relies on textual *diffs*.

	Patch distance		Patch size		
Program	**Average**	**Median**	**Average**	**Median**	**Sum**
Browserify	13.00%	7.95%	0.22%	0.13%	0.25%
Exectimer	21.40%	17.40%	1.28%	0.51%	1.81%
jQuery	14.10%	10.60%	0.05%	0.01%	0.27%
Lodash	2.21%	0.03%	0.05%	0.02%	1.37%
Minimist	22.20%	20.50%	0.58%	0.52%	0.53%
Plivo-node	6.84%	0.43%	0.21%	0.11%	1.73%
Pug	18.70%	12.30%	1.56%	0.32%	5.43%
Tleaf	12.60%	8.69%	0.94%	0.36%	4.85%
Underscore	15.20%	12.40%	0.08%	0.07%	0.33%
UUID	20.20%	12.00%	0.90%	0.48%	1.73%
XML2js	19.20%	14.80%	0.28%	0.19%	0.16%
All programs	10.10%	1.59%	0.29%	0.07%	2.16%

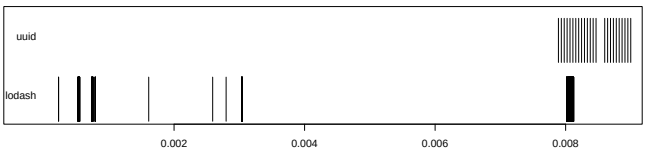

Figure 1. Distribution of patches over the extension of the source code for the UUID and Lodash programs. UUID has only two large patches in proportion to the size of the source code. Lodash has several smaller patches (in proportion) scattered throughout the code.

correlations, while average patch size (r_s=-0.85) and median patch size (r_s=-0.99) show negative and robust correlations to program size. The mean size of all patches found in an optimization round shows negative and moderate correlation to program size (r_s=-0.42) and 90% of all patches found for all programs have up to 9 LOC.

The larger the program, the smaller are the patches in proportion to program size and the closer they are to each other. The recurrent pattern of having an average distance larger than the related median distance reveals the existence of a few distant patches that drag the average upwards, while a larger number of patches in the same region of the source code keep the median low. The same pattern is observed for the patch size: the local search finds a few large patches, but most patches are small. Figure 1 shows a barcode chart depicting the distribution of patches over the source code for the best variant found by SHC for the *uuid* (small) and *Lodash* (large) programs. The horizontal span represents the whole source code, while the bars represent lines affected by a patch.

Table 4 shows the number of optimization rounds of the local search finding zero patches (that is, failed to find an improved version of the target program), one patch, two patches, three patches, four patches and five or more patches in the smallest variant that passes all test cases. The table shows that the number of patches tends to grow with program size. It is corroborated by a positive and moderate correlation ($r_s = 0.56$) between the number of rounds finding five or more patches and program size. 22.4% (148) rounds failed to find an improved target program variant. 77.6% (512) of the remaining 629 rounds found more than one patch in the target program variant.

Bringing together both analyses, we observe a large number of small patches clustered close together in pools distributed over the extent of the source code of large JavaScript

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TSE.2019.2928293, IEEE Transactions on Software Engineering

Table 4
Number of optimization rounds running the local search that found zero, one, two, three, four or five or more patches in the source code.

Program	0	1	2	3	4	5+
Browserify	37	9	5	1	3	5
Exectimer	13	23	13	8	3	0
jQuery	0	4	3	6	7	40
Lodash	0	4	2	3	2	49
Minimist	33	11	7	8	0	1
Plivo-node	2	3	7	6	10	32
Pug	4	9	13	8	6	20
Tleaf	5	6	7	4	5	33
Underscore	3	5	12	4	8	28
UUID	12	12	19	8	5	4
XML2js	39	16	0	2	2	1
Percentiles	22%	15%	13%	9%	8%	32%

programs. If we assume that the characteristics of the patches found by SHC can be found in an optimally reduced program variant (we will return to this hypothesis later), this finding explains why the GP using an AST-based solution representation and single-point crossover was not adequate for search-based source code reduction for JavaScript programs.

Table 4 shows that several mutations are required to find the reduced variants produced by SHC for a massive target program. The chances of randomly selecting a proper mutation in a given application of the mutation operator are small because such mutations represent only 2.16% of the source code, on average. While SHC concentrates the applications of the mutation operator in a single individual (the best-known solution), the chances of applying all correct mutations to the same individual in a GP are smaller because the algorithm spreads the number of mutations to be applied over all individuals in the population.

Thus, GP relies on the crossover to combine individuals with distinct mutations and produce a new individual containing all mutations from its parents. Since patches are small and clustered together, given two individuals with a valid patch on each, the probability of selecting a point to cut both AST and produce the combination of both patches in an offspring is proportional to the space between these patches. If large spans of source code separated consecutive patches, the crossover operator could pick any point between them in both AST and the patches would be recombined in the offspring. As most patches as separated by less than 1.59% of the source code, the chances of selecting an appropriate recombination point on both parents are also small.

To exemplify the discussion above, Listings 1 and 2 show the original and reduced variant versions of the *getTimeFieldValues* function of the *uuid* target program. The shortened version was generated through an optimization round running SHC. It removed the expressions used to calculate the *ts* and *hm* variables, as well as the parameter received by the function (formerly used in the discarded expressions), the *hi* and *timestamp* properties from the object returned by the function. Assume for the moment that both versions pass all test cases despite the changes in the calculation process (they do, actually, but we will return to that later).

The mutation operator in the GP can remove the expression that calculates *ts*. The same applies to the removal of the expression leading to the value of *hm*. The case against GP lies in the recombination of both patches: the algorithm can only produce the code of the reduced offspring by applying

```
1  UUIDjs.getTimeFieldValues = function (time) {
2    var ts = time - Date.UTC(1582, 9, 15);
3    var hm = ts / 4294967296 * 10000 & 268435455;
4    return {
5      low: (ts & 268435455)*10000 % 4294967296,
6      mid: hm & 65535,
7      hi: hm >>> 16,
8      timestamp: ts
9    };
10 };
```

Listing 1. Function *getTimeFieldValues* (original code)

```
1  UUIDjs.getTimeFieldValues = function () {
2    var ts;
3    var hm;
4    return {
5      low: (ts & 268435455)*10000 % 4294967296,
6      mid: hm & 65535
7    };
8  };
```

Listing 2. Function *getTimeFieldValues* (reduced variant)

both mutations in the same individual or by recombining two individuals affected by the mutations. As such mutations appear next to each other, the single-point crossover must be executed precisely between the AST nodes representing lines 2 and 3 to produce an offspring having both mutations.

Therefore, while the local search accumulates successful mutations on the same individual, a GP based on an AST solution representation and single-point crossover tends to evolve a population in which each carries a small part of the mutations. Where recombination fails to bring together the building blocks of a good solution, the accumulative nature of local searches tends to examine different mutations in the same solution parts.

Previous researchers have recognized the difficulty of applying crossover in source code improvement problems. Harman et al. [3], [13] use GP with a patch representation to improve the performance of C++ programs. Their crossover operator was configured to perform a given number of attempts to find a program variant that compiles and passes all test cases. If it fails to generate a valid individual after a predefined number of attempts, a mutation operation was executed in its place. Weimer et al. [10] and LeGoues et al. [20] also rely on a patch representation to describe variants of target programs evolved using GP. Such representation addresses the problem of selecting proper recombination points as only these patches are part of the solution. It also reduces the demands on memory to represent the solutions being manipulated by the algorithm. Fogel and Atmar argue that crossover is a generalization of several mutations performed at once, being itself a form of mutation. The authors subsequently claim that mutations result in better searches [15], [16].

An alternative to GP would be a systematic search traversing the target program AST and analyzing the effects of removing each node. This traverse might carry a solution that accumulates all positive patches. However, to look at all nodes in an extensive program requires a considerable budget of test suite evaluations. The third part of the exploratory experiment targets the discovery of types of nodes that produce the most efficient changes.

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TSE.2019.2928293, IEEE Transactions on Software Engineering

Table 5
JavaScript node types, their frequency in programs and the number of times they were selected as nodes to be removed by the local search.

Topmost node types	Count	Frequency	Rank
ExpressionStatement	1,818	5.1%	19.2
Identifier	827	36.6%	13.0
VariableDeclaration	1,119	1.5%	11.4
Literal	755	11.6%	8.5
ReturnStatement	581	1.2%	5.9
IfStatement	550	1.4%	5.6
Property	475	2.3%	4.9
CallExpression	381	5.7%	4.0
BinaryExpression	386	4.4%	4.0
FunctionDeclaration	386	0.3%	3.9
MemberExpression	297	11.8%	3.4
VariableDeclarator	213	2.5%	2.2
FunctionExpression	212	1.4%	2.2
ObjectExpression	177	0.8%	1.8
AssignmentExpression	162	3.4%	1.7
UnaryExpression	106	1.1%	1.1
ArrayExpression	98	0.8%	1.0

3.3 Examining node types

There exist 53 types of JavaScript instructions according to the ECMA-262 Standard[2]. These instructions fill the AST nodes of any JavaScript program, while the edges of these trees represent different relationships among nodes, such as the block of statements that should be executed in case a condition is *true* or the expression that calculates the value of a given property of an object.

The set of all node types represents 100% of the search space for a systematically traverse of the JavaScript source code improvement problem. However, both the frequency with which these instructions appear in the source code and their impact in source code improvement vary. While the search space can be wholly traversed within a reasonable budget of fitness evaluations for small programs, it may be impossible to test all alternatives for larger ones.

To find a balance between the completeness of full traverse and the processing effort required for such a task, we used the results from the local search to find the relative importance of each node type. While the last subsection used the textual *diff* between a variant found by the local search and its related target program; we now turn to a *diff* calculated from the AST representation, that is, the set of branches from the AST that were in the target program, but were removed from its variant.

Table 5 presents the topmost node types found in patches proposed by the local search. By topmost we mean that each mutation performed by the search removes one node (having a given node type), but the removal of such node may imply eliminating other nodes. For instance, if an *IfStatement* node is removed, the *BlockStatement* nodes associated with both results of its conditional expression are also removed, along with the *ConditionalExpression* node representing the condition itself. The *Count* column in Table 5 indicates the number of times a given node type was the topmost node of a patch proposed by the local search for all target programs.

The *Frequency* column shows the frequency with which the node types appear in JavaScript programs according to Rocha and Sobral [14], who reports on a study addressing the structure of JavaScript code. They analyzed 34,000 JavaScript programs from the NPM[3] repository and, among other information, reported the frequency of occurrence for every type of instruction in the ECMA-262 specification. If we are to select node types to examine while traversing the search space, those nodes which frequently appear in JavaScript programs will require more effort to evaluate, while the occurrences of rare node types may be examined more quickly.

Finally, the *Rank* column presents a ratio calculated according to Equation 1 for any given node type N_t. While this ratio has no physical meaning, it grows with *Count* and *Frequency*. Frequent node types in JavaScript programs and node types which frequently appeared as topmost in local search results will be assigned higher values in the *Rank* column. Lower rank values denote low-frequency node types, both in a general analysis of JavaScript programs and as topmost results in patches proposed by the local search. The table is limited to nodes having a *Rank* higher than 1.0.

$$Rank(N_t) = \frac{Count(N_t)}{100 \times (1 - Frequency(N_t))} \quad (1)$$

The node types presented in Table 5 represent 91.8% of the nodes of a randomly-picked JavaScript program. Therefore, the reduction in search space is not significant (8.2%). However, for large programs, the optimizer may not analyze nodes containing *Identifiers* and *Member expressions*, as these nodes usually happen as part of a larger expression. By removing these node types from the scope of a systematically traverse, the search space is reduced to 43.4% of the tree size. Such a reduction allows for a systematically traverse using less processing effort than the analysis of the whole search space. Such traverse is explained in the next section.

4 Local search code improvement for JavaScript

In this section, we present the Deterministic First Ascent Hill Climbing (DFAHC), an automated optimization process to reduce the size of JavaScript programs using local search as the core optimization algorithm.

We employ a First-Ascent Hill Climbing (FAHC) algorithm that uses the source code of the target program represented by its abstract syntax tree (AST) as an initial solution. The neighborhood-generation operator for the FAHC removes one node from the AST representing the current solution, subsequently eliminating all nodes attached to the selected one. Though the resulting solution may differ from the current solution by several nodes (the removal of one node may drop a branch from the AST containing several nodes), the resulting syntax tree is considered a neighboring variant because the search algorithm directly removed only one instruction. In this sense, the search procedure is a (1+1)EA that starts from a solution representing the current implementation of the target program instead of a random solution.

The key challenge to apply local search in source code improvement is the size of the neighborhood for any given program. While manipulating a program through its syntactic tree, a neighbor can be any variant of the original syntax tree in which a node is either added or removed. Given the diversity of nodes in JavaScript syntax trees and the average size of a JavaScript program, building the entire neighborhood

2. http://www.ecma-international.org/publications/standards/Ecma-262.htm

3. https://www.npmjs.com/

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TSE.2019.2928293, IEEE Transactions on Software Engineering

of a single program might exhaust the amount of time and computing power available for the optimization. To address the neighborhood size problem, we leveraged on the results of Section 3.3 to determine the kinds of nodes leading to the more extensive improvements when removed from the syntax trees of the target programs. Then, we used these nodes to build a neighborhood operator for a technique to improve source code based on a local search algorithm.

Given a budget of B fitness evaluations, the local search follows the order of node types presented in Table 5. For a given node type, it identifies all of its occurrences in the target program and tests, one-by-one, their removal from the source code. Every time a node is removed, a variant program is created and all test cases designed for the target program are executed upon this variant. If a variant passes all test cases, it is considered a valid variant, is selected as the current solution, and the search continues from its AST. Otherwise, the search process restores the removed branch to the former AST and moves to select another previous solution node to find a better neighbor. After exhausting all occurrences of the given node type, the next node type is selected. After exhausting all occurrences of all node types, the search restarts from the first node type. The search stops after a complete pass without improvement or after executing the test cases B times.

If the target program has less than $\frac{B}{2}$ instructions on its AST, the local search uses all node types presented in Table 5. For larger programs, the *Identifier* and *MemberExpression* node types are discarded to allow a broader traverse of the search space. These node types were discarded because they are very frequent in JavaScript programs and they also tend to be removed along with other node types, such as *Expressions*.

5 Evaluation of the proposed algorithm

In this section, we describe the experiment that was designed and executed to evaluate the effectiveness of the proposed local search approach for improving JavaScript programs.

We compared the DFAHC to the Stochastic Hill Climbing (SHC) used in Section 3, a Stochastic First Ascent Hill Climbing (SFAHC), and a GP using the patch representation (GPPR). SFAHC is similar to DFAHC except that it randomly selects the next node type to be examined for removal (considering the node types in Table 5) and its occurrence in the target program, while DFAHC strictly follows the order established in Table 5. GPPR is configured as the GP in Section 3 but describes the variants of the target program using a patch representation, instead of an AST-based representation. To allow a fair comparison, GPPR can remove only node types listed in Table 5.

SHC and SFAHC were executed ten times to handle variations in solution quality due to their stochastic nature. GPPR was executed 30 times, but for a subset of the target programs. The algorithms were executed in the same HPC environment described in Section 3, however with lower resource availability. Thus, we reduced the number of optimization rounds in comparison to Section 3. All algorithms were given a budget of 5,000 test suite evaluations.

5.1 Research questions

The experiment reported in the next subsections addresses the following research questions:

RQ1: How does DFAHC perform in comparison to the SHC algorithm regarding solution quality?

SHC outperformed GP in the exploratory studies and DFAHC uses information derived from the best solutions found by SHC. Therefore, we use SHC as a baseline for comparison for DFAHC and expect that the latter outperforms the former in finding improved variants for the selected target programs.

RQ2: How does DFAHC perform in comparison to the SFAHC algorithm regarding solution quality?

DFAHC establishes a sequence under which a selection of node types are analyzed. It may be that addressing such node types in a random order leads to better results than the order given in Table 5. RQ2 questions whether adding a random component to node type selection increases the performance of DFAHC.

RQ3: How do DFAHC and SFAHC perform for different program sizes, test suite coverage, and test suite size?

Here, we analyze the influence of target program characteristics on the performance of DFAHC. We expect the performance to decrease for larger programs with smaller test suites presenting poorer coverage.

RQ4: Do solutions found by DFAHC present the same characteristics found for solutions of SHC?

DFAHC is based on the assumption that smaller variants of JavaScript programs would differ from the original code by several patches clustered together in pools distributed over the extent of the source code (Section 3.2). RQ4 questions whether the smaller variants found by DFAHC present these properties.

RQ5: How does DFAHC perform in comparison to GPPR regarding solution quality?

While local search has outperformed GP in the exploratory studies reported in Section 3, it may not be true if the GP uses a different representation for variants of the target program. RQ5 compares DFAHC results to a GP using the patch representation proposed by GenProg [10] [20].

Using unit tests as oracle may lead to invalid results due to incomplete or low-quality test suites. Test coverage measures the percentile of source code lines that are exercised by the test suite. However, the quality of a test suite is much more complicated to determine as even running all lines of code does not guarantee that the tests check the expected results of their execution or these lines were executed in scenarios diverse enough to capture all variations of the expected functionality of the software.

Therefore, besides the quantitative analysis performed to answer the proposed research questions, we present a qualitative analysis that examines the code removed by the optimizer and discusses whether it is a solid improvement or one that was only possible due to a poor test suite. Then, the usage of the proposed approach to improved either the code, the test suite, or both are discussed in the light of the antagonism of test suites designed to detect if the functions of the software are preserved after some modification allowing the optimizer to suggest changes that break these features.

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TSE.2019.2928293, IEEE Transactions on Software Engineering

Table 6
Main characteristics of the subject programs added to the experiment. The first column shows the number of lines of code in each program; next, the number of test cases in their suite, the percentile of statements covered by the test suite, the number of times each program was downloaded in April 2018 (divided by 1,000) and the version selected for improvement (n/a in the lack of version numbers).

Program	LOC	# Tests	% Cov	Usage	Version
D3-node	92	89	91%	17	n/a
Decimal	1,595	22,509	96%	630	9.0.1
Esprima	6,615	1,352	100%	46,000	3.1.3
Express	1,137	1,865	100%	12,475	4.16.2
Mathjs	15,602	4,087	94%	479	3.20.2
Moment	9,978	2,514	96%	54,720	2.17.1
Node-semver	1,324	2,057	99%	46,665	5.5.0
UglifyJs2	4,098	172	91%	2	3.1.2

5.2 Programs under analysis

The eleven JavaScript programs used in the experiment described in Section 3 were selected as subjects for our experiments, along with eight new programs. Again we chose only heavily-used JavaScript libraries with test suites having at least 90% statement coverage. The new programs are listed below and their characteristics are shown in Table 6.

- **D3-Node** is a facade to access D3 library methods in the server side of Web applications;
- **Decimal** defines an arbitrary-precision decimal type and its operations for JavaScript programs;
- **Esprima** is a high performance, standard-compliant ECMAScript parser that is written in JavaScript;
- **Express** is a minimal and flexible Node.js framework that provides features for web and mobile apps;
- **Mathjs** is an extensive library of math operations for JavaScript programs;
- **Moment** parses, validates, manipulates, and displays date and time information;
- **Node-semver** a JavaScript implementation of the Semantic Versioning specification;
- **UglifyJs2** is a JavaScript parser, minifier, compressor and beautifier toolkit.

5.3 Quantitative Analysis

Table 7 summarizes the results found for the target programs using DFAHC, SHC, and SFAHC. Each percentage shows the reduction in the number of characters as a fraction of the program's size. The table shows the improvement found by DFAHC, as well as the median, mean ± standard deviation, and maximum improvements found by SFAHC and SHC.

All algorithms were able to find smaller and feasible variants for all target programs. The distribution of solution quality found by each algorithm is depicted in Figure 2. While the algorithms found consistent improvements for some target programs, substantial improvements were found for others. For instance, it is hardly believable that 80.1% of the source code of a JavaScript program as used as jQuery can be removed without affecting its functionality. However, such a variant passes all test cases developed for the program.

To answer RQ1, we observe that all solutions found by SHC represent smaller improvements than those found by DFAHC for 16 out of 19 target programs. On the other hand, all solutions found by SHC were superior to those found by DFAHC for a single program (*plivo-node*). The boxplots depicting solutions found by SHC cross the line representing the variant found by DFAHC twice (for *UUID* and *XML2js*). We employed a Wilcox test (alpha=5%) to determine whether the distribution for these programs was significantly different to the solutions yielded by DFAHC and found that the distribution is not significantly different for *UUID* (*p-value*=0.375), but is significantly different for *XML2js* (*p-value*=0.013). We conclude that SHC outperforms DFAHC for two programs, while the opposite occurs for 16 programs, with one draw. For the two programs on which it outperformed DFAHC, SHC removed nodes whose types are not limited by Table 5, including *BreakStatement*, *ThisExpression*, and *BlockStatement*.

To answer RQ2, we observe that all solutions found by SFAHC represent smaller improvements than those found by DFAHC for two programs (*Lodash* and *XML2js*). On the other hand, for none of the selected programs all results favor SFAHC. A perfect draw is observed for four programs (*D3-node*, *Minimist*, *Pug*, and *Tleaf*), both DFAHC and SFAHC finding the same results for all optimization rounds. Finally, the boxplots for SFAHC cross the line representing the solution found by DFACH for 13 programs, leading to significant differences for *Decimal.js*, *Express*, *jQuery*, *Esprima*, *Plivo-node*, and *Underscore* (Wilcox test at 5%), the first three scoring for DFAHC and the remaining ones for SFAHC. Therefore, we cannot find a definite difference in performance between the algorithms, DFAHC outperforming SFAHC for five programs, the reverse for three programs, and both algorithms finding similar results for eleven programs. While the randomness added to SFAHC has the cost of losing predictability, it may be useful for programs on which the deterministic results provided by DFAHC are limited.

To answer RQ3, we calculated the correlation (*Spearman* rank-order correlation) between the improvement observed in the variant produced by DFAHC and the target program size ($r_s = -0.21$), the test suite size in the number of test cases ($r_s = -0.02$), and the percentile of statement coverage of the test suite ($r_s = -0.23$). All correlations are small and negative, indicating that the larger the program or test suite coverage, the lower the improvement found by the algorithm. We also analyzed whether the difference between the median improvement found by SFAHC and the improvement produced by DFAHC are correlated to the same measures. We observed small correlations for all measures, the strongest correlation being with test suite coverage ($r_s = -0.36$), while program size and test suite size are less correlated to the difference between the algorithms ($r_s = -0.03$ and $r_s = -0.26$, respectively). Therefore, only test suite coverage and size seem to favor DFAHC over SFAHC.

To answer RQ4 we calculated the mean and median size of patches in variants produced by DFAHC for the 19 target programs under analysis, as well as the total size of these patches, the mean and median distance between them. The mean size of patches for all programs is 0.34% of the source code size (was 0.29% in the exploratory study) while the median size remains 0.07%. The mean size of all patches on an instance basis is much higher than in the former study (22% as compared to 2.16%), but this can be explained by the larger number of patches found by DFAHC in comparison to SHC.

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TSE.2019.2928293, IEEE Transactions on Software Engineering

Table 7
Size improvement for JavaScript target programs found by DFAHC and median, mean (with standard deviation), and maximum size improvements found by SFAHC and SHC over 10 optimization rounds.

		Median		Mean ± St. Dev.		Max	
Program	**DFAHC**	**SFAHC**	**SHC**	**SFAHC**	**SHC**	**SFAHC**	**SHC**
Browserify	25.40%	25.40%	14.20%	25.25% ± 0.32%	13.50% ± 2.86%	25.40%	16.28%
D3-node	26.00%	26.00%	24.21%	26.00% ± 0.00%	24.21% ± 0.00%	26.00%	24.21%
Decimal	32.51%	18.03%	6.98%	20.66% ± 6.28%	6.44% ± 2.05%	34.71%	9.22%
Esprima	3.88%	3.88%	0.33%	3.88% ± 0.01%	0.35% ± 0.11%	3.89%	0.64%
Exectimer	26.76%	26.76%	23.05%	26.37% ± 0.50%	22.85% ± 0.85%	26.76%	23.59%
Express	15.89%	15.86%	8.32%	15.70% ± 0.30%	8.56% ± 1.35%	15.89%	11.24%
jQuery	80.10%	80.05%	20.02%	80.05% ± 0.03%	20.79% ± 3.82%	80.13%	27.77%
Lodash	1.59%	1.22%	1.27%	1.20% ± 0.26%	1.21% ± 0.28%	1.48%	1.56%
Mathjs	27.04%	27.80%	19.04%	27.74% ± 0.78%	19.07% ± 1.27%	28.93%	21.00%
Minimist	2.78%	2.78%	1.84%	2.78% ± 0.00%	1.85% ± 0.08%	2.78%	1.96%
Moment	7.23%	7.23%	1.43%	7.12% ± 0.35%	1.55% ± 0.34%	7.25%	2.32%
Node-semver	13.46%	13.40%	4.12%	13.42% ± 0.34%	4.35% ± 0.65%	13.71%	5.61%
Plivo-node	15.34%	15.90%	55.62%	15.92% ± 0.38%	56.12% ± 2.96%	16.38%	61.27%
Pug	39.35%	39.35%	31.09%	39.35% ± 0.00%	30.61% ± 2.20%	39.35%	33.85%
Tleaf	65.80%	65.80%	64.02%	65.80% ± 0.00%	63.71% ± 0.81%	65.80%	64.05%
UglifyJS2	13.20%	13.88%	3.14%	12.23% ± 4.07%	3.06% ± 0.58%	13.94%	4.08%
Underscore	11.48%	11.48%	4.66%	11.48% ± 0.00%	4.63% ± 1.49%	11.48%	8.02%
UUID	20.82%	22.15%	21.57%	21.43% ± 2.49%	21.45% ± 1.86%	23.63%	24.35%
XML2js	4.12%	0.00%	9.75%	0.00% ± 4.40%	10.33% ± 5.79%	2.79%	17.61%

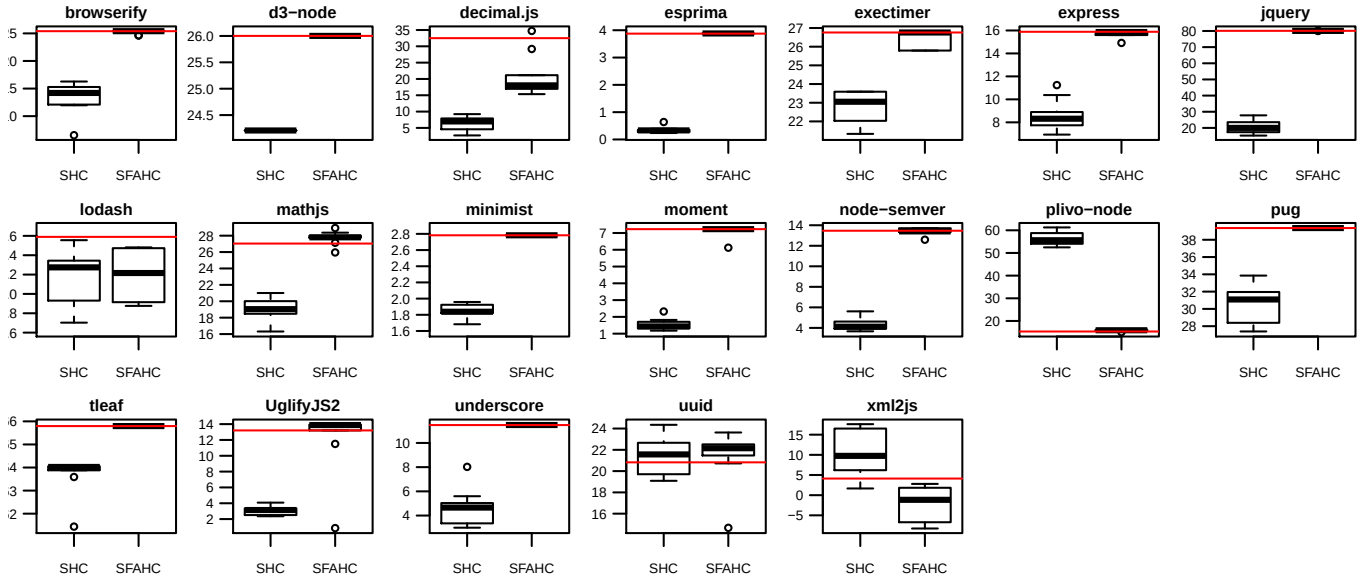

Figure 2. Distribution of the improvements found by different algorithms over ten optimization rounds. The target program is represented by zero in the y-axis, while the horizontal line represents the optmized variant found by DFAHC in the charts.

Mean distance between patches on an instance basis decreased to 1.14% of the source code size (was 10.10%), but the median of these distances also decreased to 0.51% (was 1.59%). Thus, the solutions found by DFAHC, which are closer to the optimum than solutions found by SHC, present the same characteristics found in the former study when compared to the target program: small patches that hang close together in the source code (small median distance) are grouped into pools that are far away from each other (large mean distance) in the code. It enforces the assumption that the characteristics of the solution space can be used to enhance the performance of JavaScript source code optimizers, such as DFAHC.

Table 8 presents the results of DFAHC and the 30 rounds of GPPR for the 12 smallest target programs. To answer RQ5, we observe that DFAHC outperforms the median and mean results yielded by GPPR for all programs, while GPPR's best results only outperform DFAHC for one instance (*Tleaf*). The cumulative nature of the local search outperforms the GP as the latter spreads the number of available mutations over its population. While the patch representation allows the crossover operator to contribute with the evolutionary process, the number of mutations required to attain the same improvement of the JavaScript source code suggests that accumulating them in a single individual evolved through a point process is better than spreading them over several individuals and recombine their features. Thus, at least for small programs we have initial evidence that even a GP based on a patch representation is outperformed by a local search whose search space is constrained to the most relevant node types while searching for reduced variants of JavaScript target programs. More extensive experiments using larger programs, more evaluations, and different configurations on the number

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TSE.2019.2928293, IEEE Transactions on Software Engineering

Table 8
Size improvement for JavaScript target programs found by DFAHC and median, mean (with standard deviation), and maximum size improvements found by GPPR over 30 optimization rounds.

Program	DFAHC	GPPR		
		Median	Mean ± St. Dev.	Max
Browserify	25.40%	0.00%	0.00% ± 0.00%	0.00%
D3-node	26.00%	17.45%	15.01% ± 5.81%	22.00%
Exectimer	26.76%	8.73%	8.60% ± 3.51%	13.17%
Express	15.89%	0.67%	0.87% ± 0.80%	2.79%
Minimist	2.78%	0.52%	0.59% ± 0.28%	1.27%
Node-semver	13.46%	0.00%	0.00% ± 0.00%	0.00%
Plivo-node	15.34%	4.41%	4.07% ± 1.60%	6.98%
Pug	39.35%	1.58%	1.88% ± 1.12%	5.73%
Tleaf	65.80%	44.28%	44.68% ± 11.21%	71.95%
Underscore	11.48%	1.63%	1.41% ± 0.60%	2.31%
UUID	20.82%	6.38%	5.81% ± 2.78%	10.57%
XML2js	4.12%	0.31%	0.63% ± 1.10%	4.49%

of generations and the size of the population are required to strengthen this conclusion.

As for the time required for optimization, it is difficult to ascertain the necessary amount of time because of the target programs vary in size and in the time needed to run their test cases. The HPC environment presents its challenges to measuring time due to the overhead needed for parallelization and the competition with other jobs for resources. If the algorithms were executed in a single processor of the HPC environment, this would amount to about 600 days of data processing. This considerable processing cost drove the analysis of a deterministic local search, which could be executed in a feasible time for medium-sized target programs in conventional desktop computers with sufficient memory.

5.4 Qualitative analysis

Next, we examine the changes that were found by the linear search proposed in Section 4 for three or more target programs. We are interested in finding patterns that may guide developers to better code or test cases, as well as giving directions to improve the optimizer itself.

Use strict: The "use strict" directive was introduced in JavaScript 1.8.5 to indicate that the code should be executed under certain constraint such as explicitly declaring all variables and using unique parameter names in functions. The optimizer removed the "use strict" directive, as its absence does not interfere with the execution of the test cases.

Unused functions: The optimizer removed functions from target programs, sometimes removing the entire code; otherwise leaving only the function signature. For instance, the *fromURN* and *fromBytes* functions were removed entirely from the UUID program, while all instructions within the *fromBinary* function were removed. The test cases did not exercise the first two functions. Test cases called the last function, but its return values were not verified. Thus, the two former functions were removed, and only the signature of the last one was preserved. This result reinforces that the optimizer can support the tester by showing alternative ways of testing the program under interest. To check whether the problem here is a lack of coverage for functions, we wrote a new test case for the *fromURN* function and submitted it to the program developers through a *pull request*[4]. They have not accepted nor rejected the pull request so far.

Function parameters: In almost all programs we observed the removal of parameters from functions. Sometimes the parameters are removed only from the function signature; otherwise, they are removed from function invocations. Most of these parameters are not used by the functions or coalesce to a default value if not provided in the invocation, causing the test cases to run correctly. Also, a function may undergo a refactoring and stops using a parameter, but it is still a forgotten signature of the method. Also, there may be cases where the situation is reversed, and the change causes a defect. This kind of modification needs to be checked by a developer for every case. The absence of coverage in test cases or discarding result values from functions instead of comparing them to expected values may lead to this kind of optimization.

Function invocations: As function parameters, in almost all target programs we observe that function invocations were removed, that is, the execution of such functions does not interfere in the test cases. This type of change is related to a lack of coverage, leading the optimizer to remove the function declaration and its invocations completely.

Default values: Some of the target programs, such as D3-node, TLeaf, and Pug, have default values that are used in case of a function (or object) is invoked without a certain parameter (or property) required for its execution. In such cases, the requested parameter coalesces to the default value. Using D3-node as an example, almost all of its default values were removed. It was possible because its test cases always pass values and verify the results of the functions they call. These programs lack test cases to exercise the use of default values. In summary, we can classify this group of modifications as a possible lack of specific coverage: the code is tested, but the test cases cannot prevent the optimizer from removing the instructions. Again, we sent a *pull request*[5] to the target program's developer with our suggestion to improve tests. The developer accepted the new test cases as part of the program's test suite. It provided us the first evidence that the optimizer can indirectly help to improve the quality of the test cases.

Properties on objects: JavaScript programmers often use objects as key-value dictionaries to store temporary values in memory. In this case, one way to release memory is to remove unused keys from these objects using the 'delete' statement. Because of the statement nature, where property or an object no longer exists in the scope of a function, the tests are not directly influenced by it. So, this type of code was removed by the optimizer. It can be seen in *Moment*, *Lodash*, *Underscore*, *UUID*, and *Esprima*. Even though it is a correct change (do not clear memory), this type of modification can increase memory consumption and should be evaluated by a developer. The *pull request* cited in the unused functions section includes candidate test cases to cover this situation.

Dependencies: JavaScript programs that are written for the NodeJs environment use the *require* function to include other files or libraries. This function was removed from some target programs, such as *Mathjs*, *Pug*, and *XML2JS*. For instance,

4. https://github.com/pnegri/uuid-js/pull/20

5. https://github.com/d3-node/d3-node/pull/33

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TSE.2019.2928293, IEEE Transactions on Software Engineering

the 'pug-strip-comments' dependency has been removed from *Pug*. This dependency handles comments within templates before the library parses these templates. In this example, removing the dependency does not cause an error in the object meant to receive the template without comments: the difference in the result is that the comments have not been removed from the template. Because the unit tests of the library do not check if there are comments in the template, they still run without failure. However, this change introduces a bug as the template is left with comments that should have been removed. This type of change can be considered a lack of coverage and, again, contributes to a perception about the low quality of the tests themselves.

Return: The removal of 'return' statements within functions was observed in almost all target programs. This type of removal directly refers to the lack of coverage, where code runs successfully, but library tests do not check its result.

Conditionals: The removal of conditional statements has been observed in almost all target programs (whole IF blocks or some of their conditions). In most cases, it is possible to observe a lack of coverage in exception scenarios as test cases do not cover error handling. As an example, we can observe the removal of an IF from the *Minimist* library. The removed condition checks, inside of *isNumber* function, whether the parameter sent is a number. This function does not have test cases to cover it directly. In contrast, several test cases exercise its code, though none of them use a non-numeric parameter, that is, that exception condition is never actually tested. To test our hypothesis about the quality of the tests we included in the library a new test scenario where we pass a non-numeric parameter and check if the library does the treatment correctly. With this new test, Optimization was not able to remove the condition. The modification was submitted to the author of the library [6]. Despite the example of incorrect modification produced by the optimizer, there are situations where a developer would need to evaluate the correctness of this type of modification.

Uncovered instructions: Except for *Browserify*, *Esprima*, *Express*, and *Minimist*, all target programs had some code simply removed because the tests did not cover it. Developers should always check removals for lack of coverage. After all, it is easy to accept that there may be correct and useful code for such removals. However, even code covered by test cases was removed. These code snippets were exercised by the tests but made no difference to their execution. An example can be seen in the *Exectimer* target program, for which the optimizer removed a set of instructions that sort data inside the *getTimerWithTicks* function. All tests covering this function used ordered values, that is, from the perspective of tests there was no need for sorting. However, in practice there may be unordered data and the change may add defects to the program's users. To test that the removed code was indeed correct and the problem was a lack of coverage, we forked the project repository and submitted two new test cases based on unsorted data to the author of the program through a *pull request*[7]. The developer accepted the new test cases as part of the test suite for the library. It provided us the second evidence that the optimizer can indirectly help in the quality of the observed tests.

6. https://github.com/substack/minimist/pull/125
7. https://github.com/alexandrusavin/exectimer/pull/17/files

5.5 Discussion

A concern that emerges from the results discussed previously is that the more the optimization process can find changes, the lower is the perceived test suite quality. Sometimes this perception is due to the lack of coverage for specific scenarios that would exercise code statements or declarations, but in many circumstances, we observe coverage that lacks quality: either the results of code execution are not verified to an oracle or the code is executed under situations that do not raise specific conditions of interest. Thus, test coverage is not enough to guarantee code quality.

But then, what other quality metrics for test suites can guide us in this discussion? To our knowledge, no metric can generate the results that the optimizer generates. The contribution of this research is the creation of a technique to foster the improvement of test cases and source code for JavaScript programs, the optimizer's results exposing a lack of quality in the tests or modifications for improvement in the source code. One piece of evidence supporting this assertion is that the developers readily accepted the tests we modified and submitted to their projects. As we lack metrics to capture functional requirements coverage by test cases effectively, the optimizer may help to expose functional aspects that are not adequately tested, such as the cases mentioned in Section 5.4.

The joint improvement of test cases and source code has been proposed by Arcuri and Yao [25], who use GP to evolve a *buggy* program built from a formal specification and search-based testing techniques to create a test suite (from the same specification) that finds bugs in the program under analysis, leading to a competitive co-evolution between the program and its tests. We approach the problem from a size-reduction perspective, and while we do not automatically fix the program to its reduced size, the optimizer finds functional coverage limitations without a formal specification.

Another discussion emerging from the results is the need for a computational environment to improve JavaScript code. Such may hinder the practical application of the proposed approach. However, this environment was used to reduce the observation time and to provide the memory required to shorten several extensive target programs in parallel, particularly while analyzing the behavior of memory-demanding genetic programming. An environment such as Lobo Carneiro allowed us to test scenarios, improve the optimizer, and observe the results again and again. Even taking the cost of sharing resources in such an environment in consideration, it increased the pace of our experiments. However, once determined which heuristic and configuration gave the most robust results, the processing power needed to improve a target program falls to levels compatible with a good desktop computer. For instance, we applied the optimizer to *Moment* (9,978 LOC and 2,514 test cases) using DFAHC in a conventional computer (a DELL OptiPlex 3040 with a Core i5 CPU and 16GB RAM running Ubuntu Linux). It took 01:27 hours to run the optimization, almost five times faster than the 06:51 hours in Lobo Carneiro.

However, this is a single observation. Though the optimization looks faster on a personal computer, it is only possible to run one optimization round at a time in such a

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TSE.2019.2928293, IEEE Transactions on Software Engineering

computer. The computational power of Lobo Carneiro allowed watching 30 simultaneous executions of each target program, which reduced the total time required from years to months. Just as an example, the results of the last experiments (19 libraries and three heuristics) count 159 days (or 3822 total hours). Those 159 days were reduced to 39, a quarter of the entire time that would be needed on a personal computer.

Finally, we have the comparison of a GP algorithm with local search. In the exploratory studies, we observed two main characteristics that rendered the GP unable to rival the local search results. The first is the fact that most of the valid results are very close to the original code. For a problem where the best solutions are close to the current solution, the exploitation mechanisms of local search algorithms are more efficient than the exploitative GP. Also, one-point crossover was proved inefficient to find valid alternatives of code due to the distribution of the mutations to be merged. In some of the related work [3] [12] [13] [15] [16] [10] [20] the authors have observed at least part of this behavior and created specific operators and representations to drive the GP.

GP was significantly improved by using the patch representation proposed by the GenProg team [10] [20], though the local search still outperformed it. Such results were tested only for small programs where the local search sometimes has enough budget to test all alternatives systematically. More extensive experiments are required for larger programs, on which the behavior of both algorithms may change. On the other hand, the patch representation may also benefit the local search, further reducing the memory consumed by the algorithm proposed in Section 4.

5.6 Threats to Validity

From an internal validity perspective, measurement can be seen as a threat. The improvement in program size was measured as the number of characters after code minification, where comments, blank lines, and some static code transformations are performed. One may argue that the number of characters is just a proxy for file size, which ultimately influences load time, since JavaScript allows using UTF-8 chars (some of which requiring more than one byte to be represented). However, the use of extended chars is not common in source code and, therefore, the correlation between file size and the number of characters is high.

On the conclusion validity, we observed that while DFAHC outperforms SFAHC for some programs, such results are close and sometimes superior for SFAHC. It is not possible to conclude that one approach is better than the other by analyzing the target programs selected for the experiment. However, DFAHC is cheaper than SFAHC as it needs to be run once to produce the results reported in the preceding sections. Anyway, it would require a larger sample to determine whether SFAHC is significantly superior to DFAHC. The experiments with the patch representation were also limited to small target programs and more investigations are required to strengthen the evidence that favors DFAHC.

Still on conclusion validity, it is reasonable to question why we have not compared the results produced by DFAHC with a state-of-the-art technique in research or practice. We found that practitioners have so far given more attention to JavaScript than academics and, therefore, state-of-the-art JavaScript improvement tools are found in practice. There are strong arguments about which is the best tool to reduce the size of JavaScript programs, but most of these arguments concentrate on the *UglifyJS* minifier and the *Google Closure Compiler* (GCC) tool. Bengtsson [21] tested both tools in 90 programs and found that results from *UglifyJS* were 0.8% smaller than those produced by GCC. On the other hand, Facebook developers favor GCC for dead code elimination and function inlining [22]. We have compared *UglifyJS* and GCC for the programs used in our studies and found that GCC on average increases the size of the programs by 20.71% (ranging from -3,60% to 109,8%). Therefore, we have not compared our approach to a state-of-the-art tool because such tool (in this case, *UglifyJS*) is already the baseline for our comparisons.

Finally, with regards to external validity, we need a human being to evaluate modifications at process end, where (s)he evaluates the results and accepts or not the code changes. In industrial practices, it is well known that code review is not a necessarily executed activity. This activity is a crucial point for acceptance of the modifications found by the optimization process. Therefore, we have a threat regarding the development process of a particular project not to contemplate the activity of code review and, consequently, do not absorb the changes. One way to handle this risk is to couple the optimizer to the project build process, where a continuous integration server could run the optimizer, read the results, and send a patch to the developer for evaluation.

6 Conclusion

This paper presented DFAHC, a search algorithm that reduces the size of JavaScript programs to decrease their load and execution time, particularly in web and mobile applications. DFAHC was designed according to the results of exploratory studies performed to characterize the solution landscape for the problem of source code size reduction in JavaScript programs.

The exploratory studies show that local search finds better results than standard GP due to the distribution of the patches required to improve the source code, which hinders the effectiveness of the crossover operator. They also show the most essential node types in JavaScript syntax trees to be considered for removal by the search.

The second set of experiments were performed to determine whether adding a stochastic component to DFAHC would increase its performance and how DFAHC compared to the GenProg approach. The results show that a stochastic component related to the selection of candidate node for removal from the syntax tree not necessarily improves the deterministic process and that the latter outperforms the GP based on a patch representation, at least for small instances.

As future work, we intend to explore whether it is possible to apply other slicing techniques to JavaScript programs and how they compare to the proposed optimization. It is unclear if the characteristics of the JavaScript language mentioned in Section 2.1 will allow using current slicing techniques. If possible, these techniques may allow improving the optimization performance by signaling statements with potential for removal. We also intend to explore the optimization of other non-functional characteristics besides program size, such as memory consumption and energy footprint. Besides, we want

to integrate the optimization framework with CI processes in industrial settings, an engineering challenge to verify whether the technique presented here can scale to more massive code bases and be connected to other tools, such as submitting patches as pull requests to a human engineer.

Acknowledgments

This research was developed with the support of the Núcleo Avançado de Computação de Alto Desempenho (NACAD) of COPPE/UFRJ. CAPES and CNPq supported this research. Prof. Travassos and Prof. Barros are CNPq Researchers.

References

[1] Petke, J., Haraldsson, S., Harman, M., Langdon, W., White, D., Woodward, J. *Genetic Improvement of Software: a Comprehensive Survey. IEEE Transactions on Evolutionary Computation, Vol. 22, Issue 3, pp. 415–432, 2018*

[2] Leach, P.J.; Mealling, M.; Salz, R. *Rfc 4122: A universally unique identifier (uuid) urn namespace, 2005*

[3] Harman, M., Langdon, W., Jia, Y., White, D., Arcuri, A., Clark, J. *The GISMOE challenge: Constructing the Pareto program surface using genetic programming to find better programs. Proceedings of the 27th IEEE/ACM International Conference on Automated Software Engineering (ASE), 2012*

[4] Arcuri, A., White, D., Clark, J., Yao, X. *Multi-objective improvement of software using co-evolution and smart seeding. Asia-Pacific Conference on Simulated Evolution and Learning, pp. 61–70, 2008.*

[5] Anderson, C. *Type Inference for JavaScript. Ph.D. thesis, Department of Computing, Imperial College London, March 2006.*

[6] Bierman, G.; Abadi, M.; Torgersen, M. *Understanding typescript. European Conference on Object-Oriented Programming, pp 257–281, 2014.*

[7] R. Armstrong, D. Gannon, A. Geist *Toward a common component architecture for high-performance scientific computing. The Eighth International Symposium on High Performance Distributed Computing, 1999*

[8] Overbey, J., Xanthos, S., Johnson, R., & Foote, B. *Refactorings for Fortran and high-performance computing. Proceedings of the 2nd International Workshop on Software Engineering for HPC System Applications, pp. 37–39, 2005.*

[9] LeGoues. C., Nguyen, T., Forrest, S., Weimer, W. *GenProg: A generic method for automatic software repair. IEEE Transactions on Software Engineering, Vol. 38, Issue 1, Jan-Feb 2012.*

[10] Weimer, W., Nguyen, T., Le Goues, C., Forrest, S. *Automatically Finding Patches Using Genetic Programming* International Conference on Software Engineering (ICSE), 2009

[11] Harman, M.; Mansouri, S. A.; Zhang, Y. *Search based software engineering: A comprehensive analysis and review of trends techniques and applications. Department of Computer Science, King's College London, TR-09-03, 2009.*

[12] Langdon, W.B., Harman, M. *Improving 3D medical image registration CUDA software with genetic programming. Proceedings of the 2014 Annual Conference on Genetic and Evolutionary Computation, pp. 951-958, 2014.*

[13] Langdon, W.B., Harman, M. *Optimizing existing software with genetic programming.. IEEE Transactions on Evolutionary Computation, Vol. 19, Issue 1, pp. 118-135, 2015..*

[14] Silva, D., Sobral, L. *Um Estudo em Larga Escala sobre a Estrutura do Código-fonte de Pacotes JavaScript. Graduate final Report, 2017, Applied Computer School of Center of Exact Sciences and Technology, Unirio. Rio de Janeiro..*

[15] Senaratna, N. I. *Genetic Algorithms: The Crossover-Mutation Debate. Technical Report, University of Colombo, 2017.*

[16] Fogel, D., Atmar, J. *Comparing Genetic Operators with Gaussian Mutations in Simulated Evolutionary Processes Using Linear Systems. Biological Cybernetics, Vol. 63, pp. 111–114, 1990.*

[17] Myers, E. *AnO(ND) difference algorithm and its variations Algorithmica, Vol. 1, pp 251–266, November 1986.*

[18] Brownlee, J. *Clever Algorithms: Nature-Inspired Programming Recipes Lulu.com, 2011*

[19] Forrest, S., Mitchell, M. *Relative Building-Block Fitness and the Building-Block Hypothesis Foundations of genetic algorithms 2, Morgan Kaufmann, 1993*

[20] LeGoues, C., Dewey-Vogt, M., Forrest, S., Weimer, W. *A systematic study of automated program repair: Fixing 55 out of 105 bugs for $8 each Proceedings of the 34th International Conference on Software Engineering, 2012.*

[21] Bengtsson, P. Advanced Closure Compiler vs UglifyJS2, at https://tinyurl.com/zskubok, last accessed 13-June-2019, 2016

[22] Abramov, D., Vaughn, B. Behind the Scenes: Improving the Repository Infrastructure, at https://tinyurl.com/ybevwjk8, last accessed 13-June-2019, 2017

[23] Landsborough, J., Harding, S., Fugate, S. Removing the Kitchen Sink from Software, Genetic Improvement Workshop, 2015, pp. 833–838, Spain

[24] Yeboah-Antwi, K., Baudry, B. Removing the Kitchen Sink from Software, Genetic Improvement Workshop, 2015, pp. 839–844, Spain

[25] Arcuri, A., Yao, X. A Novel Co-evolution Approach to Automatic Software Bug Fixing, IEEE Congress on Evolutionary Computation, 2008, pp. 162–168, China

[26] Langdon, W., White, D., Harman, M., Jia, Y., Petke, J. API-Constrained Genetic Improvement, Symposium on Search-based Software Engineering, 2016, pp. 224–230, USA

[27] Cody-Kenny, B., Manganiello, U., Farrelly, J., Ronayne, A., Considine, E., McGuire, T. and O'Neill, M. Investigating the Evolvability of Web Page Load Time Proc. of the Nature-inspired algorithms in Software Engineering and Testing, EVOSET'18, Italy, 2018

[28] Brownlee, A., Petke, J., Alexander, B., Barr, E., Wagner, M., White D. Gin: Genetic Improvement Research Made Easy Proc. of the Genetic and Evolutionary Computation Conference (GECCO'19), Prague, CZ, 2019

[29] White, D. GI in No Time Proc. of the Genetic and Evolutionary Computation Conference (GECCO'19), Berlin, DE, 2017

[30] An, G., Kim, J., and Yoo, S. Comparing Line and AST Granularity Level for Program Repair using PyGGI Proc. of the 4th International Genetic Improvement Workshop, pp 19–26, Gothemburg, Sweden, 2018

[31] Petke, J., Langdon, W. B., Harman, M. Applying genetic improvement to MiniSAT. Proc. of International Symposium on Search Based Software Engineering, pp 257-262, Berlin, Heidelberg, 2013