de Almeida Farzat et al. Journal of Software Engineering Research and Development (2018) 6:12
https://doi.org/10.1186/s40411-018-0056-2

Journal of Software Engineering Research and Development

RESEARCH Open Access

Challenges on applying genetic improvement in JavaScript using a high-performance computer

Fábio de Almeida Farzat[1*], Márcio de Oliveira Barros[2] and Guilherme Horta Travassos[1]

* Correspondence:
fabio.farzat@cos.ufrj.br
[1]Computers and System Engineering Department, COPPE/UFRJ, Cx Postal 68501, Cidade Universitária, Rio de Janeiro, RJ, Brazil
Full list of author information is available at the end of the article

Abstract

Genetic Improvement is an area of Search Based Software Engineering that aims to apply evolutionary computing operators to the software source code to improve it according to one or more quality metrics. This article describes challenges related to experimental studies using Genetic Improvement in JavaScript (an interpreted and non-typed language). It describes our experience on performing a study with fifteen projects submitted to genetic improvement with the use of a supercomputer. The construction of specific software infrastructure to support such an experimentation environment reveals peculiarities (parallelization problems, management of threads, etc.) that must be carefully considered to avoid future research threats to validity such as dead-ends, which make it impossible to observe relevant phenomena (code transformation) to the understanding of software improvements and evolution.

Keywords: Genetic Improvement, Source code Optimization, Search Based Software Engineering

1 Introduction

Genetic Improvement (GI) (Harman et al. 2012) is a set of techniques that allows the evolution of software through a process of genetic programming, aiming to improve the software regarding a set of quality criteria. Genetic Programming (Arcuri et al. 2008) is an automated method for creating a computer program from the high-level definition of a problem.

GI preserves the functional properties of the software. It can be seen as an automatic source code refactoring technique that generates functionally equivalent variants, which offer improvements in one or more non-functional properties such as usability, flexibility, performance, among others. The variants produced by the process are readable for a human and should be checked by software engineers before their use because changes can remove correct software behavior. However, the technique removes from the software engineers the laborious task of navigating among implementation alternatives to reach the one meeting the required software non-functional characteristics. Instead, the Optimization process proposes a text patch with changes that might attend to the needs expressed by a fitness function that drives the optimization.

The application of this kind of research is naturally experimental. Different configurations of an optimization algorithm (optimizer) should be executed until the observation that there is no difference in the optimizer performance regardless of the program under analysis can be observed. The large number of syntactical elements comprising a programming language and the diversity of selection, crossover, and mutation operators that can be applied to the source code leads to an extensive list of optimizer composition alternatives. A long cycle of experimentation is required to understand the patterns of changes made to the program and revealing optimization settings for a given language and domain. With each experiment cycle, more information is acquired, promoting adjustments to the optimization process, and requiring more experimentation cycles to study their effects.

Each experiment cycle generates a program variant, verifies that it compiles and executes the program's test cases dozens of times to take into account the random aspects inherent from the selected improvement operators. Furthermore, the optimizer checks whether the program variant represents an improvement from the standpoint of the non-functional criteria under analysis. This process is costly regarding requiring significant computer resources to be executed.

JavaScript is an interpreted programming language that is available on multiple platforms, has limited memory management, and uses objects whose properties and methods can change at runtime. To our knowledge, no GI technique has been applied to JavaScript programs so far. We have designed and built an execution time optimizer targeting in reducing execution time, capable of handling a broad range of JavaScript programs on multiple platforms and encompassing all features provided by the language. The need to reduce the execution time of JavaScript code becomes a relevant problem, where market solutions already exist.[1] Some libraries studied in this survey exceed 10 million downloads per month. This number only implies new uses of the library. The number of web applications already running and using these libraries cannot be determined accurately. Optimizing the code of such a library in 1 s, for example, can mean a significant overall reduction.

The construction and testing of the optimizer revealed the need for using high-performance computing (HPC) resources (supercomputing) to enable the execution of the experimental plan. It was due to the amount of memory required to maintain, at the same running time, multiple source code variants and their correspondingly testing in order to validate variants correctness.

This paper summarizes the challenges observed to plan and execute an experimental study to optimize the execution time of JavaScript programs involving high-performance computing (HPC) in Software Engineering, Such challenges include rewriting GI operations to cope with JavaScript code, parallelization of a single thread system (NodeJs) to allow using the resources provided by the HPC environment, among others.,. The experimental study reported here does not need human intervention to apply the treatments and to collect the independent variables values. High-performance computing was required because the application of treatments takes on average 360 hours (15 days) of execution in a high-end computation node (Section 4.5). The application of at least 90 treatments (two optimization strategies and three algorithms applied in 15 software projects) in the available time required the use of a supercomputer. We hope that the experiences described in this paper and the proposed solutions can be useful to support

other researchers in the field, allowing anticipating possible risks in the project and execution of future studies. Problems encountered in this research can lead to a way of no return.

An example is the memory consumption required to create variants of the original code. JavaScript has severe limitations on memory management. Other interpreted languages may also present the same characteristics. This study pioneers the application of GI in an interpreted and dynamic language and it is one of the first to use high-performance computing in Software Engineering research.

This paper extends the discussions and results presented in (Farzat et al. 2017). It detailed information about the experiments that were executed and included further data explaining the results and supporting the explanation on the observed challenges. Of the first three libraries observed were added the results of a further ten, totaling 13 observations in the HPC environment. It is organized into nine sections. The first contains this introduction. Section 2 presents former works related to the application of GI in Software Engineering. The third section explores the characteristics of JavaScript and its execution contexts. Next, Section 4 details the implementation of the toolkit for applying GI in the context of JavaScript as well as the challenges of planning and executing such experiments. The fifth section describes the optimization process, basing on preliminary results observed in three JavaScript libraries. Next, preliminary results are presented to support the explanation of challenges. Section 7 discusses the threats to the validity regarding our findings, followed by a collection of lessons learned in Section 8. Next, Section 9 concludes the discussions and present considerations for future research.

2 Background

JavaScript has become the most common scripting language on the client side of Web applications since its launch in 1995, according to GitHub (Anderson 2005). This growth is due to the evolution of the systems available on the Internet, which required a more efficient interaction model between the client and server side (Anderson and Drossopoulou 2006). The ability to include some of the systems' logic in web pages led the developers to write large computer programs using a language suitable for scripting, not for large-scale programming (Bierman et al. 2014). The intensive language use led to the creation of libraries, such as jQuery, Angular and React, which support the construction of sophisticated JavaScript software. These libraries can be considered large programs, especially if we take into account that the original purpose of JavaScript was to produce small programs. For example, jQuery has 11,026 lines of code and 294,199 characters in a single file.

JavaScript is object-based. It uses object prototypes to compose classes and allow inheritance (Bierman et al. 2014). Objects consist of string mappings (property names) for values. Properties can be added and removed from these objects at runtime, and their names can be dynamically calculated. Also, variable values are freely converted from one type to another with few exceptions for which automatic conversion does not apply.

Another essential aspect of JavaScript programs is their physical size. As software written in JavaScript executes on the client side (in a browser), the source code must be transferred to the client before its execution. The larger the code, the longer the

transfer time will be. In this sense, software engineering created techniques for reducing the size of JavaScript code, such as minification (Fountoukis and Chatzistavrou 2009). These techniques reduce the names of variables, methods, and objects, as well as remove white spaces and comments in excess. The minification process is also used to make it difficult to understand the JavaScript code since it becomes practically unreadable for a human.

In addition to browsers, JavaScript programs can currently run on smartphones and servers. On mobile platforms (e.g., iOS, Android, Windows Mobile), software systems written in JavaScript are called hybrids (Overbey et al. 2005). A hybrid system uses platform-specific browsers by changing their appearance and allowing JavaScript programs to interact with device features and simulate an application written and compiled in the platform's native language. Regarding the servers, a JavaScript engine called NodeJs runs as an HTTP server and allows serving pages and processing information, as well as other server behaviors, by using JavaScript as the base language.

Because of the increased use of JavaScript in the last decades [XX] and characteristics cited before, we must find a set of changes that improve a JavaScript source code from an execution time perspective. The motivation of this work is in context of construction and maintenance of web systems.

2.1 Genetic improvement

In 1992, John Koza used genetic algorithms (GA) to automatically generate programs for accomplishing specific tasks, such as solving mathematical expressions. The author baptized the method as Genetic Programming (GP). In GP, syntax trees contain function or terminal nodes to represent the programs. Functions can be arithmetic, Boolean, conditional, iteration-related, recursion-related, in addition to domain-specific functions for the problem in question. Terminals are variables or constants. Therefore, GP is a GA specialization in the domain of manipulating computer programs.

Genetic Improvement (GI) makes use of GP to evolve a human-written software (a target program) taking into account a set of quality criteria. Like GP, GI uses a syntax tree representation to evolve the target program over a sequence of generations in search of improved versions from the selected quality criteria perspective.

2.2 Local search

Local search is a class of point-based algorithms that use heuristics to systematic traverse the neighborhood of a given individual 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 it finds a better neighbor, 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.

3 Related work

There is much research in the area of GI. Much of the work focuses on improving the execution time of the programs submitted for optimization (Orlov and Sipper 2011; Petke et al. 2013). However, the technique can also be applied to improving other non-functional requirements (Harman et al. 2012). Research conducted using GI is divided into four major groups (Silva et al. 2009): defect correction; improvement of execution time; migration and transplantation; and dynamic adaptation. Much of the present work on GI focus on improving the target programs execution time as we can see in the (Petke et al. 2018) Survey. However, the technique can be applied to improve other non-functional requirements. Our research focuses on the second group (runtime improvement), aiming to reduce the transfer and execution time of a JavaScript target program.

Petke et al. (2013) applied GI to a system called MiniSAT, an open source project written in C++ to solve Boolean problems (SAT, Satisfiability or Boolean Satisfiability Problem). It uses state-of-the-art technologies for solving SAT problems, including Unit propagation, Conflict-driven clause learning and Watched literals (Jensen et al. 2009). According to them, the genetic improvement of the MiniSAT was a significant challenge, since expert programmers evolved it over the years. The researchers wanted to check if it was possible to improve the best human solution execution time. Harman et al. (2012) described the optimization approach. The optimization process was applied to one core system class (*Solver.cc*), and the observed improvement was 1% in the best case (measured in lines of code), where assertions were erased from the code.

White et al. (2010) proposed a framework for C program optimization using multi-objective GI to meet pre-configured non-functional criteria, where the software engineer can extend the evaluation behavior to include further criterion as needed (memory, disk, energy, among others). In particular, they evaluated the computer programs execution time. The authors selected eight distinct functions of the software for a simulation-based evaluation and submitted them to the optimization process. The results show an exciting improvement: in the best case, there was a 5% improvement over the original computer program. It is worth noting that some improvement patterns have been identified, such as the types of instructions executing faster than others do. However, the limitations in the study reduce the capacity of observation, due to i) the tests to evaluate a change were selected according to some structural criterion (coverage, for example), assessing the changes from a different perspective from the original; (ii) the experiment optimized the functions separately, observing improvements only in this isolated context; and (iii) there was no update of these functions in the original software for evaluation and comparison with all of them updated.

Harman et al. (2014) applied GI in the migration and transplantation of functionalities between software systems in operation. The researchers experimented with using an instant messaging system (Pidgin), and another one of text translation (Babel Fish). The goal was to add text translation behavior to Pidgin. The technique is divided into two parts: growth and graft. In the growth part, a software developer selects the parts responsible for the functionality along with their tests so that the optimization process looks for a group of instructions (classes and methods) optimized from a runtime perspective. In the graft part, the challenge is to find insertion points in the target software. To do so, the researchers randomly choose a point and apply three distinct types

of mutation (variable replacement, statement replacement, and statement swapping). The study performed thirty rounds of the optimization process, with a population of 500 individuals evolved over 20 generations. The authors reported that in addition to finding real solutions, this was the first work of code transplantation using SBSE to that date.

Cody-Kenny et al. (2015) proposed a tool called *locoGP* that uses GI in Java computer programs to reduce their size, measured by the number of instructions. The tool reads the source code in Java, assembles its syntactic tree, and applies crossover and mutation operations. Operations can swap, erase, or clone nodes of the syntax tree as selected for the operation. Program instances implementing 12 distinct sorting algorithms (*Insertion Sort*, *Bubblesort*, *BubbleLoops*, among others) supported the tool evaluation. In all cases, *locoGP* was able to find better alternatives than the implementations presented to it, that is, it reduced the number of instructions used in each of these algorithms.

Although we have not found previous experiences of using HPC for the execution and analysis of experimental studies in the context of Software Engineering, the technical literature presents works involving the application of Software Engineering techniques in HPC environments. Fountoukis and Chatzistavrou (2018) propose the use of design patterns for software written for HPC environments. Two standards are presented to address bottlenecks in parallel execution, a common problem in this context. Overbey et al. (2005) present a tool called Photran for automatic refactoring of Fortran code. The tool uses static code transformations to increase the use of resources in HPC environments. According to the authors, the advent of object-oriented Fortran made it possible to observe static patterns of instruction exchange related to parallelism.

The related works presented here represent a small set of literature review works that we did during the research. Despite seeking inspiration in these researches, none influenced directly the implementation decisions described in this paper. The influence is presented in the experimental plan described in the next section.

4 Experimental plan

The study planned in this research aims to observe the results of genetic improvement in JavaScript code (technology-oriented experiment). The experimental plan follows a basic structure: context selection, hypothesis formulation, variable selection, participant selection, experiment design, instrumentation and validity evaluation, as suggested by (Wohlin et al. 2012).

Context selection involves the balancing between making the study valid for a specific context or the general domain of Software Engineering. Our scenario in this experimental study is Specific x Generic, where we will observe the results from the perspective of generalization in JavaScript. For this, the null hypothesis regards applying GI in JavaScript code does not produce appropriate results.

The independent variable selected was the test cases execution time, measured in milliseconds. Since the optimization perspective is the code execution time, running the test suite faster is the optimization criterion guiding the search. The treatment applied is the transformation of code instructions with the purpose of reducing (remove

Table 1 Thirteen libraries observed. The first column shows the number of lines of each library

Library	Loc	Tests	Coverage %	Downloads (X1000)
EXECTIMER	228	37	91%	0.6
PUG	358	240	98%	226
LODASH	11,877	2077	94%	30,799
MINIMIST	235	140	92%	25,617
MOMENT	9978	2514	96%	5442
UNDERSCORE	1481	198	95%	8710
UUID	240	21	91%	11,497
XML2JS	538	83	93%	3783
PLIVO-NODE	928	26	94%	0.1
GULP-CCCR	525	17	90%	0.1
EXPRESS-IFTTT	160	29	93%	0.2
TLEAF	282	167	91%	0.3
BROWSERIFY	755	570	99%	1987

Column tests give the number of unit tests. Coverage shows the percent of instructions that are exercised by unit tests and Download column shows the number of downloads of each library for November of 2016

or transform) instructions, where it is expected that with fewer instructions the execution time is also smaller.

The observed sample is described in Section 6, Table 1. The criterion for the selection of participants was the coverage percentage, where only libraries with 90% or more of code coverage were selected. The selection procedure was manual. The researchers searched the online repository called NPM[2] and selected the libraries randomly, based only on the coverage criteria.

The experimental design is one factor and more than two treatments. The comparison regards the average improvement observed between treatments (GI, Local search, and random search). All instrumentation of the experimental study is automated in the Optimizer, as will be seen in Sections 5 and 6.

During the Optimizer construction, these experimental studies were performed twice on different platforms: using ordinary machines and using HPC (Section 5). The first configuration executed 30 optimization cycles for each library in which the genetic algorithm used an arrangement of 100 individuals evolved over 50 generations. In HPC, due to the available resources, we increased the observation for 60 optimization cycles and repeated the other configurations. Each variant presenting an original code improvement was executed 20 times to improve the execution time reliability of the test case suite. The initial implementation of each library received the same treatment. It intends to obtain a consistent baseline for comparisons against not controlled components residing in the execution time measure of programs in a multitasking environment, although restricting the number of applications running in parallel.

The results obtained with conventional machines were discarded due to the need to observe the normalized effects. The main reason for rejecting the first results is to maintain the basis of comparison using an HPC environment since the code optimization perspective was the execution time during the Optimizer construction. The observed libraries execution times are different between the ordinary machines and HPC environment.

During the tool development, the most complex problems were the lack of standardization for execution of unit tests in JavaScript; the need for an engine to interpret programs written in this language, and the difficulties to compose execution environments compatible with the platforms where the projects. All of these represent challenges faced during the study execution. Therefore, the experimental environment integrated different tools and libraries to allow applying GI without platform restrictions and using all JavaScript resources.

5 Genetic improvement for JavaScript source-code

With the objective of constructing a tool that could serve as the basis for the execution of experiments of this nature in JavaScript language, performing of some failed tasks brought the necessary knowledge to move forward with the work. During 3 years of research, we built four "Optimizer" versions, each with its own weak and strong points. In this section, we present these versions, the challenges addressed on each research step, and we cover in detail the latest version available. The applied solutions relevant for HPC environments have their characteristics, and their usage in software engineering experiments is, in many cases, neither conventional nor straightforward.

5.1 The *ECJ* tool

Previous experiments in GI (White 2010) mention the ECJ[3] library use to support the application of genetic programming, grammatical evolution, and genetic improvement. Based on these previous experiences, we created a simple plan, which consisted of observing the operators built in a controlled code (Section 5), and developed the first version of the optimizer using the ECJ's modules related to morphological evolution. Grammatical evolution refers to the use of grammar to describe the source code to be optimized instead of a syntactic tree. The first version of the optimizer was unable to create optimized variants of large libraries. That is, a library with more than a thousand lines of code would not be subject of future experiments. Besides that, we could not represent some situations commonly encountered in JavaScript programs, such as overloaded methods, recursive methods, and the use of global JavaScript objects. Therefore, it was not possible to use just ECJ to support our automatic experiments.

Therefore, a module to generate the grammar was built as an external executable to ECJ. It uses ANTLR,[4] which is a well-known component for generating grammars. ANTRL allowed creating a generic parser and exporting the information to a file in the ECJ format. Besides, we decided to abandon the search for an optimization suite that was already done and started the development of a complete JavaScript Optimizer after the process of generating the grammar from JavaScript was ready and isolated from ECJ. However, we would need a JavaScript interpreter to contemplate the supported ECJ steps (the genetic algorithm and operations on grammar), as well as the execution of unit tests of libraries and their evaluation.

5.2 The *Jurassic* project

The grammar generator was written in C# using ANTLR components. Therefore, we sought an interpreter also drafted in C#. After preliminary evaluations, we opted for the Jurassic project,[5] which allowed us to interpret JavaScript source code without the

need to generate a grammar to apply the genetic operations. In the second version of the optimizer, we represented the JavaScript source code as a syntax tree. The changes suggested by the evolutionary operators were applied to the syntactic tree, and afterward, the variant source code was generated. This allowed us to overcome the limitations encountered with ECJ, but revealed new problems: (i) the Jurassic library presents low performance to interpret the source code of large libraries (with a thousand or more lines of code); (ii) the limitations found in the syntactic tree construction, like recursion, did not allow Jurassic to correctly interpret some libraries using dynamic methods (defined at *runtime*); and (iii) Jurassic does not support the version of JavaScript (ECMA 6.0) used in the development. Using the most current version of JavaScript allows observing current and shared libraries in private/market projects. Therefore, Jurassic was also discarded.

5.3 Chrome V8

Next, we looked for JavaScript interpretation engines that could be embedded in the Optimizer implementation. The option to build the third version of the optimizer was Chrome V8, which could be added to the Optimizer due to the C# and C++ interoperability. Therefore, it was necessary to build the entire Chrome V8 development environment and compile a specific version for that purpose. With the use of Chrome V8, the time required to interpret JavaScript programs was reduced and we could guarantee the compatibility with the latest programming language version. The crossover and mutation operations were implemented, and support was added for the unit tests execution.

However, new problems emerged in this version. By creating the first real original code variations, sometimes the optimizer produced code containing recursive calls or infinite loops. These features caused memory leaks and stack overflows in Chrome V8, stopping the optimizer. Also, each new library required configuration and development effort to support the library's specific features and unit tests (browsers, dependencies, and environment variables. Since Chrome V8 is not a browser, unit-testing support was not provided by default. Therefore, it was coded on an instance basis. Nevertheless, it was possible to perform automatic experiments in three libraries and observe their results described in Section 5.

5.4 Nodejs and typescript

From the analysis of previous results, we decided to build a fourth version of the optimizer addressing all the observed problems, including the default support for the used programming language version (ECMA 6.0) and requiring less effort (dependencies and environment variables) to observe more projects written in JavaScript. Therefore, a new Optimizer was developed making explicit all the requirements so far identified. The fourth version was entirely built on JavaScript using the NodeJs environment. Because NodeJs allow the using of Chrome V8, it preserves all the performance benefits gained in the third version and handles all JavaScript features (for instance global object, browser functions, and support for unit tests).

NodeJs allows adding libraries only by configuration, generating code variants using existing JavaScript libraries, and running unit tests natively, using a browser or any

library feature without modification in the library files. However, one of the difficulties in writing this new version was the fact that the Optimizer should be written in JavaScript, a programing language that is not suitable for large and complex programs such as the optimizer. However, the use of Typescript, a public-domain superset of JavaScript proposed by Microsoft that aims to simplify the development of large-scale applications (Bierman et al. 2014), mitigates this issue. Typescript provides a system of modules, classes, and interfaces, as well as primary types. One of its goals is to provide a smoother transition of programmers coming from object-oriented or strongly typed languages to JavaScript. Typescript allows generating JavaScript code, that is, it has a translator to produce a final JavaScript code with the characteristics and behaviors encoded in Typescript (e.g., classes). Also, any JavaScript code is a valid Typescript code. It allows using JavaScript libraries without any adaptation, merely referencing them and making use of their methods directly in the Typescript environment. Although the purpose of Typescript is to generate JavaScript code, its programming environment needs a JavaScript translation before its execution. Therefore, the Typescript programmer does not interact with the JavaScript code at development time, which is the primary factor for its adoption in this case.

Another challenge imposed by NodeJs is being a single thread, i.e., it only executes one process at a time. It is not possible, for instance, to generate and control two activities on the same execution stack. It prevented the Optimizer from adequately dealing with problems like lack of memory and fatal errors. Because Optimizer operations (as well as unit tests) can cause failures of these types, we have situations where the Optimizer execution terminated unexpectedly due to a fatal error or lack of resources. To address this problem, the Optimizer was redesigned to work in two layers: a server and a client layer. The purpose was to run an operation that might fail on the client and keep the primary process protected inside the server. The WebSocket protocol, which allows direct communication between different machines using the HTTP protocol, was used to exchange messages between the layers. Although it operates over HTTP, it lets a server to stay actively connected to clients throughout the process.

Optimizer exception handling started to work by controlling the connection status to a particular client. Once the server decides which operations need to be performed, it distributes them to the available clients, which in turn will carry out the operations. If one of these clients fails, the connection to the server will terminate unexpectedly. After the server notices the connection drop, it redirects the operation that caused the failure of another client. During the processing of operation by the clients, the server waits until a group of operations, which only have logical meaning together (all crossings and mutations in the genetic algorithm case), are completed and then proceed with the primary optimization process.

At this point, we begin the reproduction of the three experiments performed with the Optimizer third version. However, performance problems and lack of computational resources made it impossible to execute the experiments entirely and required us to restart the procedure a few times. Using multiple computers in a connected (HTTP) context has resulted in performance issues. Also, the memory consumption is very high for large libraries, with 5000 lines or more, making it impossible to use conventional platforms for their execution.

5.5 Using a high-performance computing environment on SE experiments

In July of 2016, it arose an opportunity to use a supercomputer of COPPE/UFRJ. Lobo Carneiro (LOBOC), in honor of the late professor Fernando Luiz Lobo Barboza Carneiro (1913–2001), is a supercomputer with the capacity of 226 teraflops, 16 Tbytes of RAM and 720 terabytes of disk, capable of running 226 trillion of mathematical operations per second. With these features available, the single thread limitation no longer exists (memory limitation over one single thread of NodeJs), and the fault handling returned to the previous state without having to split the Optimizer tasks among multiple clients (simpler and faster). It allowed us to conduct experiments with larger libraries (in lines of code) in a much shorter period.

To run the Optimizer in a parallel environment, like LOBOC, we produced a new specific optimizer version for that environment (fifth version). LOBOC runs SUSE Linux Enterprise Server version 12. As NodeJs does not have a specific package for this operating system, we compiled it under the assigned user account for use. Once NodeJs was running in LOBOC, the Optimizer was able to run its tests without significant problems. However, there was a complication: despite the NodeJs process can address the 16 terabytes of available memory; it was not able to see all available processors.

The LOBOC architecture is organized as a cluster, that is, it consists of 252 nodes, each with a processor composed of 24 cores using HyperThreading technology to execute up to 48 tasks in parallel. A queue system, called Altair PBS Professional, allocates the nodes to a process. The method of allocating resources (disk, memory, nodes, and cores) is simple: through a native Linux bash script file, it is possible to describe what features are needed to run a program and make use of them.

Even after allocating more nodes, NodeJs processes continued to use only 48 cores (24 physical cores, seen by the system as 48 cores due to HyperThreading). The PBS launches the Optimizer root process on one of these nodes that act as the execution host and informs that other node will compose the cluster during the session through a text file passed via command line. By itself, the Optimizer does not know the other nodes existence (and their cores). It was necessary to handle the details file sent by PBS inside the optimization process initialization.

First, each node proceeded to perform a genetic algorithm configuration, using an algorithm core and the other 47 cores for the evaluation of the objects under analysis (individuals). Then, 60 genetic algorithm rounds were executed in parallel (one on each node). We used 60 parallel rounds because the automatic experiments were designed to work with 60 optimization process observations per instance. Thus, we were able to run all the rounds of one library in parallel by the algorithm (for instance, Random Search) at a time. This composition consumed 2880 cores with 500 GB of RAM. In this scenario, the average duration of an automatic experiment was reduced to 10% of the previous total time (already in HPC, but without parallelism). Since we have all the necessary rounds in parallel, the last trial ending determines the total experiment time. Before finishing a library, others were started and the total time observed was the sum of these trials.

5.6 The current version of the optimizer

The fourth version[6] was completed in July 2016 and was evaluated through experiments with thirteen libraries with the goal to reduce the code execution time of these

libraries. The list of libraries that were observed is in Table 1. They were chosen based on size (measured in lines of code and presented in the LOC column), their use (measured in downloads per month), as well as the coverage and size of its suite of unit tests. The Downloads column shows the number of downloads for each library in December 2016. The suite size is measured in the number of unit test cases reported in the Tests column, while the coverage represents the percentage of library code statements exercised during the test case suite execution (Coverage column). As the whole process uses unit testing as an oracle, this is the primary criterion for selecting new libraries (code) for analysis. Little code coverage can generate false positives, i.e., removal or alteration of valid but untested code that will only be discovered during the results of the human analysis.

6 Preliminary results

As the adjustments described in the previous section were produced and the fourth version of Optimizer met the execution requirements, it was necessary to include more complicated projects to begin evaluating the proposal in real cases. From the list presented in Table 1, three libraries were selected as part of an initial set of instances submitted to the Optimizer: a) *Moment*, which treats dates in general (internationalization, *timezones*, among others); b) *Underscore*, a set of utilitarian routines (*foreach*, *cloneObject*, among others) to support JavaScript development; and c) *Lodash*, a framework of development functions in JavaScript that is an extension of Underscore methods. After completing the automatic experiments with these first three libraries, all the others planned automatic experiments were performed, totaling 12 distinct method observations.

In the first run (conventional computers), the process took 28 days to complete the three libraries on three machines with identical configuration: Intel Core i7 3.4 GHz quad-core processor and HyperThreading technology (up to eight simultaneous tasks) and 4GB of memory. As described in the experimental plan, these results were discarded, and we carried out the second execution of the automatic experiment using LOBOC.

Table 2 shows the preliminary results obtained in the study conducted in the supercomputer, after their completeness. Comparing these executions, we verified that the execution time of the three algorithms had a significant reduction: for example, The Optimization process fell from 28 days to 6 days of execution. These results are very superior of the 10% cited before. The previously observed mean was only three libraries. It is possible to observe an improvement in the test cases execution time in all libraries. In particular, an improvement of more than 70% is seen in the case of *Tleaf*.

A simple set of modifications to the source code is primarily responsible for the improvements observed in Table 2. An example is shown in Fig. 1 where we have an excerpt from the original Moment library code (on top) compared to a snippet of code found by the optimization process (on bottom). In this example, a condition was removed from an IF statement and improved code execution time (in 2.2%) without changing test results. It is possible to notice that the removed condition validates the "_locale" property of the "input" object used inside the IF (second line of code). It is a code snippet covered by the Moment unit tests. Still, ideally, a software engineer who knows the purpose of the modified routine evaluates the change and decides whether to incorporate it into the library. Removing a condition, where this condition is an

Table 2 Results of the fourth version of the Optimizer, representing the test cases execution time (in milliseconds) of the original implementation and the best variant found by the optimization process

Library	Original time	Best time	Improvement
UUID	903	779	13.73%
MINIMIST	2482	2283	8.02%
PLIVO-NODE	1041	824	20.85%
GULP-CCCR	905	803	11.27%
EXECTIMER	1158	1069	7.69%
EXPRESS-IFTTT	1475	1209	18.03%
TLEAF	26,005	7507	71.13%
JADE	6020	5200	13.62%
NODE-BROWSERIFY	60,391	58,200	3.63%
MOMENT	8541	8353	2.20%
LODASH	13,028	12,832	1.50%
UNDERSCORE	5308	4494	8.67%
XML2JS	7870	6466	17.84%

Original time, Best time e Improvement columns are means

internal call to another routine, may show that the test cases have not been well planned or are not well written.

In the case of the Underscore library, the observed improvement came from a simple change: the dimensions declaration of an array was removed, as it can be seen in Fig. 2. The same is used in a function called *times*, responsible for executing *n* times a received function per parameter. Since in the tests this function is performed in scenarios with a large number of repetitions (these tests are using a random number to execute internal checks, with a probability between 100 and 1000 repetitions for this function), it becomes the marked average improvement of 8.67% main determinant.

In the Tleaf library case, the results are more impressive, indicating 71% of time improvement. In this case, the Optimizer made substantial and significant modifications. The changes found in the library code are divided between the removal of parameters in a function call, elimination of the declaration and initialization of variables, removal of attributes in objects, assignment of values to variables and full body of functions. Of this group, the researchers' attention was drawn to the fact that the Optimizer was able to remove entire functions: sometimes its declaration is completely removed and in

```
var ret = newDuration(duration);
if(isDuration(input) && hasOwnProp(´input´,
´ locale´))
        ret._locale = input._locale;
```

```
var ret = newDuration(duration);
if(isDuration(input))
        ret._locale = input._locale;
```

Fig. 1 Moment original code (top) compared to the optimized code (down)

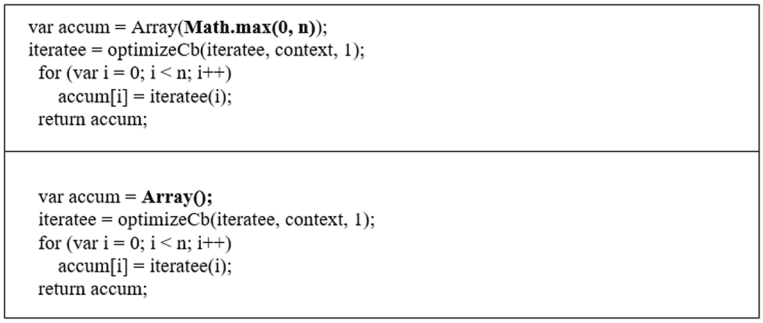

Fig. 2 Underscore original code (top) compared to optimized code (down)

others all its code while maintaining its declaration. The same type of behavior was observed in the uuid, gulp-cccr, express-ifttt, Plivo-Node, Jade, xml2js, and Node-Browserify libraries.

To understand how the Optimizer was able to remove entire functions, it was necessary to read the test cases as well the library code. During reading, it was clear that the reason why the removal happened is due to the behavior of the library. It changes according to the presence or absence of certain values in specific code properties (objects and properties).

Tleaf is a library for automatic generation of test code for Angular controllers, i.e., the physical path of a file containing one or several controllers written in Angular is passed to the library, in addition to the output path for automatic code generation of those controllers. Its test code is based on internal templates written inside the library code (string concatenation and use of some dictionaries). The file with test code generated by the library contains empty methods which the developer may include the appropriate testing behavior. In summary: Tleaf is a library that automatically generates empty test functions to increase a developer's productivity in Angular technology. It has 131 unit tests.

Once a software engineering understands the library domain, s/he can understand the results obtained by the Optimizer. It was able to remove properties from the internal templates used by the library to generate the test codes. For example, the Optimizer removed an attribute in a template called 'validate.' This attribute had 'required' as its value. This attribute determines that in the generation of a test code the attributes values in the Angular controllers are mandatory. Once this attribute (and its value) has been removed, the function that writes the code and makes the validation mandatory is no longer necessary for the library and, with that, it was removed. The same removal behavior was observed in three other library functions (*identifyDeps*, *addUnitDependency,* and *isEmptyString*).

The same behavior was observed in uuid and Plivo-node. However, in the gulp-cccr, express-ifttt, jade, xml2js, and Node-Browserify libraries, the relevant factor was the lack of direct code coverage of the removed functions. To illustrate the situation, we will use the uuid library as an example.

Inside uuid, the Optimizer has found two cases (*fromURN* and *fromBytes* functions) where there was a problem regarding the absence of coverage in the tests, that is, at no

```
UUIDjs.fromBinary = function (binary) {
    var ints = [];
    for (var i = 0; i < binary.length; i++) {
        ints[i] = binary.charCodeAt(i);
      if (ints[i] > 255 || ints[i] < 0) {
          throw new Error('Unexpected byte in binary data.');
      }
    }
    return UUIDjs.fromBytes(ints);
};
```

```
UUIDjs.fromBinary = function () {
};
```

Fig. 3 The *fromBinary* function original code (top) compared to optimized code (down)

time these functions are exercised during the unit tests execution. So, it was possible to remove them and still succeed in running the tests.

This fact helps in to confirm that the unit tests coverage directly affects the optimization process results. There is also evidence contributing to show a tester (or developer) characteristics in their test cases that need attention because they do not exercise a reasonable source code area.

Another compelling case within the uuid library is the *fromBinary* function that is fully covered by the unit tests. All its code has been removed, and only its declaration statement is left as shown in Fig. 3. Even its input parameter (binary) has been deleted. Even with the tests exercising this function (it is performed once during the tests), it was possible to remove all its behavior because the tests do not execute any verification on the function's return value: it is only fired indirectly by the tests of other functions. Therefore, only its public interface (its signature) must exist so that the tests do not fail. Such a result reinforces that this type of optimization can also support testers by showing alternative ways of testing or even creating some quality metrics for them.

All libraries have had positive results, that is, the Optimizer was able to find variations of the original code that run in less time where it is possible to realize that the altered or removed statement directly affects the code execution of its function or library. Typical examples of modifications include unused variable declarations or the removal of their initialization values (which are not used), parameters removed in internal function calls and others. These results show the importance of a good design of tests cases to subsidize the genetic improvement, as well as the need for the review of a software engineer after the optimization process. All changes produced by the Optimizer were confirmed after manual analysis.

Also, the presented results reinforce the need for an experimental plan for genetic improvement unique to JavaScript. It should contemplate the construction of an improved model applied to library functions (in its fourth version, the Optimizer acts on all the library code every round of optimization), and rank the most used functions in the test cases to guide their selection during the optimization process.

7 Threats to validity

The validity of the results observed through an experimental study should be addressed in all phases of its life cycle, from planning to analysis. This section outlines the major threats to validity regarding our findings.

A vital source of threats is the difficulty of setting up the libraries used in the experiments. To add a new library to Optimizer, we first needed to measure its test suite statement coverage. Measuring coverage of test suites designed for JavaScript programs is not a standardized procedure, that is, we lack a well-known "recipe" to follow. Once the test suite statement coverage is identified (> 90%), it is necessary to locally setup the library environment and its dependencies to allow the Optimizer to make changes in the source code and execute the test suites.

Though some automation should support it, the process of setting up new libraries and determining their test suite statement coverage usually requires the manual developer intervention. It subjects experimentation to the researcher intervention and, therefore, requires the researchers to follow strict rules (clone repository, install and configure the Istanbul tool[7] and extract coverage percent's) for the sake of consistent replication and generalization (external validity).

Finally, we are interested in a qualitative analysis of the optimized libraries. In Section 5, objective improvement measures (see Table 2) were given far less attention than the specific changes proposed to some libraries in the discussion following Table 2. Therefore, we have not used statistical tests to assert whether the results of independent optimization rounds were due to chance (conclusion/construct validity). We intend to apply such tests in the future to compare the power of different optimization algorithms, but in the sense of results attained and the time required to find these results.

8 Lesson learned

During the Optimizer construction of the some essential lessons were learned. Some of them could have led the research to a premature end, whether a solution for that thread was not possible in viable time. They were someway depicted in Sections 5 and 6.

8.1 Correct choice of language interpretation engine

Choosing the right interpretation engine is one of the initial decisions when one is studying interpreted languages. Nevertheless, one of the most critical decisions. The criteria that should be observed for the choice are (i) compatibility with the correct version of the code of the subjects chosen for observation and (ii) the performance required to observe thousands of variations of this code. The troubleshooting about this item is discussed in Sections 5.2 and 5.3.

8.2 Representation used for code variants (solutions)

Representing a solution is a common problem in SBSE. Therefore, there is a small trap in this decision. Using a standard representation in other searches, such as code lines or even AST of code can lead to very high memory consumption, as seen in Section 5. Le Goues et al. (2012) propose using a *Patch representation* to reduce memory consumption for genetic algorithms targeting automated bug fixing. This type of representation might reduce significantly the memory consumed by the optimization process of this kind of research. The troubleshooting about this item is discussed in Sections 5.4, 5.5 and 5.6.

8.3 Tests quality

The tests percent coverage criteria that were used to select the subjects was considered necessary to prevent the Optimizer from removing "correct" instructions, that is, instructions that the test exercise them. However, we do not consider analyzing the quality of these tests as a selection criterion. After analyzing some previous Optimizer results, we noticed that covered instructions and even whole functions were removed entirely. This behavior is directly associated with the perceived quality of the tests in question. One point to be observed in this type of research is how the test design has, in addition to completeness (measure in coverage percent), quality.

We did not find a quality metric of tests that could support us in this decision and, therefore, only in the analysis of the previous results it was possible to revise the code of the tests in the perspective of the modification produced by the Optimizer. The troubleshooting about this item is discussed along with all Section 6.

8.4 Environment for running tests

Finally, and not the least important, we have the support for unit testing. In some more traditional languages, this type of support is often native or embedded in theirs IDEs. In the case of JavaScript is not the case. Each project chooses a library or writes on its own the necessary support, adding or creating components for its purpose. Some libraries need browser testing for the complete solution of your tests. So every case is a case. This issue almost impacts observation of subjects in this research, since for each new library a significant development effort was required, as seen in Section 5. Therefore, if the language does not have native support or a way of performing the tests that are common, it is necessary to include a selection criterion of the subjects of the experiment in order to mitigate the risks that cannot observe the results.

8.5 Some other situations

Some of the lessons learned in the construction of the Optimizer were classified as specific to the development of the Optimizer. For example, platform switching from the typical environment to the HPC environment or the absent of libraries requirements/documents. These two and other situations are described in detail in Section 5 of the text.

9 Conclusion

Genetic improvement can bring positive results even outside the context of traditional software. This paper reports our experience in applying GI in the context of JavaScript. The challenge of planning and executing an extensive and automatic experimental study, the application of automatic code optimization in JavaScript programs and the need for supercomputing resources reinforce the importance of the experimental plan.

Over 2 years, using four different programming languages, we constructed an Optimizer, a relevant experimental plan instrument. The fourth version (June / 2016) allows you to apply genetic improvement to a wide variety of JavaScript libraries. All of its development was done in Typescript, using the NodeJs runtime.

The use of HPC was essential for conducting these automatic experiments, due to the need for maintaining hundreds of original code variations. It was not possible to

predict accurately at the planning beginning the real study needs, such as the use of high-performance computing to observe the results. In particular, it is clear that a more accurate estimate of the required computational resources is key to the success of studies of this nature. One of the possible factors that did not allow us to have an accurate forecast of the necessary resources comes from the lack of related work applying genetic improvement in interpreted languages such as JavaScript. Without the memory and processing resources available in LOBOC, it would not be possible to observe the behaviors in libraries of this magnitude.

The analysis of data resulted from the thirteen automatic experiments indicates a new perspective: can GI in the private code of specific functions yield better results? Some related work (Harman et al. 2012; Arcuri et al. 2008; Petke et al. 2013) have already performed GI experiments focused on the expressly selected code of some software functions or classes. However, there is no clear criterion for the selection of functions for optimization, and a new set of experiments is needed to looking for a set of specific JavaScript criteria for select functions or code parts.

This research is unprecedented when applying genetic improvement in JavaScript. However, its novelty is amplified by using a supercomputer such as the LOBOC, which allowed observing the results presented in Section 5 for the first time. It brought definite indications of the application of genetic improvement in JavaScript, indicating the possibility of reducing the execution time of the thirteen analyzed libraries.

The contributions of this article are in the sharing of knowledge to apply genetic improvement in a new scenario (JavaScript). Observing positive results of automatic code improvement outside the domain of Object Oriented languages contributes directly to the advancement of this type of technique in Software Engineering. The search continues to locate source code improvement standards based on analysis of the results generated by the Optimizer, the definition of genetic operators specific to JavaScript, and the creation of plug-ins for IDEs that aim to offer automatic refactoring options for Developers.

10 Endnotes

[1] https://github.com/mishoo/UglifyJS2

[2] https://www.npmjs.com/

[3] https://cs.gmu.edu/~eclab/projects/ecj/

[4] http://www.antlr.org/

[5] https://github.com/paulbartrum/jurassic

[6] https://github.com/ffarzat/JavaScriptHeuristicOptmizer

[7] https://istanbul.js.org/

Acknowledgments

The authors would like to thank *Núcleo Avançado de Computação de Alto Desempenho* at COPPE/UFRJ, CAPES and CNPq for their support. Prof. Barros and Prof. Travassos are CNPq Researchers.

Funding

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

Availability of data and materials

Please contact the first author for data requests.

Authors' contributions

FF performed the experiments and analyses while supervised by GT and MB. MB and GT repeated the analyses with a revision bias, proposed and supervise corrections. FF, MB, and GT jointly wrote the text. All authors read and approved the final manuscript.

Competing interests

The authors declare that they have no competing interests.

Publisher's Note

Author details

[1]Computers and System Engineering Department, COPPE/UFRJ, Cx Postal 68501, Cidade Universitária, Rio de Janeiro, RJ, Brazil. [2]Post-Graduate Information Systems Department, PPGI/UNIRIO, Av. Pasteur 458, Urca, Rio de Janeiro, RJ, Brazil.

Received: 20 March 2018 Accepted: 19 September 2018
Published online: 06 October 2018

References

Anderson C, Giannini P, Drossopoulou S (2005) Towards type inference for JavaScript. ECOOP 2005-Object-Oriented. http://doi.org/10.1007/11531142

Anderson CL, Drossopoulou S (2006) Type inference for Javascript. Doctoral dissertation, Imperial College, London, UK

Arcuri A, White DR, Clark J, Yao X (2008) Multi-objective improvement of software using coevolution and smart seeding. In Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics). http://doi.org/10.1007/978-3-540-89694-4_7

Bierman G, Abadi M, Torgersen M (2014) Understanding typescript. Proceedings of the 23rd European Conference on Object-Oriented Programming, Genova, IT

Cody-Kenny B, Lopez EG, Barrett S (2015) locoGP: Improving Performance by Genetic Programming Java Source Code. In Genetic Improvement 2015 Workshop. http://doi.org/doi:10.1145/2739482.2768419

de Almeida Farzat F, Travassos G, de Oliveira Barros M (2017) Desafios para o Planejamento e Execução de Experimentos Utilizando Melhoramento Genético em JavaScript. In: XX Ibero-American Conference on Software Engineering. p. 525–538

Harman M, Jia Y, Langdon WB (2014) Babel Pidgin: SBSE can grow and graft entirely new functionality into a real world system. In Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics). http://doi.org/10.1007/978-3-319-09940-8_19

Harman M, Langdon WB, Jia Y, White DR, Arcuri A, Clark JA (2012) The GISMOE challenge: constructing the pareto program surface using genetic programming to find better programs (keynote paper). In: Proceedings of the 27th IEEE/ACM International Conference on Automated Software Engineering (ASE'12). http://doi.org/10.1145/2351676.2351678

Jensen SH, Møller A, Thiemann P (2009) Type analysis for JavaScript. In Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics). http://doi.org/10.1007/978-3-642-03237-0_17

Le Goues C, Dewey-Vogt M, Forrest S, Weimer W (2012) A systematic study of automated program repair: Fixing 55 out of 105 bugs for $8 each. In Proceedings - International Conference on Software Engineering. http://doi.org/10.1109/ICSE.2012.6227211

Orlov M, Sipper M (2011) Flight of the FINCH through the Java wilderness. IEEE Transactions Evolutionary Computation 15(2): 166–182

Fountoukis SG and Chatzistavrou DT (2009). Pattern Based Object Oriented Software Systems Design for High Performance Computing. https://www.semanticscholar.org/paper/Pattern-Based-Object-Oriented-Software-Systems-for-Fountoukis-CHATZISTAVROU/2ccab1ae9af32cc0a1d109becb6d93928fb1fb91?tab=abstract

Overbey J, Xanthos S, Johnson R, Foote B (2005) Refactorings for Fortran and high-performance computing. In Proceedings of the second international workshop on Software engineering for high performance computing system applications - SE-HPCS '05. http://doi.org/10.1145/1145319.1145331

Petke J, Haraldsson SO, Harman M, Langdon WB, White DR, Woodward JR (2018) Genetic Improvement of Software: A Comprehensive Survey. IEEE Transactions on Evolutionary Computation. http://doi.org/10.1109/TEVC.2017.2693219

Petke J, Langdon WB, Harman M (2013) Applying genetic improvement to MINISAT. Proceedings of the Symposium on 6th Search Based Software Engineering, St Petersburgh, RU

Marques-Silva J, Lynce I, Malik S (2009) Conflict-driven clause learning SAT solvers. Frontiers in Artificial Intelligence and Applications. http://doi.org/10.3233/978-1-58603-929-5-131

White DR (2010) Genetic Programming for Low-Resource Systems. PhD Thesis, Dapartment of Computer Science, University of York, UK

Wohlin C, Runeson P, Höst M, Ohlsson MC, Regnell B, Wesslén A (2012) Experimentation in software engineering. Experimentation in Software Engineering. http://doi.org/10.1007/978-3-642-29044-2