OTIMIZAÇÃO PARA REDUZIR O TAMANHO DE CÓDIGO-FONTE JAVASCRIPT

Fabio de Almeida Farzat

Tese de Doutorado apresentada ao Programa de Pós-graduação em Engenharia de Sistemas e Computação, COPPE, da Universidade Federal do Rio de Janeiro, como parte dos requisitos necessários à obtenção do título de Doutor em Engenharia de Sistemas e Computação.

Orientadores: Guilherme Horta Travassos
Márcio de Oliveira Barros

Rio de Janeiro
Dezembro de 2018

OTIMIZAÇÃO PARA REDUZIR O TAMANHO DE CÓDIGO-FONTE JAVASCRIPT

Fabio de Almeida Farzat

TESE SUBMETIDA AO CORPO DOCENTE DO INSTITUTO ALBERTO LUIZ COIMBRA DE PÓS-GRADUAÇÃO E PESQUISA DE ENGENHARIA (COPPE) DA UNIVERSIDADE FEDERAL DO RIO DE JANEIRO COMO PARTE DOS REQUISITOS NECESSÁRIOS PARA A OBTENÇÃO DO GRAU DE DOUTOR EM CIÊNCIAS EM ENGENHARIA DE SISTEMAS E COMPUTAÇÃO.

Examinada por:

_____

Prof. Guilherme Horta Travassos, D.Sc

_____

Prof. Márcio de Oliveira Barros, D.Sc

_____

Prof[a]. Silvia Regina Vergilio, D.Sc

_____

Prof. Leonardo Gresta Paulino Murta, D.Sc

_____

Prof. Paulo de Figueiredo Pires, D.Sc

_____

Prof. Toacy Cavalcante de Oliveira, D.Sc

RIO DE JANEIRO, RJ - BRASIL

DEZEMBRO DE 2018

Farzat, Fabio de Almeida

Otimização para reduzir o tamanho de código-fonte JavaScript / Fabio de Almeida Farzat – Rio de Janeiro: UFRJ/COPPE, 2018.

XIII, 76 p.: il.; 29,7 cm.

Orientador: Guilherme Horta Travassos

Márcio de Oliveira Barros

Tese (doutorado) – UFRJ/COPPE/Programa de Engenharia de Sistemas e Computação, 2018.

Referências Bibliográficas: p. 70-73.

1. Otimização de código-fonte. 2. Engenharia de software baseada em busca. 3. Heurísticas. 4. Melhoramento Genético. 5. Programação genética.

I. Travassos, Guilherme de Horta, *et al.* II. Universidade Federal do Rio de Janeiro, COPPE, Programa de Engenharia de Sistemas e Computação III. Título.

À minha família, pelo constante apoio, compreensão e amor.

Agradecimentos

Em primeiro lugar à minha família pelo apoio, amor e incentivo neste longo período em que estive praticamente ausente do convívio familiar. Em especial à minha mãe, Aleuda Gomes de Almeida, pelo incondicional apoio, desde criança, para os meus estudos, crescimento como profissional e como pessoa.

Ao meu orientador, prof. Guilherme Horta Travassos, que além da orientação em si, me ensinou muito sobre o ponto de vista científico, ético e profissional, com os ensinamentos e, principalmente, com o exemplo. Cresci bastante como pessoa e como profissional durante os anos do doutorado em grande parte com contribuição sua. Sou-lhe, sinceramente, muito grato por tudo.

Ao meu orientador, prof. Márcio de Oliveira Barros, que me conduziu desde o mestrado até o doutorado, sempre incentivando e buscando extrair o máximo de qualidade e aprendizado possível. Sem sua participação, ensinamento e paciência o trabalho não teria chegado até a completude e com o nível de qualidade em que chegou. Não tenho como agradecer por tudo que já aprendi com o senhor.

Aos professores Silvia Regina Vergilio, Leonardo Gresta Paulino Murta, Paulo de Figueiredo Pires e Toacy Cavalcante de Oliveira por participarem da minha banca de defesa de doutorado e oferecerem contribuições para sua melhoria.

Aos meus amigos da COPPE, em especial do Grupo ESE, pela companhia, discussões, contribuições e bons momentos durantes esses anos.

Ao meu irmão caçula, André de Almeida Farzat, pela contribuição com o necessário para o desenvolvimento do ferramental de apoio. Sem a sua ajuda nos benditos “Call Back” e “Promise”, o ferramental levaria muito mais tempo do que foi o necessário para ficar concluído.

À Coordenação de Aperfeiçoamento de Pessoal de Nível Superior (CAPES) pela bolsa de estudos de Doutorado.

Ao Núcleo de Computação de Alto Desempenho (NACAD) pelo uso do Lobo Carneiro nos experimentos da tese.

Resumo da Tese apresentada à COPPE/UFRJ como parte dos requisitos necessários para a obtenção do grau de Doutor em Ciências (D.Sc.)

OTIMIZAÇÃO PARA REDUZIR O TAMANHO DE CÓDIGO-FONTE JAVASCRIPT

Fabio de Almeida Farzat

Dezembro/2018

Orientadores: Guilherme de Horta Travassos

Márcio de Oliveira Barros

Programa: Engenharia de Sistemas e Computação

Esta Tese aborda o problema de otimização de tempo de carga de software, especificamente software escrito na linguagem de programação JavaScript, uma linguagem interpretada, baseada em objetos e amplamente utilizada no desenvolvimento de aplicativos e sistemas para a internet. Estudos experimentais foram projetados para avaliar a hipótese de que técnicas heurísticas já aplicadas com sucesso em linguagens orientadas a objeto poderiam ter resultados positivos na redução do tempo de carga de programas escritos em JavaScript. Para tanto, um ferramental que permitisse observar a aplicação de heurísticas selecionadas em programas JavaScript foi construído e executado em um ambiente de computação de alto desempenho. Os resultados dos estudos preliminares foram utilizados para criar um procedimento de busca que varre o código JavaScript criando variantes do programa que sejam menores e passem em todos os casos de teste do programa original. Aplicamos este procedimento a 19 programas JavaScript, variando de 92 a 15.602 linhas de código, e observamos reduções de 0,2% a 73,8% do código original, bem como uma relação entre a qualidade do conjunto de casos de testes e a capacidade de reduzir o tamanho dos programas.

Abstract of Thesis presented to COPPE/UFRJ as a partial fulfillment of the requirements for the degree of Doctor of Science (D.Sc.)

EVOLVING JAVASCRIPT CODE TO REDUCE CODE SIZE

Fabio de Almeida Farzat

December/2018

Advisors: Guilherme de Horta Travassos

Márcio de Oliveira Barros

Department: Systems Engineering and Computer Science

JavaScript is one of the most used programming languages for front-end development of Web application. The increase in complexity of front-end features brings concerns about performance, especially the load and execution time of JavaScript code. To reduce the size of JavaScript programs and, therefore, the time required to load and execute these programs in the front-end of Web applications. To characterize the variants of JavaScript programs and use this information to build a search procedure that scans such variants for smaller implementations that pass all test cases. We applied this procedure 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 its size.

ÍNDICE

ÍNDICE DE FIGURAS

ÍNDICE DE TABELAS

1 Introdução

Neste capítulo, apresentamos o problema e o contexto envolvido nesta Tese, bem como as questões de pesquisa que apoiam a investigação. Além disso, estabelecemos os objetivos a serem cumpridos para responder às questões de pesquisa e como eles serão atingidos por meio de uma metodologia baseada em evidências.

1.1 Introdução

A demanda por sistemas de software por parte das organizações tem aumentado significativamente nas últimas décadas. Esse fato se reflete no número de sistemas de software que o mercado tem produzido, além dos sistemas de código aberto. Esses novos sistemas precisam atender a requisitos funcionais cada vez mais complexos, integrando-se com outros sistemas, prevendo adaptações e mudanças em um intervalo de tempo cada vez menor. Esses fatores contribuem para o aumento da complexidade do software.

Além dos requisitos funcionais, existem outros requisitos que podem influenciar na complexidade do software. Em vários projetos, os recursos disponíveis para a execução dos sistemas podem ser restritos. Exemplos de ambientes de execução restritos são os aplicativos para celular, aplicativos web e sistemas embarcados. Nesses tipos de sistema, o contexto de execução é limitado em várias perspectivas, indo do uso do software em um dispositivo sem teclado ou com tela de tamanho reduzido até o hardware com capacidade de processamento limitada e autonomia energética reduzida. Chamamos essas restrições de requisitos não funcionais, ou seja, requisitos que não tem relação direta com o objetivo do software, mas que precisam ser atendidas para seu pleno funcionamento (White, Arcuri, & Clark, 2011).

Produzir software para atender aos requisitos não funcionais é uma tarefa difícil para os programadores, principalmente porque suas preocupações normalmente estão nos requisitos funcionais, por muitas vezes confusos ou mal documentados. Há também os fatores externos à função dos desenvolvedores e típicos de projetos de software, como prazo e recursos financeiros limitados. Em um cenário assim, muitas das soluções propostas para o desenvolvimento do software não são avaliadas sob os aspectos não funcionais, mas apenas por seu comportamento funcional.

Nesse sentido, existem técnicas e práticas de Engenharia de Software para minimizar o risco de problemas não funcionais. Compiladores, por exemplo, podem realizar pequenas transformações no código-objeto de um sistema com o objetivo de melhorar o seu tempo de execução ou reduzir o volume de recursos consumidos ao executar o código. No entanto, os compiladores aplicam um conjunto fixo de transformações que preservam a semântica do código original e essa restrição impede que eles tratem todas as nuances possíveis do problema como, por exemplo, a exclusão de instruções não utilizadas (Triantafyllis, Vachharajani, Vachharajani, & August, 2003).

Testes também podem ser utilizados para verificar o atendimento aos requisitos não funcionais do software. Um bom projeto de testes pode revelar se o software teve seu comportamento modificado em relação à determinada característica não funcional devido às alterações recentes no código fonte. Porém, projetos de teste são geralmente caros e não é comum que tenham objetivos claros quanto às características não funcionais do software (Glinz, 2007). Temos então um cenário em que sistemas são modificados frequentemente e estas alterações podem afetar aspectos da sua execução que por vezes só são percebidos tarde demais, quando o sistema já está em uso e a qualidade dos produtos da empresa já está comprometida.

Esse quadro se torna ainda mais grave quando o trazemos para o contexto de softwares que possuem restrições para sua execução. Um aplicativo Web executando em um navegador é muito sensível a aspectos não funcionais, uma vez que o computador hospedeiro pode variar de configuração, exigindo que o software seja executado em um processador mais lento ou que tenha acesso a menos memória. Além disso, nesses aplicativos a linguagem de programação mais comumente utilizada é o JavaScript (Richards, Lebresne, Burg, & Vitek, 2010), que é uma linguagem interpretada, ou seja, ela precisa ser traduzida para linguagem de máquina em tempo de execução.

Em 1995, a Netscape lançou o JavaScript como sendo uma “linguagem de fácil uso, baseada em objetos, para o desenvolvimento de scripts”, com o objetivo de permitir que não programadores estendessem o comportamento de páginas através de código que executasse do lado do cliente, no próprio navegador. Desde então, JavaScript tornou-se a linguagem de *script* mais comum no lado cliente de aplicativos Web. Por outro lado, a evolução dos sistemas disponíveis na Internet, que exigiu um modelo de interação mais eficiente entre os lados cliente e servidor destes sistemas, e a capacidade de manter a parte da lógica dos sistemas dentro de páginas que executam no lado cliente, levou os desenvolvedores a escrever grandes programas em uma linguagem que não foi concebida para a programação em larga escala (Jensen, Møller, & Thiemann, 2009).

Porém, JavaScript não evoluiu apenas nesse contexto. Atualmente, a linguagem também pode ser encontrada em utilização em cenários fora do navegador, conforme mostra a Figura 1. Aplicativos em *smartphones* e mesmo servidores fazem uso de JavaScript para criação de sistemas complexos. Quando o ambiente de execução é uma plataforma móvel (iOS, Android, Windows Mobile, entre outras), esses sistemas são chamados de híbridos (Wasserman, 2010). Sistemas híbridos fazem uso de navegadores específicos[1] de cada plataforma, alterando sua aparência e comportamento padrão, e permitindo que o JavaScript interaja com os recursos do dispositivo e parecendo um aplicativo nativo (escrito e compilado na linguagem nativa) daquela plataforma.

Figura 1 Contextos de execução do JavaScript

Ao contrário de outras linguagens tradicionais, JavaScript é *object-based*, ou seja, usa protótipos de objeto para compor classes e permitir herança (Jensen et al., 2009). Esses objetos são mapeamentos de *strings* (nomes de propriedades) para valores. Em geral, propriedades podem ser adicionadas e removidas destes objetos durante a execução e seus nomes podem ser calculados dinamicamente. Além disso, os valores de variáveis são livremente convertidos a partir de um tipo para outro tipo, com poucas exceções em que não se aplica a conversão automática. No caso dos valores especiais, *null* e *undefined* não podem ser convertidos em objetos e apenas os valores do tipo função (ponteiros de função ou todo o corpo de uma função) podem ser invocados. Essas características trazem novos desafios no que tange à análise estática de código-fonte, além de contribuírem diretamente para o aumento da complexidade estrutural dos aplicativos construídos usando JavaScript.

Com o passar dos anos, novas tecnologias foram criadas usando o JavaScript. Ajax (Mesbah & Van Deursen, 2007), WebSockets (Ubl & Eiji, 2010) e *Single Page Applications* (Mesbah & Van Deursen, 2007) são exemplos da evolução dos usos desta linguagem. Surgiu também um grande número de bibliotecas que suportam o uso dessas tecnologias, tais como

[1] https://github.com/electron/electron

JQuery[2], AngularJS[3] e React[4]. Em sua maioria, essas bibliotecas são programas grandes quando consideramos o objetivo original do programa JavaScript se manter simples e de fácil uso. Apenas como referência, JQuery é uma biblioteca mundialmente conhecida, possui 11.026 linhas de código e 294.199 caracteres em um único arquivo.

O uso de grandes bibliotecas gera uma dicotomia com um aspecto não-funcional importante do JavaScript: o tamanho físico do código-fonte em disco, especialmente quando o programa não é executado localmente. Quando a execução acontece no navegador, o arquivo com o código-fonte é sempre transmitido para o cliente antes de sua execução. Portanto, quanto maior for o arquivo, maior será o tempo de transferência do mesmo e, consequentemente, maior consumo de energia. Se um programa depende de bibliotecas para que possa ser executado, estas bibliotecas também precisam ser transferidas, carregadas e interpretadas no cliente, ações que mais uma vez dependem do tamanho do programa JavaScript.

1.2 Definição do Problema

Existem técnicas de redução do tamanho do código-fonte específicas para JavaScript, como a minificação. A minificação de código JavaScript consiste em uma série de alterações controladas com o objetivo de reduzir o tamanho do código-fonte que será interpretado e executado no navegador (Jensen et al., 2009; Richards et al., 2010). A técnica reduz os nomes de variáveis, métodos e objetos, além de remover espaços em branco em excesso e comentários.

O processo é amplamente utilizado em sistemas web e traz ganhos no tempo de transmissão do código. O processo de minificação também é usado para dificultar a cópia do código, uma vez que o mesmo se torna praticamente ilegível para um ser humano. Porém, existe um ponto negativo: não é possível voltar ao código original uma vez que o mesmo é minificado. Além disso, a minificação é um processo essencialmente léxico, que remove separadores desnecessários e reduz os identificadores para reduzir o tamanho do código-fonte. A minificação não analisa se determinadas partes do código são realmente necessárias para a correta execução do código-fonte.

Em decorrência da intensificação do uso de JavaScript nas últimas décadas, é preciso encontrar um conjunto de alterações que aprimorem um código-fonte escrito nesta linguagem sob a perspectiva de tamanho. Uma estratégia neste sentido consiste em desenvolver um

[2] https://jquery.com/

[3] https://angularjs.org/

[4] https://facebook.github.io/react/

procedimento de busca heurística para encontrar um conjunto de alterações no código-fonte que reduzam o seu tamanho sem alterar o seu comportamento. Este conjunto de alterações seria, em seguida avaliado por um engenheiro de software e, se considerado correto e válido, incorporado ao código-fonte do software que está sendo otimizado.

Assim, seria possível reduzir o número de instruções de uma biblioteca em JavaScript, reduzindo o seu tempo de processamento e de carga nos navegadores. Seria possível também, indiretamente, reduzir o consumo de energia e de banda de Internet para o uso do mesmo. Se aplicado em escala global, o impacto da redução de algumas poucas instruções em bibliotecas muito utilizadas pode ser muito grande. Apenas a título de exemplo, algumas bibliotecas estudadas nessa pesquisa superam os 10 milhões de novos *downloads* por mês.

Desta forma, o ferramental proposto nesse trabalho de pesquisa foi projetado para executar, se necessário, sob um processo de integração contínua, onde a cada nova alteração a otimização é executada e, se possível, um conjunto de alterações para a avaliação do engenheiro de software responsável pelo software é proposto. Esse conjunto de alterações é avaliado como um novo *commit*, onde é possível fazer a verificação através de diferenças textuais (*patches*), ou seja, comparar dois textos e visualizar suas diferenças. Esse processo de comparação representa uma prática comum em projetos de desenvolvimento de software que estejam sob um sistema de gerenciamento de versões.

1.3 Objetivo da pesquisa

O cenário exposto nas seções anteriores caracteriza um problema que nessa Tese foi nomeado como otimização de código JavaScript para reduzir o tempo de carga.

Esta Tese tem como objetivo de investigar e desenvolver uma técnica de busca heurística para a otimização de código-fonte aplicável a programas escritos na linguagem JavaScript visando encontrar alterações no código-fonte que reduzam o seu tamanho e, indiretamente, melhorem o tempo de transferência e carga destes programas. Como efeito colateral, entendemos que os resultados podem contribuir para a redução do consumo de energia de processamento, embora esta observação, medição e avaliação não façam parte do escopo da Tese. Entretanto, este trabalho apresenta um modelo formal para o problema e um método heurístico de busca para encontrar as alterações supracitadas.

Desta forma, este trabalho caracteriza o problema e apresenta uma proposta de solução que consiste de uma técnica e uma ferramenta capaz de processar arquivos escritos na linguagem JavaScript com seus roteiros e casos de testes unitários, aplicar um conjunto de

transformações sobre o código-fonte, avaliar estas transformações sob a perspectiva de tamanho observado pelo número de instruções resultantes e propor modificações no código original. Para avaliar a aplicação de tal procedimento, foram selecionadas 19 bibliotecas escritas em JavaScript usualmente utilizadas em diferentes aplicações pelos desenvolvedores. O critério de seleção levou em conta o tamanho da biblioteca (em linhas de código) e o fato de possuírem testes unitários abrangentes (com cobertura de código superior a 90%).

1.4 Questões de pesquisa

As questões de pesquisa a seguir foram derivadas do problema apresentado na Seção 1.3, estabelecendo o escopo e as principais direções de investigação.

- RQ1: É possível reduzir o tamanho de um código-fonte escrito em JavaScript usando técnicas heurísticas de busca?

  Pesquisas anteriores encontraram resultados positivos na otimização heurística de código fonte, conforme será abordado em detalhes no Capítulo 2. Essa questão de pesquisa avalia se o mesmo resultado positivo pode ser observado em JavaScript.

- RQ2: Quais são as características do espaço de busca por variantes de programas JavaScript onde foram aplicadas operações visando obter reduções de tamanho?

  Um dos maiores desafios em aplicar técnicas de busca heurística é o entendimento do espaço de busca das soluções, conforme será discutido no Capítulo 3. Compreender a organização desse espaço é um dos critérios para escolher os algoritmos, a representação da solução e a representação de vizinhança adequados ao problema. Essa questão de pesquisa visa caracterizar o espaço de busca por variantes de programas JavaScript.

- RQ3: Que tipos de alterações são realizadas por um procedimento de busca heurística para reduzir o tamanho de programas JavaScript?

  Nesta questão de pesquisa analisamos os tipos de modificações produzidas por algoritmos heurísticos de busca com o objetivo de classificar as mudanças em grupos que possam ser generalizados em recomendações aplicáveis *a priori* por desenvolvedores de software que utilizem a linguagem de programação JavaScript. Os tipos de alteração serão analisados no Capítulo 4 deste documento.

1.5 Contribuições esperadas

O objetivo desta Tese é aplicar técnicas heurísticas de melhoramento de código-fonte sobre programas escritos na linguagem JavaScript para encontrar alterações no código-fonte que reduzam o tamanho (em número de instruções, consequentemente, em número de caracteres) destes programas. Entre as contribuições esperadas da Tese, destacam-se:

- Uma técnica heurística de busca para reduzir o tamanho de código-fonte escrito em JavaScript mantendo a sua funcionalidade;
- Um ferramental capaz de processar código-fonte JavaScript e aplicar a otimização proposta;
- A caracterização do espaço de busca por variantes de programas JavaScript que sejam equivalentes funcionais e tenham tamanho reduzido;
- Um plano experimental para validação da técnica heurística de busca proposta, comparando-a com alternativas mais simples e mais complexas;
- Um conjunto de evidências sobre o uso de melhoramento genético como técnica para otimização de tamanho de código-fonte em JavaScript.

1.6 Metodologia da pesquisa

A metodologia de pesquisa para este trabalho começa com um estudo bibliográfico sobre técnicas heurísticas para otimização de código-fonte, com o objetivo de apoiar a criação de um ferramental para a aplicação dessas mesmas técnicas em JavaScript (Capítulo 3). Após a conclusão do ferramental, uma investigação foi conduzida para caracterizar o espaço de busca do problema de redução de código-fonte em JavaScript e permitir escolher uma estratégia de busca heurística adequada para o problema (Capítulo 3). Por fim, tivemos a construção de um plano experimental para observação dos resultados ao aplicar a otimização em JavaScript utilizando o ferramental construído (Capítulo 4). A Figura 2 apresenta os estágios que compuseram a metodologia da pesquisa.

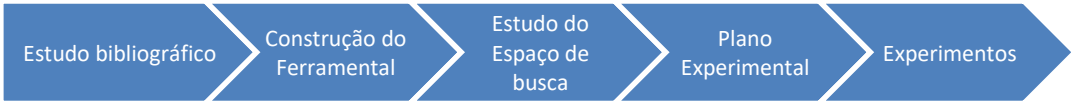

Figura 2 Metodologia da pesquisa utilizada na Tese

Antes da construção do ferramental, conduzimos uma pesquisa bibliográfica sobre técnicas de melhoramento genético de código-fonte e o ferramental utilizado em trabalhos

relacionados a este tema. O objetivo foi entender o estado da arte na área, descobrir que ferramentas existem e se poderiam ser utilizadas com programas JavaScript. De posse deste conhecimento, foi desenvolvido um Otimizador próprio para aplicação de técnicas de melhoramento genético em programas escritos em JavaScript (Capítulo 3, seção 3.2).

Quando a construção do ferramental foi concluída, conduzimos a orientação de um trabalho de conclusão de curso de graduação no qual foi realizado um estudo sobre programas escritos na linguagem JavaScript, visando caracterizar o tamanho, complexidade e o uso dos diferentes tipos de nós que compõem a gramática da linguagem de programação em programas JavaScript (Silva, D., Sobral, 2017). Essa pesquisa foi importante para o estudo de caracterização do espaço de busca discutido no Capítulo 3 (seção 3.3).

Em seguida, executamos um conjunto de experimentos com o intuito de evidenciar as características do espaço de busca, como a distribuição das soluções e os tipos de operadores aplicados para chegar a elas. Como parte destes experimentos, realizamos uma comparação entre algoritmos de busca local e algoritmos genéticos no que tange à sua capacidade de encontrar boas soluções nesse espaço de busca. Realizamos uma análise entre a frequência de uso dos tipos de nós da gramática JavaScript e a frequência de alteração ou remoção desses tipos de nós nas soluções encontradas pelos dois tipos de algoritmos, visando identificar padrões de alteração que reduzam o tamanho do código-fonte JavaScript. Os resultados provenientes desta análise permitiram estabelecer a granularidade da otimização e os critérios de vizinhança para uma busca local.

Em seguida, conduzimos uma avaliação da ordenação das funções para otimização de seu código de acordo com a frequência do seu uso pelos testes unitários, ou seja, a otimização priorizava a melhoria do código das funções mais utilizadas pelos casos de teste. Concluímos pela necessidade de otimizar o código por completo e, com isso, passamos para a construção de um plano experimental neste sentido. Por fim, executamos os experimentos sobre 19 bibliotecas e observamos os resultados da Otimização utilizando a configuração final da busca local proposta (Capítulo 4) em comparação com buscas mais simples (aleatória) e mais complexas (considerando todos os tipos de nós da gramática de programas JavaScript).

1.7 Estrutura do texto

Este trabalho está organizado em cinco capítulos. O primeiro capítulo compreende esta introdução. O segundo capítulo apresenta o ramo de pesquisa que aborda a otimização de código fonte de software sem a intervenção humana. Este capítulo aborda as técnicas de

melhoria mais comuns e discute o uso das heurísticas genética e local na busca de soluções para o problema. O capítulo ainda discute trabalhos relacionados o tema.

O terceiro capítulo descreve o desenvolvimento e a construção de um ferramental para observação da aplicação de técnicas heurísticas de otimização de código-fonte em JavaScript e um experimento com objetivo exploratório que visou traçar o perfil do espaço de busca na otimização de código-fonte JavaScript. O capítulo apresenta as características do problema, seu espaço de busca e o plano experimental projetado para explorá-lo.

O quarto capítulo apresenta a técnica proposta e discute os resultados obtidos na sua avaliação, separando-os em resultados qualitativos e quantitativos, nos quais são analisadas as modificações encontradas pela técnica proposta. Além disso, o capítulo aborda as ameaças à validade do experimento executado para analisar a técnica proposta.

Por fim, o quinto capítulo contém as considerações finais e contribuições dessa pesquisa.

2 Otimização automática de código fonte

Neste capítulo, apresentaremos o ramo da Engenharia de Software Orientada a Busca (Search Based Software Engineering) e seu sub-ramo de Refatoração de código, onde essa Tese está classificada. Algumas das principais técnicas e trabalhos relacionados a este contexto serão apresentados e discutidos.

2.1 Introdução

Heurísticas são técnicas que têm por finalidade a busca de soluções boas, próximas da solução ótima, para problemas combinatórios. Esse tipo de técnica visa baixar o custo computacional de criar soluções de qualidade, ainda que nem sempre seja encontrada a solução ótima para um determinado problema. Um ponto fraco desse tipo de abordagem é que não existem garantias da qualidade da solução gerada, nem mesmo sobre quão próxima ela está da solução ótima. Porém, em problemas cujo espaço de busca cresce rapidamente, para instâncias grandes e onde não se pode presumir a distribuição das soluções no espaço de busca, os métodos heurísticos são muito utilizados para encontrar soluções próximas ao ótimo (Smith-Miles & Lopes, 2012).

A Engenharia de Software Baseada em Busca (ou *Search Based Software Engineering*, SBSE) é um ramo da Engenharia de Software que trata problemas de desenvolvimento de software como problemas de otimização, ou seja, um modelo formal é definido para o problema de forma a permitir que várias soluções alternativas compatíveis com o modelo sejam descobertas por um processo de busca heurística que explore sistematicamente o espaço de todas as possíveis soluções para o problema (Mark Harman, Mansouri, & Zhang, 2012). Portanto, podemos resumir a SBSE como a aplicação de métodos heurísticos em problemas de Engenharia de Software.

O termo *Search Based Software Engineering* surgiu no trabalho de Harman e Jones (Mark Harman & Jones, 2001), que tinha como objetivo caracterizar a nova área de pesquisa. De fato, os problemas na área de Engenharia de Software são complexos e admitem várias soluções. Isso os torna um excelente alvo para a aplicação de métodos baseados em busca heurística. Ainda segundo Harman e Jones (Mark Harman & Jones, 2001), para reformular um problema de Engenharia de Software como um problema de otimização é necessário:

1. Representação: escolher uma forma adequada para definir a estrutura de dados que será usada para representar as possíveis soluções do problema;

2. Função objetivo: definir uma função que fará a avaliação das soluções encontradas e definirá um valor absoluto para cada uma, permitindo assim uma comparação entre as mesmas;

3. Operadores: nas técnicas de busca heurística é necessário aplicar distorções ou mudanças na solução original. Estas alterações são realizadas por operadores, que são configurados conforme cada busca heurística. A configuração de operadores apropriados é parte do problema de selecionar uma heurística adequada para um problema.

Portanto, para aplicar métodos de busca heurística em problemas da Engenharia de Software temos três esforços principais. Esse pode ser considerado o maior desafio em trabalhos da área, dado que não há um método ou técnica que identifique (Mark Harman, 2007) a representação ideal para um problema, assim como suas funções objetivo e os operadores que devem ser utilizados na otimização. Em cada problema o engenheiro de software precisa resolver antecipadamente (geralmente a partir do resultado de estudos experimentais) que representação, qual (ou quais) função objetivo e que operadores ele utilizará para abordar o problema sob análise.

Em especial, existe um ramo da Engenharia de Software Baseada em Busca que aborda o problema de otimizar o código fonte de um programa na perspectiva de um ou mais critérios específicos, tais como consumo de memória, tempo de execução, consumo de energia, entre outros (Petke et al., 2018). Esse trabalho de pesquisa está inserido nesse ramo, onde o objetivo é otimizar o código fonte de programas escritos em JavaScript na perspectiva de reduzir o seu tamanho. Até onde pudemos observar, essa pesquisa é pioneira em utilizar tais técnicas em código JavaScript. Este capítulo está dividido em seis seções, começando por esta introdução. A Seção 2.2 apresenta os principais algoritmos de busca heurística utilizados nesse trabalho. A Seção 2.3 apresenta o ramo da Engenharia de Software Baseada em Busca que trata de otimização de código fonte, apresentando um breve histórico e trabalhos relacionados. A Seção 2.4 apresenta a programação genética e seu histórico. A Seção 2.5 aborda o uso de programação genética em Engenharia de Software. Por fim, a Seção 2.6 faz uma análise consolidada das principais questões abordadas neste capítulo e apresenta suas conclusões.

2.2 Algoritmos heurísticos de busca

Algoritmos heurísticos de busca são técnicas que têm por finalidade a busca de soluções boas, próximas da solução ótima, para problemas combinatórios (Bäck & Schwefel, 1993). O

objetivo principal desse tipo de técnica é reduzir o custo computacional de encontrar soluções de qualidade, ainda que nem sempre seja encontrada a solução ótima. A literatura sobre técnicas de otimização exibe um grande número de heurísticas, assim como suas variações. Porém, a maior parte dos trabalhos de SBSE faz uso de um grupo relativamente limitado de heurísticas: busca aleatória, algoritmos gulosos, busca local e algoritmos evolutivos (Mark Harman, Mansouri, et al., 2012). Outras heurísticas, como Colônia de Formigas, *Simulated Annealing* e Nuvem de Partículas, são utilizadas com menor frequência e estão fora do escopo deste trabalho.

2.2.1 Algoritmos genéticos

Os algoritmos de busca heurística podem ser divididos em dois grupos: os algoritmos baseados em população e os algoritmos baseados em pontos. Os algoritmos baseados em população favorecem a exploração do vasto espaço de busca, investindo o poder de processamento do computador para avaliar a qualidade e evoluir um conjunto de indivíduos dispersos no espaço de busca de soluções possível para o problema. Algoritmos genéticos são exemplos típicos de algoritmos baseados em população (Goldberg, 1989). Esses algoritmos são úteis para problemas cujas funções objetivo podem ser rapidamente calculadas e onde a recombinação dos indivíduos componentes da população tende a levar a melhores indivíduos, ou seja, melhores soluções. No entanto, essas características não são facilmente reconhecíveis a priori e resultam de características do problema, representação da solução e uso de operadores customizados.

Algoritmos genéticos consistem em uma população de indivíduos que se reproduzem ao longo de várias gerações de acordo com a função objetivo definida. Os indivíduos mais fortes (ou seja, que exibem maior valor de função objetivo) têm mais chances de sobreviver, se reproduzir e passar as suas características genéticas para as gerações seguintes. Novos indivíduos, que vão formar as gerações seguintes ao longo do processo evolutivo, são resultado da aplicação sistemática de operadores. Os operadores mais comuns são o cruzamento e a mutação. O operador de cruzamento consiste em escolher duas soluções da população, trocar material genético entre elas e gerar uma (ou mais) novas soluções. Existem vários tipos de cruzamento possíveis em algoritmos genéticos. Porém, cada um deles depende do problema e da representação escolhida para a solução. Já no operador de mutação, uma das soluções componentes da população ou resultante do cruzamento é escolhida para sofrer uma mudança em seu material genético. Assim como no cruzamento, existem vários tipos possíveis de mutação que também dependem do problema e da representação escolhida para a solução. A Figura 3 ilustra o fluxo de execução de um algoritmo genético genérico. Ao evoluir uma

população de indivíduos independentes, os algoritmos genéticos podem analisar uma grande área do espaço de busca. Essa característica permite que eles escapem dos mínimos locais, tornando-os excelentes estratégias heurísticas para obter boas soluções para problemas de otimização combinatória.

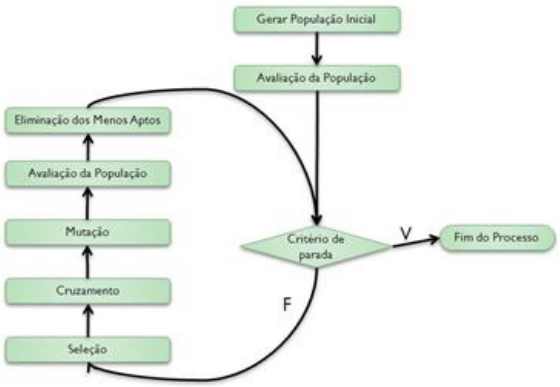

Figura 3 Exemplo do fluxo de execução de um algoritmo genético

2.2.2 Algoritmos de busca local

A busca local é uma classe de algoritmos baseados em pontos que usa uma sistemática para percorrer a vizinhança de um determinado indivíduo procurando por um vizinho melhor (com maior valor na função objetivo), se for possível encontrá-lo. Um vizinho de um determinado indivíduo A é o resultado da aplicação de uma mutação sobre A, como a alteração de uma característica individual. A vizinhança de um indivíduo A é definida pelo conjunto de vizinhos distintos que podem ser gerados pela aplicação de uma determinada mutação sobre A.

Quando um vizinho melhor é encontrado, ou seja, o valor da função objetivo é melhor do que o valor da mesma função para a solução de origem, o procedimento de pesquisa é repetido para examinar a vizinhança deste vizinho. Algoritmos de busca local favorecem a região da solução atual ao invés da exploração do espaço de busca, concentrando a investigação na região vizinha de um dado indivíduo e negligenciando o restante do espaço de busca.

Existem vários tipos de busca local. Um dos algoritmos mais usados em SBSE é o *Hill Climbing* (Mark Harman, 2007). O *Hill Climbing* é um algoritmo de busca local guloso que possui duas variantes principais: a subida mais íngreme e subida mais próxima (*Last-ascent Hill Climbing ou LAHC e First-ascent Hill Climbing*, ou FAHC, respectivamente). A diferença entre elas é que na subida mais íngreme todos os vizinhos são avaliados e a heurística se move para o melhor deles. Na versão da subida mais próxima, a heurística se move para o primeiro melhor vizinho encontrado. No trabalho de Hierons et al. (Carson, 2000) temos o detalhamento da implementação da subida mais íngreme, contendo os passos descritos na Figura 4.

```
1: S ← solução inicial
2: repita
3:   R ← Mutação(S)
4:   Se Qualidade(R) > Qualidade(S) então
5:     S ← R
6: até que S seja a solução ideal ou o tempo tenha se esgotado
7: devolva S
```

Figura 4 Descrição do algoritmo de Hill Climbing

Os algoritmos de busca local são métodos computacionalmente simples. Seu custo de execução é geralmente uma fração do custo de um algoritmo baseado em população. Eles tendem a ser eficientes e eficazes se partem de uma boa solução, ou seja, de uma solução que esteja próxima da solução ótima.

2.3 Refatoração de código fonte

Existem sete grandes áreas de pesquisa em Engenharia de Software baseada em Busca, a saber: especificação/requisitos, técnicas e ferramentas de *design*, validação de modelos e verificação de sistemas, testes e depuração, distribuição, manutenção, métricas e gerenciamento de projetos (Mark Harman & Jones, 2001). O ramo que trata a otimização de código fonte está inserido em manutenção e está intimamente relacionado com a refatoração de código fonte.

Refatoração trata do problema de modificar um software sem alterar a sua semântica. Encontrar alterações de refatoração é um problema complexo e com grande número de soluções possíveis. Quando combinado com algum outro objetivo, como por exemplo, métricas estruturais de projeto de software (como acoplamento ou coesão), torna-se um excelente cenário para aplicação de heurísticas. Um bom exemplo de pesquisa com esse viés é o trabalho de Di Penta et al. (Penta, 2005), que propuseram um método para remoção de objetos não utilizados, clones e dependências circulares em grandes bibliotecas com o objetivo de reduzir seu tamanho (em número de instruções), assim como reduzir o acoplamento entre os objetos.

Outro exemplo é o trabalho de Ouni et al. (Ouni, Kessentini, Bechikh, & Sahraoui, 2015) que exploraram o problema de *code-smells* em projetos de software. *Code-smells* são o resultado de mudanças que um software sofre ao longo do tempo e que deterioram, pouco a pouco, o *design* original do sistema (Brown, Malveau, Mowbray, & Wiley, 1998). Segundo os autores, apesar de existirem várias técnicas automáticas para reparo de *code-smells*, nem todos os tipos permitem correções automáticas. Citam ainda que o número de *code-smells* pode ser tão elevado em sistemas de grande porte que pode não ser possível fazer o tratamento automático. Portanto, os autores propõem uma técnica chamada 'Reação Química' (*Chemical Reaction Optimization* ou CRO) para encontrar movimentos de refatoração que maximizem o conjunto de

code-smells resolvidos. Eles aplicaram a técnica em cinco instâncias de software *open source* com resultados positivos quando comparados com outras técnicas semiautomáticas (manual e apoiada por análise estática de código) e de SBSE (Algoritmo Genético, Arrefecimento Simulado e Análise de Partículas). A melhoria encontrada pelos pesquisadores foi observada na quantidade de *code-smells* corrigidos, assim como na qualidade dessas correções quando observada a prioridade para cada tipo diferente de *code-smell* eliminado do código.

Adnane et al. (Ghannem, El Boussaidi, & Kessentini, 2013)(Koza, 1994) adaptaram um algoritmo genético clássico, tornando-o iterativo, com o objetivo de encontrar soluções para o problema de refatoração utilizando a opinião dos engenheiros de software. Em um trabalho anterior, as ações de refatoração eram previamente cadastradas em um repositório que recomendava uma refatoração com base em similaridade estrutural. Na adaptação dos autores, o engenheiro passou a poder opinar e sugerir mudanças nesses padrões estruturais. Eles aplicaram o método e o algoritmo em duas instâncias do mundo real. A base de comparação foi o próprio algoritmo genético sem alterações e sem uso de opinião dos desenvolvedores de software. Os resultados mostram que a técnica aumentou a cobertura da refatoração clássica considerada nesse estudo, sendo mais eficiente.

Como é possível observar nas pesquisas citadas acima, há um conjunto de trabalhos com o objetivo de refatorar automaticamente o código fonte sob um ou mais critérios. Desse conjunto de trabalhos nasceu o ramo que trata do melhoramento automático de código fonte utilizando a heurística genética. A esse ramo, chamamos de melhoria automática de código fonte. Em especial, o melhoramento genético é a técnica mais comumente utilizada nos trabalhos deste ramo. Na próxima subseção, discutiremos o histórico desse ramo e apresentaremos trabalhos relacionados. Vale citar que muitos dos trabalhos já aplicam soluções muito próximas as que foram utilizadas nessa pesquisa. Porém, até onde sabemos, essa pesquisa é a primeira a observar as técnicas em JavaScript.

2.4 Programação genética

Em 1992, John Koza utilizou algoritmos genéticos para encontrar, de maneira automática, programas para cumprir determinadas tarefas. As primeiras tarefas pesquisadas envolviam a resolução de expressões matemáticas. O autor batizou o método de Programação Genética (Koza, 1994). Na Programação Genética, programas são representados como árvores de expressões contendo nós de dois tipos: função ou terminal. Funções podem ser operações aritméticas, booleanas, condicionais, funções de iteração, recursão, além de funções específicas do domínio do problema em questão. Terminais podem ser variáveis ou constantes.

O exemplo apresentado na Figura 5 representa a expressão matemática $3x^2+2x+1$. A partir dessa representação, a Programação Genética aplica os operadores genéticos para encontrar uma melhor árvore sob determinado aspecto e para determinado objetivo. Portanto, a Programação Genética é considerada uma especialização de algoritmos genéticos no domínio da manipulação de programas de computador.

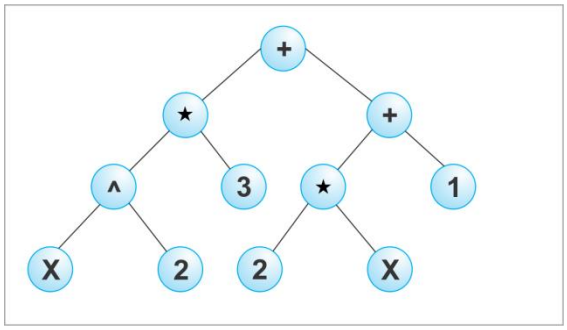

Figura 5 Exemplo de Árvore de Expressão

Em programação genética, a operação de cruzamento acontece da seguinte forma: são escolhidos dois indivíduos; em cada indivíduo é escolhido um ponto aleatório para o cruzamento; a sub-árvore existente a partir do ponto escolhido no primeiro individuo é inserida no ponto de cruzamento do segundo indivíduo e vice-versa. A Figura 6 ilustra essa situação. Operações de cruzamento como este costumam modificar o tamanho dos indivíduos devido a sua representação em árvore. Em algoritmos genéticos convencionais, que usam outro tipo de representação, esse aumento não é frequentemente observado.

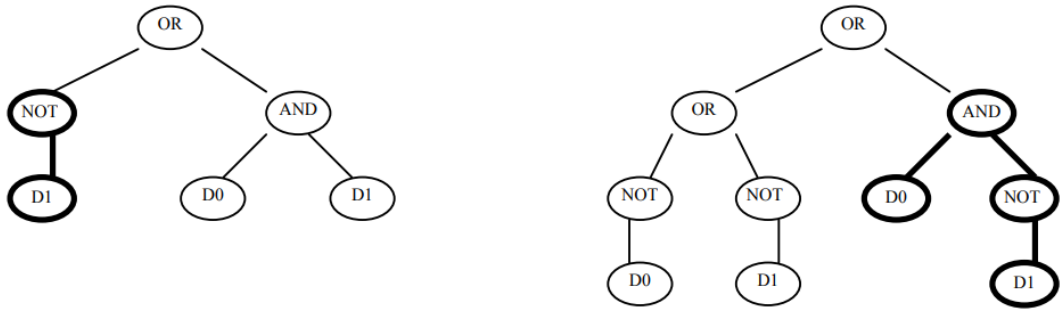

Figura 6 Exemplo de cruzamento em programação genética

Já o operador de mutação, no contexto da programação genética, consiste em selecionar aleatoriamente um indivíduo e um ponto de corte e substituir sua árvore a partir daquele ponto por uma outra árvore criada aleatoriamente.

Chamamos de melhoramento genético de sistemas (*Genetic Improvement*, ou GI) a melhoria de um software utilizando Programação Genética (Mark Harman, Langdon, et al., 2012). A partir de um sistema escrito por seres humanos, o melhoramento genético tenta evoluir

o software através de um processo de Programação Genética, levando em consideração um conjunto de critérios de qualidade previamente estabelecidos. Quando implementado para preservar as qualidades funcionais, o melhoramento genético pode ser visto como uma técnica de refatoração automática do código-fonte com o objetivo de melhorar as características não funcionais. Quando discutimos o uso de melhoramento genético para código-fonte, é possível produzir mudanças legíveis que podem ser verificadas tanto por engenheiros quanto por ferramentas de teste modernas.

Assim como a Programação Genética, o melhoramento genético usa uma árvore para processar as gerações em busca da solução ótima. Porém, no melhoramento genético é utilizada a Árvore de Sintaxe Abstrata (*Abstract Syntax Tree*, ou AST). A Figura 7 mostra um exemplo de AST para uma linha de código válida em código JavaScript.

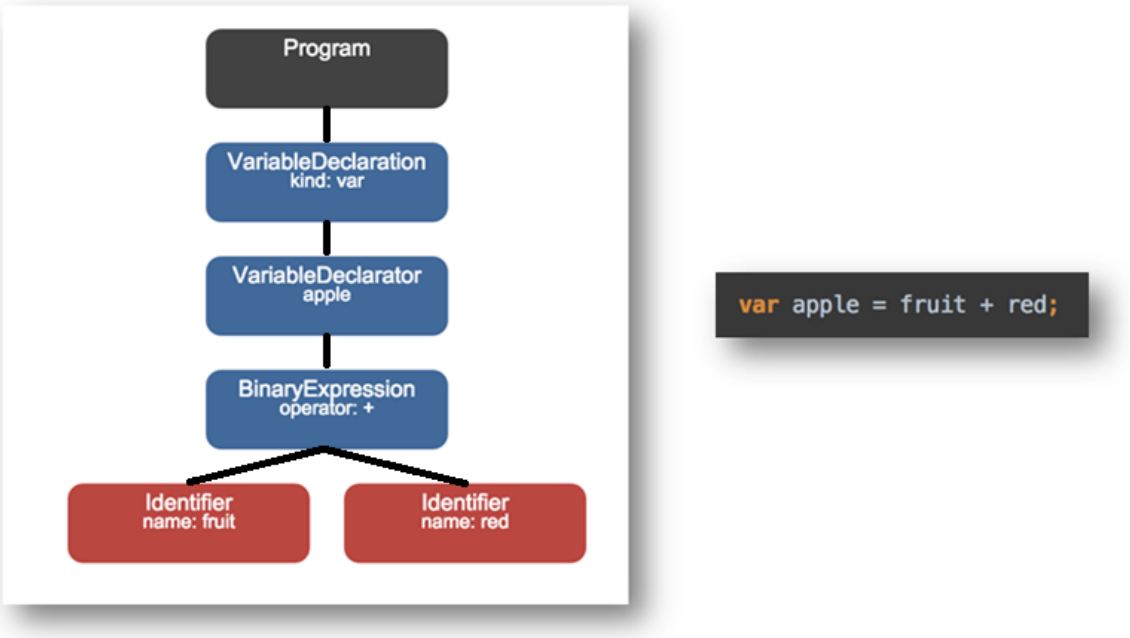

Figura 7 Exemplo de árvore de sintaxe abstrata

2.5 Programação genética na Engenharia de Software

Boa parte dos trabalhos de pesquisa na área de melhoramento genético são focados na melhoria do tempo de execução dos programas submetidos à otimização (Petke et al., 2018). Porém, a técnica pode ser aplicada na melhoria de outros requisitos não funcionais. Segundo Goues et al. (Le Goues, Nguyen, Forrest, & Weimer, 2012), as pesquisas conduzidas usando GI se dividem em quatro grandes áreas: correção de defeitos (Mark Harman, 2007; Le Goues, Dewey-Vogt, Forrest, & Weimer, 2012), melhoria do tempo de execução (Cody-Kenny & Barrett, 2013; William B Langdon, Modat, Petke, & Harman, 2014; Petke, Langdon, &

Harman, 2013), migração e transplante (Mark Harman, Jia, & Langdon, 2014) e adaptação dinâmica (Schulte, Weimer, & Forrest, 2015). No *survey* de Petke et al. (Petke et al., 2018), os pesquisadores relatam que melhoramento genético tem suas bases em vários tópicos, incluindo transformação de programas, síntese de programas, teste de software além de programação genética em si.

As aplicações de otimização em código fonte são uma consequência natural dos critérios escolhidos. Os critérios podem ser divididos em funcionais e não funcionais, onde existe uma grande dificuldade no que tange as propriedades não funcionais, que é a medição da propriedade desejada. Para propriedades não funcionais, como consumo de energia, medições precisas podem ser inviáveis. No entanto, essa precisão não é necessária para a otimização em si. O processo de otimização requer apenas uma função objetivo que orientará a busca de variantes desejáveis do software.

A métrica mais comum é o número de casos de testes do software passando com sucesso, ou seja, quanto mais testes uma nova solução consegue executar com sucesso, melhor é essa solução. O número de casos de teste tem sido a medida predominante na otimização de propriedades funcionais e não funcionais de código fonte. Em particular, Schulte et al. (Schulte et al., 2015), Arcuri et al. (Arcuri & Yao, 2008), (Arcuri, 2008), Wilkerson & Tauritz (Wilkerson, Tauritz, & Bridges, 2012), Forrest et al. (Forrest, Nguyen, Weimer, & Le Goues, 2009) e Le Goues et al. (Le Goues, Dewey-Vogt, et al., 2012) utilizaram como avaliação do resultado da otimização a execução com sucesso de toda a suíte de testes. Ryan et al. (Ryan & Walsh, 1995), (Walsh & Ryan, 1996), (Ryan & Ivan, 1999), (Ryan & Walsh, 1997) e Williams & Williams (Rosca, 1997), (Williams & Williams, 1996) também usaram a quantidade de casos de teste executados com sucesso em seu trabalho de paralelização.

A propriedade não funcional melhorada com mais frequência é o tempo de execução (Petke et al., 2018). No entanto, essa métrica varia entre sistemas e hardware. Ela varia até no mesmo hardware e sistema. O número de linhas ou instruções executadas é considerado como um *proxy* para o tempo de execução, independentemente do sistema (Petke et al., 2013), (M Harman & Langdon, 2014), (Cody-Kenny & Barrett, 2013). Langdon et al. (W. B. Langdon, 2015; William B. Langdon, 2015; William B Langdon, 2014; William B Langdon & Harman, 2015; William B Langdon et al., 2014) utilizaram os ciclos de processamento como proxy para tempo de execução. Em nossa pesquisa, adotamos o mesmo *proxy*, porém com adaptações para a realidade de JavaScript, conforme será detalhado no capitulo 3.

A categoria que mais tem trabalhos reportados é a melhoria do tempo de execução (Petke et al., 2018), onde temos trabalhos que levam em consideração mais de um aspecto não

funcional para a otimização. Petke et al. (Petke et al., 2013) aplicaram melhoramento genético em um sistema chamado MiniSAT (Kautz & Selman, 2007), que é um projeto *open source* escrito em C++ para resolver problemas booleanos (SAT). Ele usa as tecnologias mais modernas para a solução de problemas SAT, incluindo *Unit propagation, Conflict-driven clause learning* e *Watched literals.* Segundo os autores, o melhoramento genético do MiniSAT era um desafio significativo, pois o MiniSAT foi evoluído por programadores ao longo dos anos. Os pesquisadores queriam observar se era possível melhorar o tempo de execução da melhor solução humana. A abordagem usada na otimização é descrita no trabalho de Harman et al. (Mark Harman, Langdon, et al., 2012). O processo de otimização foi aplicado apenas à classe principal do sistema (Solver.cc) e a melhoria observada foi de 1% no melhor caso (medida em linhas de código), onde assertivas foram apagadas no código.

Harman et al. (Mark Harman, Langdon, et al., 2012) propuseram uma agenda de pesquisa relacionada a um framework baseado em melhoramento genético que está em desenvolvimento. O GISMOE (Mark Harman, Langdon, et al., 2012) é uma abordagem que usa o próprio sistema como oráculo para guiar o processo de otimização. Além disso, a abordagem permite que o engenheiro de software analise as várias versões propostas pelo otimizador para um determinado projeto, executando cada versão em paralelo com o código-fonte original do projeto. Tratando-se de um trabalho em andamento, o GISMOE possui diversas questões em aberto (visualização, medição/*profiling*, geração dos dados para testes, entre outras) o que representa direções de pesquisas futuras na área.

Langdon et al. (M Harman & Langdon, 2014) relatam os resultados da aplicação de melhoramento genético para gerar uma versão otimizada do nVidia CUDA *graphics card kernel*. O processo de otimização foi aplicado na biblioteca do *gzip* utilizada internamente pelo *kernel*. O objetivo era observar se é possível gerar opções otimizadas de componentes internos de um sistema legado, oferecendo ao engenheiro de software opções de refatoração ou substituição de componentes. Os resultados mostraram ao menos 55 versões que melhoraram cada qual um aspecto do tempo de execução, consumo de memória e o tamanho do arquivo compactado. Em teoria, a abordagem permite que um engenheiro analise as novas opções e proponha melhorias no código legado.

White et al. (White et al., 2011) propuseram um framework para otimização de programas em C usando melhoramento genético multiobjetivo para atender critérios não funcionais previamente configurados, onde o engenheiro de software pode estender o comportamento de avaliação para incluir critérios conforme a necessidade (memória, disco, energia, entre outros). Em particular, foi avaliado o tempo de execução dos programas. Foram

usadas oito funções distintas para a avaliação baseada em simulação. Os resultados são positivos: no melhor caso houve 5% de melhoria em relação ao tempo de execução programa original. Vale observar que foram identificados alguns padrões de melhoria, como tipos de instruções que executam mais rápido que outras e podem ser substituídas na implementação. Porém, pouco pode ser afirmado dado as limitações presentes no experimento. Os testes para avaliar uma mudança foram selecionados de acordo com algum critério estrutural (cobertura, por exemplo), avaliando as mudanças sob uma perspectiva diferente do original. Além disso, o experimento otimizou as funções separadamente, observando melhorias apenas nesse contexto isolado. Não houve atualização dessas funções no software original para avaliação e comparação com todas elas atualizadas.

Langdon et al. (Mark Harman et al., 2014) aplicaram melhoramento genético em migração e transplante de funcionalidade entre sistemas em operação. Os pesquisadores conduziram um experimento utilizando um sistema de mensagens instantâneas, chamado Pidgin, e outro de tradução de texto, chamado *Babel Fish*. O objetivo era adicionar o comportamento de tradução de textos ao *Pidgin*. A técnica é dividida em duas partes: crescimento e enxerto. Na parte de crescimento, um desenvolvedor de software seleciona as partes que compõem a funcionalidade, juntamente com seus testes, para que o processo de otimização procure um grupo de instruções (classes e métodos) otimizado sob a perspectiva do tempo de execução. Já na parte do enxerto, o desafio é encontrar pontos de inserção no software de destino. Para tal, os pesquisadores escolhem aleatoriamente um ponto e aplicam três tipos distintos de mutação (*variable replacement*, *statement replacement* e *statement swapping*). Foram realizadas 30 rodadas do processo de otimização, com uma população de 500 indivíduos evoluída ao longo de 20 gerações. Os autores concluem que além de encontrar soluções boas, esse é o primeiro trabalho de transplante de código usando SBSE reportado até então.

No trabalho de Bruce et al. (Bobby R Bruce, Petke, & Harman, 2015) os pesquisadores alteraram o experimento conduzido por Langdon et al (Mark Harman et al., 2014) para que o melhoramento genético fosse aplicado sob a perspectiva de consumo energético. Os pesquisadores utilizaram um modelo de medição de energia da Intel (*Intel Power Gadget*), que considera apenas o consumo de energia pelo processador. Memória, disco, tela e rede não são medidos e não estão contabilizados. Os autores fizeram modificações na função objetivo do otimizador para tratar essa nova medida, assim como nos operadores para que o processo pudesse acompanhar a nova função objetivo. Executaram três experimentos que evoluíram uma população de 100 indivíduos por 20 gerações. Ao final, a melhor solução foi submetida aos testes originais. A solução foi avaliada 20 vezes para capturar a incerteza imposta pelos componentes aleatórios envolvidos na medição de consumo energético. A solução que

representou o melhor caso encontrado pelo otimizador mostrou uma melhoria de 25% no consumo de energia quando comparada com a solução original.

Cody-Kenny et. al. (Cody-Kenny, Lopez, & Barrett, 2015) propuseram uma ferramenta chamada locoGP que usa melhoramento genético em programas escritos em Java com objetivo de reduzir seu número de instruções. A ferramenta lê o código-fonte em Java, monta sua árvore sintática e realiza as operações cruzamento e mutação. As operações decidem trocar, apagar ou replicar nós da árvore sintática de acordo com a operação, onde eles definiram um catálogo de opções possíveis. A avaliação é feita medindo a quantidade de instruções de cada solução. A ferramenta foi avaliada utilizando instâncias de programa que implementam 12 algoritmos de ordenação distintos (*Insertion Sort*, *Bubblesort*, *BubbleLoops*, entre outros). Em todos os casos, o locoGP foi capaz de encontrar alternativas melhores do que as implementações que lhe foram apresentadas, ou seja, reduziu o tamanho em instruções em cada um desses algoritmos.

Haraldsson et al. (Haraldsson, Woodward, Brownlee, Smith, & Gudnason, 2017) conduziram um estudo com melhoramento genético em sistemas com código fonte em Python sob a perspectiva de redução do tempo de execução. Seu objetivo era investigar os resultados dessa aplicação, além de gerar conhecimento sobre o espaço de busca. Aplicaram o experimento em um software chamado ProbAbel, com aproximadamente cinco mil linhas. A medição utilizada na função objetivo foi o tempo de execução, medido em segundos, coletado como a média de 20 execuções de cada modificação encontrada no programa. Os autores observaram uma redução média no tempo de execução de cerca de 0.5%.

Bruce et al. (Bobby Ralph Bruce, Petke, Harman, & Barr, 2018) realizaram um estudo sobre o espaço de busca em melhoramento genético para os operadores de: 1) exclusão (apaga uma linha); 2) cópia (insere a cópia de uma linha de código em um lugar aleatório do código) e 3) troca (onde uma linha aleatória é substituída por outra). O objetivo do melhoramento genético nessa pesquisa é a redução de consumo de energia. Os autores propõem uma função objetivo que usa um algoritmo de aproximação entre as variantes e o código original. A aproximação é calculada entre os resultados dos testes das variantes otimizadas de um programa e seu código original, onde, por exemplo, uma variante pode passar em 99% dos testes e ser considerada correta.

Os autores observaram os efeitos de aceitar soluções aproximadas em quatro softwares: 7zip, Bodytrack, Ferret e OMXPlayer. O objetivo da pesquisa era descobrir com qual frequência um processo aleatório realiza modificações efetivas no espaço de busca de cada um dos programas, que impacto estas modificações produzem nos resultados e como isso varia entre testes exatos e aproximados. Além disso, investigaram os efeitos que modificações eficientes

produzem quando combinadas. Eles relatam que utilizando aproximação a redução média encontrada é superior a 33,9% contra 0,76%. Relatam também que 61,5% das modificações combinadas permitem a redução do consumo de recursos computacionais, ainda gerando o resultado esperado do sistema ou uma aproximação aceitável.

2.6 Outras técnicas de otimização de tamanho em JavaScript

Fora do contexto da Engenharia de Software Baseada em Busca, também existem pesquisas com foco em redução de tamanho de código JavaScript. Um exemplo de técnica é o trabalho de Burtscher et al. (Burtscher & Livshits, 2010) na qual os pesquisadores propuseram um técnica de compressão de código JavaScript. A técnica divide a AST em 3 partes (regras, identificadores e literais) e executa uma compressão independente em cada uma dessas partes. Os pesquisadores relatam experimentos conduzidos em 9 bibliotecas JavaScript, nos quais foi possível observar uma redução entre 4% e 30% do tamanho do código fonte. Como ponto negativo, os pesquisadores citam que a técnica aumenta a necessidade de processamento no lado cliente das aplicações, mas justificam que não é expressiva visto o ganho com a redução. Não há números para discutir essa afirmação uma vez que tempo de carga e processamento da descompactação do código não foi observado nos experimentos.

A pesquisa de Fard e Mesbah (Fard & Mesbah, 2013) sobre *code smells* em JavaScript também pode ser considerada um trabalho de otimização em que uma das consequências foi a redução do tamanho do código. Entre os treze *code smells* encontrados pelos pesquisadores, há um classificado como *dead code*. *Dead code* (código morto) é um código que, mesmo após a execução dos testes ou da aplicação, não foi executado ou interpretado. Os pesquisadores propuseram uma ferramenta, chamada JSNose[5], que exercita o código JavaScript e busca os 13 tipos de *code smells* previamente identificados. Eles executaram experimentos em 11 aplicações *web* com tamanho variando entre 71 e 26.000 LOC. Os pesquisadores testaram, manualmente, por 10 minutos cada aplicação com a supervisão dinâmica da JSNose. Eles relatam ter conseguido detectar os 13 tipos de *code smells* nas aplicações observadas.

O próprio processo de minificação de código JavaScript, citado na introdução deste documento, é uma técnica de análise estática que consiste em uma série de alterações para reduzir o tamanho do código-fonte JavaScript (Jensen et al., 2009; Richards et al., 2010). Normalmente, ela é realizada em tempo de *build* da aplicação e já fica disponível com a versão final do código, onde um servidor web responde o código minificado no lugar do código original para as requisições do mesmo.

[5] http://salt.ece.ubc.ca/content/jsnose/

2.7 Considerações finais

Conforme visto nesse capítulo, o uso de algoritmos heurísticos de busca para encontrar soluções para problemas de Engenharia de Software modelados como problemas de otimização é um campo em expansão. Em praticamente todas as áreas de conhecimento da Engenharia de Software há trabalhos que aplicam técnicas baseadas em busca. Existe uma grande diversidade de heurísticas que podem ser aplicadas em diversos cenários, ainda que os algoritmos genéticos sejam a heurística mais frequentemente aplicada a problemas da Engenharia de Software. Em especial, neste capítulo foi apresentado o campo da Programação Genética, a especialização dos algoritmos genéticos no domínio de melhoria de software, e trabalhos relacionados ao tema.

No Capítulo III será apresentada uma investigação do espaço de busca do problema de otimização de código JavaScript para redução de tamanho, visando identificar se a heurística genética é adequada para este problema. Para fins de comparação, o algoritmo genético será comparado com uma busca local.

3 Caracterizando o espaço de busca na melhoria de código-fonte em JavaScript

Neste capítulo, apresentaremos a pesquisa exploratória realizada para caracterizar o espaço de busca para redução de tamanho de código-fonte escrito em JavaScript. Serão discutidos aspectos relevantes da linguagem no que tange a otimização proposta, quais as decisões tomadas e a conclusão sobre a heurística base de otimização proposta na Tese.

3.1 Introdução

Conforme discutido no capítulo 2, existe um grande número de heurísticas compatíveis com o problema de otimização de código fonte, sendo as duas principais abordagens baseadas em algoritmos genéticos (busca baseada em população) e algoritmos de busca local (busca baseada em vizinhança). O melhoramento genético, aplicação de programação genética no domínio de desenvolvimento de software, é a abordagem mais utilizada atualmente.

Neste trabalho de pesquisa, utilizamos a busca local como o algoritmo central de otimização para direcionar uma abordagem de otimização de código-fonte visando reduzir o tamanho de programas JavaScript sem alterar a sua funcionalidade. O algoritmo de busca local foi selecionado após o exame do espaço de soluções do problema. Para tal, foi necessário construir um ferramental de apoio que nos permitisse observar os resultados da aplicação de heurísticas em código JavaScript, incluindo o melhoramento genético e a busca local. Isso permitiu uma melhor compreensão do espaço de busca do problema, das opções disponíveis para tratá-lo, além de fornecer uma base de comparação entre as heurísticas.

Esse capítulo está dividido em quatro seções, começando por esta introdução. A Seção 3.2 descreve o ferramental de apoio construído para suportar a pesquisa e o ambiente de computação de alto desempenho utilizado. A Seção 3.3 discute o espaço de busca do problema e descreve o experimento exploratório executado para caracterizar este espaço. Por fim, a seção 3.4 apresenta as conclusões do estudo exploratório.

3.2 Ferramental de apoio

Para que o trabalho de pesquisa pudesse observar os resultados da aplicação de melhoramento genético em JavaScript, algumas tarefas foram executadas, fracassaram e

trouxeram o conhecimento necessário para avançar. Durante dois anos de pesquisa construímos cinco versões de um "Otimizador", cada qual com seus pontos fracos e fortes. Nessa seção vamos apresentá-las e abordar, com mais detalhes, a última versão disponível.

3.2.1 A Ferramenta ECJ

Alguns dos trabalhos anteriores em melhoramento genético citam o uso da biblioteca ECJ [6] para aplicação de programação genética, evolução gramatical e melhoria genética. Baseando-se nessas recomendações, começamos a criação do ferramental de melhoramento genético de programas JavaScript usando o ECJ, especificamente com seus módulos de evolução gramatical (Wilson, McIntyre, & Heywood, 2004), que utiliza uma gramática para descrever o software ao invés de uma árvore sintática de código. Exceto por isso, não há diferença entre o procedimento de melhoramento genético descrito no Capítulo 2 e a evolução gramatical.

Essa primeira versão teve resultados negativos quando houve necessidade de representar bibliotecas grandes (por exemplo, a biblioteca Moment.js, com 9.978 linhas de código). Gerar o arquivo de gramática que descrevia o código de cada biblioteca e cada uma de suas variações, foi o primeiro desafio. Utilizando a ferramenta não foi possível representar algumas situações simples, como métodos com sobrecarga e recursividade, além de uma situação mais complexa: o objeto global[7] do JavaScript. É comum encontrar a inicialização de objetos e bibliotecas em JavaScript dentro desse objeto global. A gramática padrão do ECJ para a linguagem JavaScript não permitiu representar corretamente essa situação. Portanto, não seria possível prosseguir usando o ECJ com essas limitações.

O módulo que faz a geração da gramática foi inicialmente escrito como um executável externo ao ECJ. Isso permitiria continuar utilizando o mesmo sem o ECJ. Ele foi produzido utilizando um componente para geração de gramáticas já disponível e utilizado em grandes. IDEs, o ANTLR[8]. O uso do ANTLR permitiu criar um analisador sintático genérico e exportar as informações em arquivo no formato que o ECJ usa. Um exemplo simples da gramática pode ser visto na Figura 8.

---

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

[7] https://msdn.microsoft.com/en-us/library/52f50e9t(v=vs.94).aspx

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

```
<prog> ::= <op>
<op> ::= (if-food-ahead <op> <op>)
<op> ::=  (progn2 <op> <op>)
<op> ::= (progn3 <op> <op> <op>)
<op> ::= (left) | (right) | (move)
```

Figura 8 Gramática JavaScript simplificada[9]

Uma vez que o processo de gerar a gramática a partir de um programa JavaScript estava pronto e desenvolvido externamente ao ECJ, o escopo do projeto de pesquisa foi alterado para implementar um Otimizador de JavaScript completo, passando a contemplar os passos que o ECJ suportava (o algoritmo genético em si e as operações sobre a gramática), além de execução dos testes unitários das bibliotecas e a avaliação das variantes de código fonte geradas pelo Otimizador. A busca por uma suíte de otimização já pronta foi abandonada nesse ponto e optamos por implementar uma única ferramenta para executar todo o processo de otimização. Para tal, precisaríamos de um interpretador de JavaScript.

3.2.2 O projeto Jurassic

Dado que o gerador de gramática foi escrito em C#, optamos por procurar um interpretador de programas JavaScript também escrito em C#. Após alguns testes preliminares, optamos pelo uso do projeto *Jurassic*[10], que nos permitiu interpretar o código JavaScript diretamente, ou seja, não havia mais a necessidade de gerar uma gramática para aplicar as operações. Poderíamos usar uma representação em árvore sintática de código, realizar mudanças na árvore e então gerar novamente o código. Isso nos permitiu superar as limitações anteriores e avançar na otimização do código. Porém, mesmo nessa nova versão ocorreram problemas.

O primeiro problema foi a lentidão para interpretar os arquivos de código-fonte de bibliotecas grandes (com mil linhas de código ou mais). Em alguns casos, o *Jurassic* levava mais de um minuto para carregar e interpretar uma biblioteca. Outros problemas foram as limitações na construção da árvore sintática: em algumas bibliotecas, a construção dinâmica de métodos e propriedades não era interpretada corretamente pela *Jurassic*. Por fim, a *Jurassic* não suporta completamente a versão mais atual do *JavaScript* (ECMA 6.0). Sendo assim, não poderíamos seguir adiante e houve necessidade de abandonar a *Jurassic*.

[9] https://github.com/ffarzat/ECJ/blob/master/ECJ/src/ec/app/ant/ant.grammar
[10] https://github.com/paulbartrum/jurassic

3.2.3 C# e Chrome V8

Nesse momento, foi realizada uma pesquisa para encontrar novas máquinas de interpretação de JavaScript que pudessem ser embutidas na atual implementação do Otimizador. A opção escolhida foi a Chrome V8 que, embora não seja escrita em C#, pôde ser adicionada ao Otimizador usando interoperabilidade entre C# e C++. Para tal, foi necessário montar o ambiente de desenvolvimento da Chrome V8 por inteiro e compilar uma versão específica com esse propósito. Uma vez que o funcionamento foi confirmado, foi possível corrigir os problemas da versão anterior e avançar na otimização do código fonte. Com o uso da Chrome V8, foi possível reduzir o tempo de interpretação das bibliotecas: todas tiveram seu código corretamente interpretado e com garantia de compatibilidade com a última versão do JavaScript. A partir de então as operações de cruzamento e mutação foram construídas seguindo a descrição feita no Capítulo 2 e foi adicionado o suporte para a execução dos testes unitários. Com isso, uma terceira versão do Otimizador foi completada e com ela foi possível observar alguns resultados preliminares da otimização de código-fonte JavaScript.

Contudo, alguns problemas mais sofisticados surgiram nessa versão. Com a criação das primeiras variações do código original, por vezes a otimização criou códigos que faziam chamadas recursivas ou loops infinitos. Foi preciso incluir tratamentos específicos para essas situações. Por si só, esse comportamento causava uma falha fatal na Chrome V8: falta de memória devido ao estouro de pilha, o que interrompia o processo de otimização por completo. Outro problema é que, para cada nova biblioteca adicionada ao processo de otimização, aumentava o esforço de configuração e desenvolvimento do suporte aos recursos específicos da mesma, além de preparar os seus testes unitários e o *framework* de suporte à execução desses testes. Uma vez que a Chrome V8 não é um navegador e o suporte aos testes unitários não era fornecido por padrão, era preciso codifica-lo.

3.2.4 NodeJs e a linguagem Typescript

Com todos os problemas até então mapeados e com a necessidade de incluir novas observações de otimização de código fonte JavaScript, uma nova versão do Otimizador foi produzida. O Otimizador, em sua última versão, foi todo construído em JavaScript sobre o ambiente NodeJs. Como a Chrome V8 é usada internamente no NodeJs, nenhum dos requisitos de desempenho foi perdido e todos os recursos específicos do JavaScript foram mantidos (objeto global, funções dos navegadores, suporte aos testes unitários, entre outros).

Os principais componentes do NodeJs permitem a sua manipulação e orquestração via código JavaScript. Essa característica foi o principal motivo da migração de tecnologia no

Otimizador. Com essa nova infraestrutura, seria possível adicionar bibliotecas apenas por configuração, gerar variantes de um código fonte usando bibliotecas JavaScript já existentes, além de executar os testes de maneira nativa, usando recursos do navegador ou qualquer recurso demandado pela biblioteca sem necessidade de modificação.

Alguns *frameworks* foram adicionados ao projeto para cumprir integrações específicas. Alguns deles se encontram disponíveis no NPM com o ambiente de desenvolvimento disponível para NodeJs. Um exemplo de framework utilizado é o KarmaJs[11] que tem como objetivo ser um executor de testes unitários escritos em JavaScript. O KarmaJs possui uma estrutura de plug-ins de navegadores, ou seja, permite que os testes sejam executados diretamente em um ou mais navegadores apenas por configuração. O KarmaJs inicia o navegador em questão, executa os testes unitários no mesmo e colhe os resultados. Vários navegadores podem ser configurados por vez, fazendo com os testes sejam executados várias vezes (uma por navegador). A lista de navegadores disponíveis é ampla e está disponível no site do framework[12].

Um dos navegadores mais utilizados com esse propósito (execução de casos de teste unitários) chama-se PhantomJs[13]. Este navegador executa como um processo em *background* e sem tela visível ao usuário. Ao contrário dos navegadores comuns, ele não renderiza HTML em tela (apenas em *buffer*). Com isso, testes de bibliotecas que precisam de recursos do navegador podem ser executados em um ambiente de terminal (ou console, no caso do Windows). Ainda assim, há suporte para exportar o resultado da renderização como uma imagem ou mesmo *pdf*, permitindo criar evidências de um resultado de teste em tela. O PhantomJs pode ser adicionado à suíte de testes unitários em tempo de configuração usando o KarmaJs, não exigindo alteração no código da biblioteca ou dos seus testes unitários.

Com a implementação baseada em NodeJs, NPM, KarmaJs e PhantomJs tivemos disponível um ferramental que permitia aplicar as heurísticas nos cenários de execução expostos anteriormente (navegador, servidor e *mobile*). Porém, antes de escrever o código do novo Otimizador optamos por adicionar uma camada de programação com o objetivo de minimizar os impactos de utilização do próprio JavaScript, dadas as suas características conforme visto no Capítulo 1. Com base em uma pesquisa sobre IDEs e ambientes para desenvolvimento de larga escala para JavaScript chegamos ao *Typescript*[14]. Trata-se de uma extensão do JavaScript que se destina ao desenvolvimento simplificado de aplicações de larga escala, proposta pela Microsoft,

[11] https://karma-runner.github.io/0.13/index.html
[12] https://karma-runner.github.io/0.13/config/browsers.html
[13] https://github.com/karma-runner/karma-phantomjs-launcher
[14] https://www.typescriptlang.org/

porém de domínio público (BIERMAN; ABADI; TORGERSEN, 2014). O *Typescript* fornece um sistema de módulos, classes, interfaces, além de tipos primários. Um dos seus objetivos é proporcionar uma transição mais suave de programadores oriundos de linguagens orientadas a objeto ou fortemente tipificadas para JavaScript. Outra característica é o fato de que qualquer código JavaScript é um código *Typescript* válido. Isso permite incluir bibliotecas já existentes escritas em JavaScript, bastando referencia-las e fazer uso de seus métodos no ambiente do *Typescript*.

Com esse ambiente definido, algumas bibliotecas foram adicionadas ao Otimizador para cumprir papéis específicos dentro do processo de otimização: produzir a árvore sintática de um código, executar mudanças na árvore, gerar o novo código compatível com o original, entre outras. As bibliotecas e seus objetivos são os seguintes:

- *Esprima*[15]: Analisador sintático de JavaScript, utilizado para gerar a árvore sintática;
- *Escodegen*[16]: Gerador de código JavaScript que usa como base a árvore do Esprima;
- *Traverse* [17] : Implementação de um *Visitor* para objetos JavaScript, utilizado para implementar os operadores genéticos da otimização (operações na árvore sintática);
- *ShellJs*[18]: Permite executar comandos diretamente no terminal ou console;
- *NPMCli*[19]: Permite executar comandos do NPM.

Mesmo partindo de bibliotecas maduras (algumas delas com mais de cinco anos de desenvolvimento e dez versões evolutivas disponíveis), encontramos dificuldades quando as incluímos no processo de otimização e levamos seu uso ao extremo. Um bom exemplo foi o caso do *Escodegen*, que possuía uma falha não tratada em um caso específico de geração de código que lidava com objetos de expressão regular (e não o seu uso costumeiro como *String*). Muito tempo de desenvolvimento foi dedicado para encontrar e corrigir falhas como essa.

3.2.5 Implementação cliente-servidor

Outro desafio da troca de tecnologia veio do fato de que o NodeJs ser *single thread*, ou seja, só executa um processo por vez. Não é possível, por exemplo, gerar e controlar duas atividades na mesma pilha de execução. Essa prerrogativa trouxe uma complicação grave ao

[15] http://esprima.org/
[16] https://github.com/estools/escodegen
[17] https://github.com/substack/js-traverse
[18] https://github.com/shelljs/shelljs
[19] https://www.npmjs.com/package/cli

Otimizador: o tratamento de cenários de falta de memória e erros fatais. Uma vez que as operações do Otimizador, assim como os testes das bibliotecas, podem causar uma falha dos tipos citados acima, temos um quadro onde todo o Otimizador caía em falha catastrófica e encerrava sua execução inesperadamente e sem controle.

Para resolver essa situação, o Otimizador teve seu projeto reformulado para trabalhar em duas camadas: uma camada servidora e uma camada de clientes. O objetivo foi executar nos clientes qualquer operação com risco de falha e manter o processo principal protegido no servidor. Para permitir a troca de mensagens entre as camadas cliente e servidor foi utilizado o protocolo *WebSockets*, que permite estabelecer uma comunicação direta e permanente entre máquinas distintas usando o protocolo HTTP. Esse protocolo vem sendo utilizado em sistemas web que tem necessidade de atualização de dados em tempo real, como sistemas de leilão online ou sistemas de visualização de dados críticos.

O tratamento de falhas no Otimizador passou a funcionar através do controle do servidor sobre a sua conexão com determinado cliente. Uma vez que o servidor decide quais operações precisam ser realizadas, o mesmo as distribui estas operações entre os clientes disponíveis que, por sua vez, executarão de fato as operações. Caso um dos clientes tenha uma falha catastrófica, a sua conexão com o servidor será encerrada de maneira inesperada. Nesse momento, o servidor percebe a queda na conexão e redireciona para outro cliente aquela operação que causou a falha. Durante o processamento de uma operação por parte dos clientes, o servidor aguarda até que um grupo de operações, que apenas possuem sentido lógico em conjunto (por exemplo, todos os cruzamentos e mutações de um algoritmo genético), estejam concluídas para então prosseguir com o processo principal de otimização.

3.2.6 Lobo Carneiro

No mês de julho de 2016 surgiu a oportunidade de usar o novo supercomputador da COPPE/UFRJ. Batizado de Lobo Carneiro, em homenagem ao falecido professor Fernando Luiz Lobo Barboza Carneiro (1913-2001), o equipamento é um supercomputador baseado em cluster composto de 252 unidades de processamento, cada uma com 24 nós de processamento que podem executar 48 threads em paralelo usando HyperThreading. Os nós de processamento podem endereçar 16 Tb de RAM e 720 Tb de espaço em disco. Em plena capacidade, Lobo Carneiro pode realizar mais de 226 trilhões de operações matemáticas por segundo. Com esse poder de processamento disponível, algumas das limitações da plataforma anterior (como a quantidade de memória disponível) deixaram de existir.

Para que fosse possível executar o Otimizador em um ambiente paralelo como o Lobo Carneiro, foi necessário realizar adaptações no código da versão anterior, produzindo assim uma versão específica para esse ambiente. O sistema operacional do Lobo Carneiro é o SUSE *Linux Enterprise Server*, versão 12. Como o NodeJs não possui um pacote específico para esse ambiente, foi necessário compilá-lo sob a conta do usuário cedido para uso. Uma vez resolvido o problema da versão do NodeJs, o Otimizador foi capaz de executar seus próprios testes. Porém, apesar do processo do NodeJs enxergar os 16 Tb de memória disponível, ele não era capaz de perceber todos os processadores disponíveis.

A arquitetura do Lobo Carneiro é de *Cluster*, ou seja, ele é composto de 252 nós, cada um com 1 processador composto por 24 *cores* com tecnologia *HyperThreading*, podendo executar até 48 tarefas em paralelo. A alocação de nós para um processo é realizada através de um sistema de fila chamado *Altair PBS Professional*[20]. O processo de alocação de recursos (disco, memória, nós e *cores*) é simples: através de um arquivo de script *bash* nativo do Linux é possível descrever que recursos são necessários para executar um programa e fazer uso dos mesmos.

Porém, mesmo após a alocação de mais nós, os processos do NodeJs continuaram a enxergar apenas 48 *cores* (24 *cores* físicos), pois o PBS lança o processo raiz do Otimizador em um desses nós, que funciona como o *host* da execução, e informa quais outros nós comporão o *Cluster* durante a sessão através de um arquivo texto passado por linha de comando. Por si só, o Otimizador não sabe da existência dos demais nós (e, consequentemente, dos seus *cores*). A recepção e tratamento do arquivo enviado pelo PBS não estavam previstos na implementação do Otimizador. Portanto, foi necessário explorar outras opções para fazer uso do potencial real do supercomputador.

Dada a inviabilidade de modificar o NodeJs para que este lance seus processos tanto no *host* quanto nos demais nós indicados no arquivo passado pelo PBS, a primeira opção foi fazer a invocação direta dos *cores* em cada nó utilizando MPI (*Message Passing Interface*)[21], um padrão aberto para comunicação entre nós de um *cluster* via rede. O MPI permitiu executar o processo de otimização no *host* alocado pelo PBS e solicitar que apenas as operações que alteram a árvore sintática do programa que está sendo otimizado e seus casos de testes fossem executados em paralelo nos outros nós do cluster.

[20] www.pbsworks.com

[21] https://www.open-mpi.org/

Com isso, o processo passou a consumir 6 nós do cluster (48 x 6 *cores*) distribuídos da seguinte forma: um como *host* do processo de otimização e os outros cinco como escravos para execução paralela dos testes. Os casos de teste de novos indivíduos gerados pelo processo de otimização precisavam ser avaliados cinco vezes para que obtivéssemos uma medida mais confiável do seu tempo de execução (calculado como o valor médio dessas cinco execuções dos testes) e, por isso, cinco *cores* foram invocados em paralelo quando os testes de um indivíduo eram avaliados. Nessa configuração foi possível avaliar 48 indivíduos simultaneamente. Posteriormente, modificamos o processo de avaliação para não utilizar o tempo de execução dos testes como o valor da função de *fitness*, devido ao fato de que o tempo é influenciado por várias variáveis externas e das quais o ambiente não possui controle algum. Essa discussão será abordada no Capítulo 4.

Com isso, o tempo de execução do Otimizador melhorou, chegando a processar uma biblioteca pequena em 40 horas contra 120 horas em um computador *desktop*. Porém, ainda havia a limitação de 48 *cores* do *host*, fazendo com que o limite do Otimizador fosse avaliar 48 indivíduos por vez, independentemente do número de nós e *cores* disponíveis no supercomputador. No *host* é mantido o processo principal, além dos indivíduos em análise. Quando uma nova avaliação se faz necessária, o *host* consome, simultaneamente, cinco cores disponíveis nos nós que compõem a sessão. Assim, consumimos 288 cores simultaneamente, usando 100 Gb de *RAM* para cada par de algoritmo e biblioteca.

Levando em conta a limitação não vencida do número de indivíduos que poderiam ser avaliados simultaneamente, um conjunto de alterações foi realizado. Primeiro, cada nó passou a executar uma configuração do algoritmo selecionado, utilizando um *core* para o algoritmo propriamente dito e os demais 47 *cores* para as avalições dos indivíduos. Em seguida, foram disparadas 60 rodadas em paralelo, uma em cada nó. Usamos 60 rodadas paralelas porque os experimentos foram planejados para trabalhar com 60 observações do processo de otimização por instância. Assim, pudemos executar em paralelo todas as rodadas de uma biblioteca por algoritmo. Nessa composição, consumimos 2880 cores e 500 Gb de RAM.

Mesmo com essas melhorias e o uso do ambiente paralelizado, as bibliotecas maiores ainda consomem um tempo muito grande devido às restrições dos seus testes. Um bom exemplo é a Jquery, cuja suíte de testes é composta por 1744 casos de testes que levam 40 segundos para executar nesse ambiente. Vale ressaltar que a Jquery precisou de uma configuração mais agressiva de memória (1TB RAM) devido ao tamanho da biblioteca, que possui 7412 linhas de código e sua árvore sintática ocupa 25 MB em memória para cada indivíduo.

3.3 Espaço de busca em JavaScript

As decisões de projeto que levaram à técnica de melhoria do código fonte descrita neste trabalho de pesquisa foram tomadas de acordo com as observações coletadas após a execução de um experimento exploratório usando um algoritmo genético e um algoritmo de busca local com objetivo de reduzir o tamanho de bibliotecas escritas em JavaScript. O experimento foi projetado para trazer uma ideia aproximada do espaço de soluções no problema de redução do tamanho do código-fonte JavaScript e foi executado na última versão do Otimizador apresentado na Seção 3.2.

O algoritmo *Stochastic Hill Climbing* (SHC) (Brownlee, 2011), (Forrest & Mitchell, 1993) é um *First-ascent Hill Climbing* (Capítulo 2) que escolhe aleatoriamente a próxima solução a ser examinada a partir do conjunto de todos os potenciais vizinhos da solução mais conhecida. Ele foi selecionado para ser executado no experimento exploratório porque é um algoritmo de busca local imparcial. Esse algoritmo pega a representação da AST de um programa e seleciona, aleatoriamente, um nó dessa árvore para ser removido. Se a variante de programa resultante passar em todos os casos de teste com sucesso, ela será selecionada como a melhor solução até o momento. Se a variante de programa não passar em todos os casos de teste, o nó removido será restaurado na árvore sintática e outro nó será selecionado aleatoriamente para remoção. Este procedimento foi repetido 5.000 vezes para cada uma das bibliotecas utilizadas no experimento exploratório.

O algoritmo genético foi configurado com uma população de 100 indivíduos, cada um representando uma variante do código original. A população inicial foi criada através da aplicação de uma única mutação no código original e evoluiu ao longo de 50 gerações, levando a 5.000 avaliações do conjunto de testes. O operador de cruzamento escolhido foi o *single point crossover* (dois indivíduos selecionados por torneio com 100% de probabilidade). O operador de mutação escolhido seleciona e remove aleatoriamente um nó da AST de um indivíduo (com 75% de probabilidade).

3.3.1 Programas selecionados para o experimento

Ambos os algoritmos foram executados em 11 programas JavaScript. São bibliotecas ou ferramentas muito usadas e possuem casos de testes que garantem pelo menos 90% de cobertura dos comandos componentes do código fonte. Do conjunto de bibliotecas que atendem a essas restrições, as escolhas para o experimento foram feitas por conveniência, algumas delas devido à experiência de uso dos pesquisadores. Tal experiência permitiu aos pesquisadores

desempenhar o papel de usuários e avaliar a efetividade das mudanças propostas por diferentes algoritmos. Estes programas estão listados abaixo e a Tabela 1 mostra suas características.

- **Browserify**: adiciona ao ambiente do NodeJs métodos para execução do código em navegadores, permitindo escrever JavaScript para várias plataformas;
- **Exectimer**: mede o tempo de execução de código JavaScript no nível de nanosegundos;
- **jQuery:** biblioteca com métodos utilitários para manipulação dos elementos no navegador;
- **Lodash:** biblioteca com métodos utilitários complementares aos da biblioteca Underscore;
- **Minimist:** faz o tratamento de argumentos em linha de comando para JavaScript;
- **Plivo-node:** fachada para criação de aplicações em JavaScript utilizando a API do Plivo;
- **Jade** (Pug): um mecanismo de templates para navegadores e NodeJs;
- **Tleaf:** mecanismo para criação de testes para *controllers* na tecnologia AngulasJs;
- **Underscore** adiciona métodos utilitários ao JavaScript sem estender os objetos internos;
- **UUID:** gera e analisa identificadores únicos globais (GUID);
- **XML2JS:** converte *Strings* no formato XML para objetos JavaScript e vice-versa.

Sessenta rodadas de otimização independentes foram realizadas para cada biblioteca e algoritmo. No total, 2.880 nós de processamento e 500 Gb de RAM do supercomputador foram reservados para essa tarefa.

3.3.2 Comparação entre busca local e algoritmo genético

A Tabela 2 apresenta as médias, desvios padrão e medianas dos resultados produzidos pelo algoritmo genético e pela busca local no experimento exploratório. As variantes do código das bibliotecas observadas, sem exceção, passam em todos os casos de teste. Os resultados são mostrados como redução percentual no tamanho da biblioteca, medida em caracteres.

O tamanho dos programas pode ser medido em várias unidades, incluindo linhas de código, número de instruções e número de caracteres. Os programas JavaScript geralmente são empacotados e minimizados antes de sua instalação em produção. O processo de minificar remove espaços em branco e quebras de linha e, portanto, o número de linhas de código deixa de ser uma medida de tamanho representativa para programas minificados. Por outro lado, o

número de instruções em um programa é apenas parcialmente relacionado ao seu tamanho em *bytes* armazenados no servidor e transferidos através da Web, pois instruções diferentes podem ser expressas em um número distinto de caracteres. O número de caracteres é uma medida de menor granularidade e também está fortemente relacionada ao número de *bytes* ocupados pelo código-fonte. Portanto, medimos o tamanho dos programas JavaScript como o número de caracteres na versão minificada de seu código-fonte.

Tabela 1 Características dos programas utilizados no experimento. A primeira coluna mostra o número de linhas de código em cada programa; em seguida, o número de casos de teste, o percentual de comandos cobertos pela suíte de testes, o número de downloads (dividido por 1000) em junho de 2018 e a versão observada.

Biblioteca	LOC	#Testes	Cobertura	Uso	Versão
Browserify	757	570	99%	1,9	14.3.0
Exectimer	195	37	91%	0,4	n/a
jQuery	7.607	937	91%	48	3.2.0
Lodash	10.795	2.077	94%	33,8	3.10.1
Minimist	193	140	99%	25,8	1.2.0
Plivo-node	609	26	91%	10,6	n/a
Pug	400	240	98%	356	4.0.0
Tleaf	133	131	96%	0,2	n/a
Underscore	1.481	198	95%	8,71	1.8.3
UUID	209	21	91%	11.497	n/a
XML2JS	526	83	93%	3.783	0.4.16

Tabela 2 Comparação entre os resultados produzidos pelo algoritmo genético (GA) e a busca local estocástica (SHC). Os resultados são apresentados como percentual do número de caracteres na versão minificada da melhor solução encontrada por cada algoritmo em cada rodada.

	GA		SHC	
Biblioteca	**Média e desvio padrão**	**Mediana**	**Média e desvio padrão**	**Mediana**
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%

Podemos notar que os resultados médios para a busca local em todas as bibliotecas são melhores que os resultados médios produzidos pelo algoritmo genético. Oito de onze medianas são maiores do que zero para a busca local, enquanto todas as medianas do algoritmo genético são iguais a zero. Esses resultados mostram que, embora o algoritmo genético consiga encontrar variantes reduzidas do código fonte, mais de 50% das suas execuções falham nessa tentativa. Essas falhas parecem ser independentes do tamanho do programa, já que os três casos para os quais a busca local falhou na maioria das 60 tentativas se referem a programas pequenos (*Browserify*, *minimist* e *xml2js*). Os desvios padrão maiores também mostram que uma execução independente da busca local não garante que o algoritmo encontrará uma boa solução.

Para entender o fraco desempenho do algoritmo genético, na próxima seção examinaremos a distribuição das variantes produzidas pela busca local.

3.3.3 Distribuição espacial dos *patches*

As variantes de código fonte produzidas por cada rodada da busca local foram comparadas ao programa original. Essa comparação produziu uma sequência de *patches* propostas pelo algoritmo para reduzir o tamanho de cada programa original. Esses *patches* foram representados como diferenças textuais (como os resultados produzidos pelo algoritmo de comparação de arquivos de um sistema de controle de versões (Myers, 1986)) e como diferenças entre ASTs, representadas como objetos JSON (JSON.org, 2014).

Tomando a representação textual dos *patches*, Tanto a distância quanto o tamanho dos *patches* têm correlação negativa com o tamanho do programa (teste de Spearman). A distância média entre *patches* (rs = -0,46) e a mediana da distância entre *patches* (rs = -0,51) apresentam correlações negativas e moderadas, enquanto a média do tamanho do *patch* (rs = -0,85) e o tamanho médio do *patch* (rs = -0,92) mostram correlações negativas e fortes com o tamanho do programa original (Cohen, 1992). Portanto, quanto maior o programa, menores serão os seus *patches* e eles estarão mais próximos um do outro. O padrão recorrente de ter uma distância média maior do que a mediana da distância revela a existência de alguns trechos distantes que arrastam a média para cima, enquanto um número maior de trechos na mesma região do código-fonte mantém a mediana baixa. O mesmo padrão é observado para o tamanho do *patch*: a busca local encontra alguns *patches* grandes, mas a maioria dos *patches* é pequena.

Tabela 3 mostra a distância média e mediana entre dois *patches* consecutivos encontrados pela busca local como percentual de tamanho do programa, medido em linhas de código. Linhas de código foram usadas nesta análise porque a simples diferença textual foi extraída do código não minificado e, portanto, legível por um ser humano. A legibilidade permite investigar tipos

de mudanças realizadas pela busca local. A tabela também mostra o tamanho médio, a mediana do tamanho desses *patches*, medidos em percentuais do tamanho de programa original em linhas de código.

Tanto a distância quanto o tamanho dos *patches* têm correlação negativa com o tamanho do programa (teste de Spearman). A distância média entre *patches* (r_s = -0,46) e a mediana da distância entre *patches* (r_s = -0,51) apresentam correlações negativas e moderadas, enquanto a média do tamanho do *patch* (r_s = -0,85) e o tamanho médio do *patch* (r_s = -0,92) mostram correlações negativas e fortes com o tamanho do programa original (Cohen, 1992). Portanto, quanto maior o programa, menores serão os seus *patches* e eles estarão mais próximos um do outro. O padrão recorrente de ter uma distância média maior do que a mediana da distância revela a existência de alguns trechos distantes que arrastam a média para cima, enquanto um número maior de trechos na mesma região do código-fonte mantém a mediana baixa. O mesmo padrão é observado para o tamanho do *patch*: a busca local encontra alguns *patches* grandes, mas a maioria dos *patches* é pequena.

Tabela 3 A distância entre *patches* consecutivos encontrados pela busca local e o tamanho desses patches, ambos medidos como percentuais do tamanho do programa original em LOC.

	Distância		Tamanho	
Biblioteca	**Média**	**Mediana**	**Média**	**Mediana**
Browserify	13,0%	7,9%	0,22%	0,13%
Exectimer	18,7%	14,9%	1,09%	0,44%
jQuery	13,7%	10,3%	0,05%	0,01%
Lodash	2,0%	0,02%	0,04%	0,02%
Minimist	18,3%	16,8%	0,47%	0,43%
Plivo-node	6,8%	0,4%	0,21%	0,11%
Pug	16,6%	10,9%	1,38%	0,28%
Tleaf	12,6%	8,7%	0,94%	0,35%
Underscore	15,2%	12,4%	0,08%	0,07%
UUID	17,6%	10,4%	0,79%	0,42%
XML2js	18,8%	14,5%	0,27%	0,19%
Todos	6,9%	0,3%	0,27%	0,07%

Tabela 4 mostra o número de rodadas da busca local que encontraram zero *patches* (isto é, não conseguiram encontrar uma variante reduzida do programa original), um *patch*, dois *patches*, três *patches*, quatro *patches* e cinco ou mais *patches* na menor variante que passa em todos os casos de teste. A tabela mostra que o número de *patches* tende a crescer com o tamanho do programa. Isto é corroborado por uma correlação positiva e moderada (r_s = +0,56)

entre o número de rodadas encontrando cinco ou mais patches e o tamanho do programa. 22,4% (148) rodadas não conseguiram encontrar uma variante reduzida da biblioteca.

Tabela 4 Número de rodadas da busca local que encontrou zero, um, dois, três, quatro ou cinco ou mais *patches* no código-fonte.

Biblioteca	**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
%	22%	20%	17%	11%	10%	42%

Conforme mostrado na Tabela 4, podem ser necessárias várias mutações para encontrar uma variante otimizada de uma biblioteca. Se as mutações são selecionadas aleatoriamente pelo algoritmo genético, a chance de escolher mutações adequadas para uma dada solução é inversamente proporcional ao número de nós na AST do programa sob análise. As chances de aplicar todas as mutações corretas sobre o mesmo indivíduo são ainda menores, uma vez que, dado o mesmo número de avaliações dos casos de testes, o número de avaliações (o limite superior para o número de mutações) se espalha entre todos os indivíduos da população.

Portanto, os algoritmos genéticos contam com o operador de cruzamento para combinar diferentes indivíduos com mutações distintas. Mesmo se considerarmos que cada mutação independente aumenta a aptidão dos indivíduos, aumentando assim as chances de sua seleção para cruzamento, as chances de um cruzamento bem-sucedido (um que produza um indivíduo combinando todas as mutações de ambos os pais e passando por todos os casos de teste) são mínimas. Tomemos, por exemplo, o operador de cruzamento *Single Point Crossover*. Como os *patches* para códigos grandes são pequenos e agrupados, dados dois indivíduos com um *patch* correto em cada um, a probabilidade de selecionar o ponto exato para cortar ambas as árvores e produzir a combinação dos dois *patches* é inversamente proporcional ao quadrado do tamanho do código. Assim, concluímos que um operador de cruzamento genérico, como o *Single Point Crossover*, não é eficaz para combinar mutações independentes, reduzindo o algoritmo genético a um conjunto de buscas aleatórias paralelas, na perspectiva desse problema.

Tal limitação pode ser observada na Figura 9 que mostra, respectivamente, as versões original e otimizada da função *getTimeFieldValues* da biblioteca *uuid*. A versão otimizada foi gerada por meio de uma rodada da busca local. Ela removeu as expressões usadas para calcular as variáveis *ts* e *hm*, assim como o parâmetro recebido pela função (usado anteriormente nas expressões descartadas), além das propriedades *hi* e *timestamp* do objeto retornado pela função. Suponha, que ambas as versões passem por todos os casos de teste, apesar das mudanças no processo de cálculo (elas passam nos casos de teste, na verdade, mas voltaremos a isso depois).

O operador de mutação do algoritmo genético mencionado acima pode remover a expressão que determina o valor de *ts*. O mesmo se aplica à remoção da expressão que leva ao valor de *hm*. O problema com algoritmos genéticos neste contexto reside na recombinação dos *patches*: o algoritmo só pode produzir um novo individuo otimizado aplicando as duas mutações no mesmo indivíduo ou recombinando dois indivíduos afetados pelas mutações. Como tais mutações aparecem próximas umas das outras, o operador de cruzamento *Single Point Crossover* deve ser executado precisamente entre os nós da AST que representam as linhas 2 e 3 para produzir um indivíduo que tenha ambas as mutações. Como os pontos de corte de ambas as árvores pai são selecionados aleatoriamente, as chances de produzir um indivíduo com as duas mutações são muito pequenas.

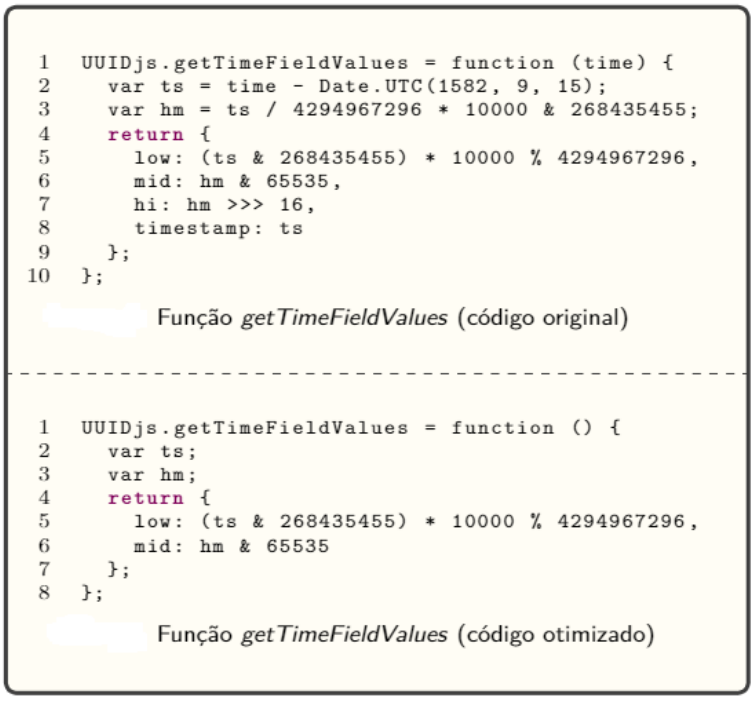

Figura 9 Função *getTimeFieldValues* original e otimizada, respectivamente

A dificuldade de aplicar o operador de cruzamento em problemas de melhoria de código fonte tem sido reconhecida por pesquisas na área ou aplicações mais amplas de algoritmos

genéticos. Harman et al. (Mark Harman, Langdon, et al., 2012) relatam que a operação de cruzamento de seu algoritmo de melhoramento genético foi construída para executar um determinado número de tentativas para localizar uma variante do código fonte que compila e passa em todos os casos de teste. Se o operador não gerar um indivíduo válido após um número de tentativas, uma operação de mutação é executada em seu lugar. Fogel e Atmar argumentam que o cruzamento é uma generalização de várias mutações combinadas, sendo ele próprio uma forma de mutação. Os autores afirmam que as mutações encontram resultados melhores do que o cruzamento (Fogel & Atmar, 1990).

Portanto, enquanto a busca local acumula mutações bem-sucedidas no mesmo indivíduo, um algoritmo genético baseado em uma AST tende a evoluir uma população na qual cada indivíduo carrega uma pequena parte das mutações. Se a operação de cruzamento não consegue reunir as partes de uma boa solução, a natureza acumulativa da busca local tende a examinar mutações diferentes nas mesmas partes da solução. Uma alternativa aos algoritmos genéticos seria uma busca sistemática percorrendo a AST original e analisando os efeitos da remoção de cada nó. Essa travessia seria capaz de gerar uma solução que acumule todos os *patches*. No entanto, analisar a remoção de todos os nós em um código fonte exige um esforço computacional considerável, sendo inviável para grandes bibliotecas. A próxima seção detalha a parte do experimento exploratório que teve como objetivo a descoberta de tipos de instrução que produzem as mudanças mais eficientes, visando reduzir o espaço de busca de interesse.

3.3.4 Tipos de instrução

Existem 53 tipos de instruções JavaScript de acordo com a norma ECMA-262[22]. Essas instruções preenchem os nós da AST de qualquer código JavaScript, enquanto as arestas dessas árvores representam diferentes relações entre os nós, como o bloco de instruções que devem ser executadas caso uma condição seja *verdadeira* ou a expressão que calcula o valor de uma determinada propriedade de um objeto.

O conjunto de todos os tipos de nós representa 100% do espaço de busca para uma travessia sistemática do espaço de busca do problema de redução do código-fonte JavaScript. No entanto, a frequência com que as instruções aparecem no código-fonte e seus impactos na redução do código-fonte variam. Embora o espaço de busca possa ser totalmente percorrido com um número razoável de avaliações para pequenos programas, pode ser impossível testar todas as alternativas para programas maiores.

[22] http://www.ecma-international.org/publications/standards/Ecma-262.htm

Para encontrar um equilíbrio entre a qualidade da solução e o esforço de processamento necessário para encontrar esta solução, usamos os resultados da busca local para encontrar a importância relativa de cada tipo de instrução. Enquanto a avaliação da seção anterior usou a diferença textual entre uma variante de código fonte encontrada pela busca local e o código original, nesta seção usamos a diferença calculada a partir da representação da AST, ou seja, o conjunto de ramificações da AST que estavam presentes no código original, mas que foram removidos de sua variante otimizada.

A Tabela 5 apresenta os tipos de instrução mais importantes encontrados nas modificações propostas pela busca local. Cada mutação realizada pela otimização remove apenas um nó (tipo de instrução), mas a remoção de tal nó pode implicar na eliminação de outros nós. Por exemplo, se um nó *IfStatement* for removido, os nós *BlockStatement* associados aos dois resultados de sua expressão condicional também serão removidos, junto com o nó *ConditionalExpression* que representa a condição em si. A coluna *Total* na Tabela 5 indica o número de vezes que um determinado tipo de nó foi o nó removido no nível mais alto da árvore de um *patch* encontrado pela busca local, considerando todos os programas observados.

A coluna *Frequência* mostra a frequência com que os tipos de nós aparecem nos programas JavaScript, de acordo com Rocha e Sobral (Silva, D., Sobral, 2017), que estudaram a estrutura do código JavaScript. Eles analisaram 34.000 programas escritos em JavaScript do repositório NPM[23] e, entre outras informações, relatam a frequência de ocorrência para cada tipo de instrução na especificação ECMA-262. Se quisermos selecionar os tipos de nó mais efetivos para ser examinados durante a travessia do espaço de busca pelo algoritmo de busca local, os nós que aparecem frequentemente nos programas JavaScript exigirão um maior número de avaliações dos casos de teste, enquanto os tipos raros de nó podem ser examinados com custo menor.

Finalmente, a coluna *Rank* apresenta uma proporção calculada de acordo com a equação 1 para qualquer tipo de nó N_t. Embora essa proporção não tenha significado físico, ela aumenta com os valores das colunas *Total* e *Frequência*. Tipos de nós que aparecem com frequência nos programas JavaScript e que aparecem como os mais nós mais altos nos *patches* encontrados pela busca local exibirão valores altos na coluna *Rank*. Valores menores indicam tipos de nó de baixa frequência, tanto na análise geral de programas JavaScript quanto nos melhores resultados em *patches* da busca local. A tabela é limitada a nós com um *Rank* maior que 0,50.

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

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

Tabela 5 Tipos de nós de programas JavaScript, sua frequência em programas e o número de vezes que eles foram selecionados como nós a ser removidos pela pesquisa local.

Tipos de nós mais frequentes	Total	Frequência	Rank
ExpressionStatement	1.561	5,75%	16,56
VariableDeclaration	994	1,87%	10,13
Literal	722	9,27%	7,96
Identifier	608	37,41%	9,71
ReturnStatement	482	1,48%	4,89
Property	389	2,10%	3,97
BinaryExpression	354	2,56%	3,63
FunctionDeclaration	317	0,48%	3,19
IfStatement	314	1,26%	3,18
CallExpression	250	8,18%	2,72
MemberExpression	151	11,84%	1,71
VariableDeclarator	96	2,25%	0,98
ObjectExpression	90	1,07%	0,91
FunctionExpression	89	2,56%	0,91
ArrayExpression	84	0,70%	0,85
UnaryExpression	82	0,73%	0,83
AssignmentExpression	61	2,00%	0,62
ConditionalExpression	57	0,21%	0,57

Os tipos de nós apresentados na Tabela 5 representam 91,74% dos nós de um programa JavaScript escolhido aleatoriamente. Se considerarmos todos estes nós, a redução no espaço de busca não é significativa (8,26%). Para programas grandes, o Otimizador não deveria analisar os nós que contêm *Identifiers* e *MemberExpressions*, pois esses nós geralmente ocorrem como parte de uma expressão maior. Ao remover esses tipos de nós do escopo do algoritmo, o espaço de busca é reduzido para 42,49% do tamanho da árvore. Tal redução permite uma travessia sistemática pelo espaço de busca usando menos esforço de processamento do que a análise de todo o espaço. A sistemática final proposta por esta Tese será detalhada no Capítulo 4.

3.4 Considerações finais

Conforme visto nesse capítulo, os possíveis cenários de uso do JavaScript e suas características trazem desafios singulares para aplicação de heurísticas paras otimização de código-fonte. Encontrar um conjunto de alterações que reduzam o tamanho de programas JavaScript utilizando a suíte de casos de teste como oráculo para avaliar a corretude em um

cenário de execução mundial pode trazer retorno em grande escala, mesmo para melhorias simples. Uma instrução removida em uma biblioteca que é executada em milhões de computadores por dia pode gerar uma economia muito grande globalmente.

Durante dois anos de pesquisa foi empreendido grande esforço para construir um ferramental de suporte que permitisse observar a aplicação de melhoramento genético nos cenários de execução de bibliotecas JavaScript. De posse deste ferramental, experimentos exploratórios foram realizados para entender a distribuição das soluções no espaço de busca e ajudar na criação de um algoritmo de busca sistemática para redução de código fonte escrito em JavaScript. Foi mostrado que nessa perspectiva de otimização, os algoritmos genéticos baseados na representação de árvore não são eficazes e perdem em qualidade das soluções encontradas quando comparados com uma busca local simples. O Apêndice A detalha como reproduzir os estudos e utilizar o ferramental em novas bibliotecas.

No Capítulo 4 serão apresentados a proposta da técnica, os resultados observados em experimento final onde a proposta foi aplicada em 19 bibliotecas em JavaScript.

4 Uma técnica automatizada para redução do tamanho de programas JavaScript utilizando busca local

Neste capítulo, apresentaremos a técnica para redução de tamanho de código-fonte escrito em JavaScript, o plano experimental para observar os efeitos da técnica em 19 bibliotecas e os resultados obtidos. Estes resultados serão discutidos sob as perspectivas qualitativa e quantitativa, assim como das ameaças a validade.

4.1 Introdução

Neste capítulo será apresentado o processo de otimização automatizado para reduzir o tamanho de programas JavaScript usando busca local como o algoritmo central de otimização. Após os estudos exploratórios e o estudo sobre tamanho e frequência de instruções em JavaScript descritos no capítulo anterior, conduzimos estudos experimentais finais para avaliação da técnica de redução de tamanho de programas JavaScript baseada nesses resultados prévios. O objetivo foi observar a aplicação de configurações distintas da busca local em um maior número de bibliotecas, com características que pudessem suportar a tomada de decisão sobre qual heurística recomendar para a redução de programas JavaScript.

O capítulo está dividido da seguinte maneira: a seção 4.2 detalha o algoritmo proposto e suas duas configurações. A seção 4.3 descreve o estudo experimental conduzido, suas questões de pesquisa, que programas foram observados e quais são as características destes programas. A seção 4.4 discute os resultados encontrados sob a perspectiva das questões de pesquisa, dividindo a discussão destes resultados em uma análise quantitativa e uma análise qualitativa. Também são discutidas as ameaças à validade do estudo. Por fim, a seção 4.5 mostra as considerações finais relacionadas ao estudo experimental.

4.2 Um algoritmo para redução de instruções em programas JavaScript

O principal desafio para aplicar busca local na melhoria de código-fonte em JavaScript é o tamanho da vizinhança para qualquer programa de tamanho não trivial. Ao manipular um

programa através de sua AST, um vizinho pode ser qualquer variante da AST original na qual um nó é adicionado ou removido. Dada a diversidade de nós na AST de código JavaScript e o tamanho médio de um programa JavaScript (3.342 linhas de código (Silva, D., Sobral, 2017)), construir toda a vizinhança de um programa pode esgotar a quantidade de tempo e o poder de computação disponível para a otimização. Para resolver o problema de tamanho de vizinhança, aproveitamos os resultados descritos na Seção 3.3 para determinar os tipos de nós que levam a melhorias mais extensas quando removidos das árvores de sintaxe dos programas sob análise. Em seguida, usamos esses nós para criar um operador de seleção de vizinhança para um procedimento de melhoria do código-fonte com base em um algoritmo de busca local.

O algoritmo proposto é um FAHC (*First-Ascent Hill Climbing*) que usa o código-fonte do programa alvo, representado por sua AST, como uma solução inicial. O operador de geração de vizinhança do FAHC remove um nó da AST representando a solução atual, eliminando todos os nós conectados ao selecionado. Embora a solução resultante possa diferir da solução atual por vários nós (a remoção de um nó pode provocar o descarte de uma ramificação da AST contendo vários nós), a AST resultante é considerada uma variante vizinha porque somente uma instrução foi removida diretamente pelo algoritmo de busca.

A busca local proposta tem duas variações em sua configuração: o DFAHC e o SFAHC. Ambas seguem a mesma sistemática de busca, a saber: dado um orçamento de avaliações de B suítes de casos de teste, a busca local segue a ordem dos tipos de nós apresentados na Tabela 5. Para um determinado tipo de nó, identifica todas as suas ocorrências no programa e testa, uma a uma, sua remoção do código fonte. Toda vez que um nó é removido, uma variante do programa é criada e todos os casos de teste projetados para o programa são executados nessa variante.

Se a variante passar em todos os casos de teste, ela será considerada válida, será selecionada como a solução atual e a busca continuará a partir da sua AST. Caso contrário, o processo de busca restaura a ramificação removida da AST e move-se para a próxima ocorrência do tipo de nó. Depois de esgotar todas as ocorrências deste tipo de nó, o próximo tipo de nó é selecionado. Depois de esgotar todas as ocorrências de todos os tipos de nós, a busca é reiniciada a partir do primeiro tipo de nó. A pesquisa é interrompida após uma rodada completa sem melhoria ou até esgotar o orçamento de avaliações. O SFAHC seleciona aleatoriamente o próximo tipo de nó a ser examinado para remoção e sua ocorrência no programa original, enquanto o DFAHC segue estritamente a ordem estabelecida na Tabela 5 e, para um determinado tipo de nó, segue a ordem estabelecida no código-fonte.

Como descrito na Seção 2.1, são três as principais decisões quando se aplica algoritmos heurísticos em problemas de Engenharia de Software. Nessa pesquisa, as decisões foram:

1. Representação: os indivíduos são representados através de ASTs de seus códigos fonte. Essa decisão foi apoiada pelo fato da pesquisa prévia observar os resultados por instrução de código. Portanto, uma representação de mais alto nível, como linhas de código ou gramática, não permitiria a observação dos resultados no nível de granularidade desejado;

2. Função objetivo: a função escolhida foi o número de caracteres observados após a minificação do código, conforme descrito na Seção 3.3. Esta função somente é aplicada em variantes de um programa original que passem em todos os casos de teste desenvolvidos para o programa original;

3. Operadores: apenas o operador que remove um nó da AST de um programa foi escolhido para a busca local que reduz o tamanho de programas JavaScript. A seleção do nó a ser removido depende do algoritmo selecionado (DFAHC ou SFAHC).

4.3 Avaliação do algoritmo proposto

Para avaliar os algoritmos propostos na seção 4.2, construímos um projeto experimental para observar os resultados da aplicação das duas configurações do algoritmo de busca local (DFAHC e SFAHC) em 19 bibliotecas JavaScript que variam em tamanho, finalidade e cobertura. O algoritmo SHC, descrito nas análises do Capítulo 3, foi utilizado como base de comparação para as duas configurações de busca local. Este algoritmo escolhe aleatoriamente a próxima instrução a ser removida da AST do programa que está sendo otimizado.

Adicionamos seis novas bibliotecas ao grupo já observado no Capítulo 3 e definimos a perspectiva de observação com três perguntas de pesquisa. Após a execução dos estudos experimentais, conduzimos uma criteriosa análise, onde é possível observar os resultados encontrados e fazer comparações entre as configurações. Além disso, agrupamos as mudanças encontradas por tipo de instrução e discutimos qualitativamente cada caso, com base em exemplos.

SHC e SFAHC, ambos com componentes aleatórios, foram executados dez vezes para lidar com variações na qualidade da solução devido à sua natureza estocástica. DFAHC, sendo um algoritmo determinístico, foi executado uma única vez. Os algoritmos foram executados no mesmo ambiente de computação de alto desempenho descrito na Seção 3, porém com menor disponibilidade de recursos. Assim, reduzimos o número de ciclos de otimização de 60 para dez. Todos os algoritmos receberam um orçamento de 5.000 avaliações do conjunto de testes.

4.3.1 Questões de pesquisa

A avaliação dos estudos experimentais supracitados será pautada pelas seguintes questões de pesquisa:

- RQ1: Como os resultados do DFAHC se comparam com os resultados do SHC em relação à qualidade da solução?

  O algoritmo SHC superou os algoritmos genéticos nas análises realizadas no Capítulo 3 e o DFAHC usa informações derivadas das melhores soluções encontradas pela SHC. Portanto, usamos SHC como uma base para comparação para o DFAHC e esperamos que este último supere o primeiro na busca por variantes melhoradas para os programas selecionados.

- RQ2: Como os resultados do DFAHC se comparam com os resultados do SFAHC em relação à qualidade da solução?

  O DFAHC estabelece uma ordem sob a qual os tipos de nós são analisados. É possível que analisar estes tipos de nós em uma ordem aleatória leve a resultados melhores do que a ordem dada na Tabela 5. Essa questão de pesquisa avalia se a adição de um componente aleatório na ordem de seleção de tipo de nó melhora o desempenho do DFAHC.

- RQ3: Como os resultados do DFAHC se comportam para diferentes tamanhos de programa, cobertura e tamanho de suíte de testes?

  Nesta questão de pesquisa analisamos a influência das características do programa que está sendo otimizado no desempenho do DFAHC. Esperamos que o desempenho diminua para programas maiores, com suítes de teste menores ou apresentando cobertura mais baixa.

Usar testes de unidade como oráculo pode levar a resultados inválidos devido a conjuntos de testes incompletos ou de baixa qualidade. A cobertura de teste mede o percentual de linhas de código-fonte que são executadas pelo conjunto de testes. No entanto, a qualidade de um conjunto de testes é muito mais difícil de determinar, pois mesmo a execução de todas as linhas de código não garante que os resultados de sua execução sejam verificados pelos casos de teste ou que essas linhas foram executadas em cenários diversos o suficiente para capturar todas as variações da funcionalidade esperada do software.

Portanto, além da análise quantitativa realizada para responder às questões de pesquisa acima, apresentamos uma análise qualitativa que examina o código removido pelo otimizador e discute se é uma melhoria válida ou que só foi possível devido a limitações da suíte de casos de teste. Em seguida, o uso da abordagem proposta para melhorar o código, o conjunto de testes ou

ambos são discutidos à luz dos conjuntos de testes projetados para detectar se as funções do software são preservadas após algumas modificações, permitindo que o otimizador sugira alterações que quebram esses recursos.

4.3.2 Programas observados

Os onze programas JavaScript usados no experimento descrito no capítulo 3 foram selecionados como sujeitos para os experimentos descritos neste capítulo, juntamente com oito novos programas. Mais uma vez, selecionamos apenas bibliotecas JavaScript muito usadas, com suítes de teste com pelo menos 90% de cobertura. Os novos programas estão listados abaixo e suas características são apresentadas na Tabela .

- ***D3-Node*** é uma fachada para acessar métodos de biblioteca D3 no lado servidor de uma aplicação Web ou aplicativo móvel;
- ***Decimal*** define um objeto decimal de precisão arbitrária e suas operações para serem utilizados em programas JavaScript;
- ***Esprima*** é um analisador ECMAScript, escrito em JavaScript, de alto desempenho e compatível com os padrões[24] internacionais que regem o ECMAScript;
- ***Express*** é uma estrutura mínima e flexível do Node.JS que fornece recursos para aplicações Web e aplicativos móveis;
- ***Mathjs*** é uma extensa biblioteca de operações matemáticas para programas JavaScript;
- ***Moment*** analisa, valida, manipula e exibe informações de data e hora;
- ***Node-semver*** é uma implementação JavaScript da especificação da *Semantic Versioning 4*;
- ***UglifyJs2*** é um kit de ferramentas de análise sintática, minificação, compressão e embelezamento (*pretty-printing)* de código-fonte JavaScript.

[24] https://www.ecma-international.org/publications/standards/Ecma-262.htm

Tabela 6. Principais características dos programas adicionados ao estudo experimental. A primeira coluna mostra o número de linhas de código em cada programa; em seguida, o número de casos de teste em sua suíte, o percentual de comandos cobertos pelos testes, o número de vezes que cada programa foi baixado em abril de 2018 (dividido por 1.000) e a versão selecionada para melhoria (n/a na falta de números de versão).

Programa	LOC	# Testes	% Cobertura	Uso	Versão
D3-node	92	19	100%	17	n/a
Decimal	1.595	22.509	96%	630	9.0.1
Esprima	6.615	1.352	100%	46	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%	5472	2.17.1
Node-semver	1.324	2.057	99%	46.665	5.5.0
UglifyJs2	4.098	172	91%	2	3.1.2

4.4 Análise dos resultados

Para responder às questões de pesquisa e permitir uma comparação entre os resultados obtidos para cada configuração da busca local, dividimos a análise em duas partes. Na primeira, a análise quantitativa dos resultados, discutimos os valores das medidas encontradas sob a perspectiva das questões de pesquisa. Já na análise qualitativa, dividimos os tipos de modificação encontrados pelos algoritmos em grupos e discutimos cada grupo com base em exemplos e no tratamento de cada situação.

4.4.1 Análise quantitativa

A Tabela 7 mostra os resultados encontrados para os 19 programas usando os três algoritmos. Cada porcentagem mostra a redução no número de caracteres como um percentual do tamanho do programa original. A tabela mostra a melhoria encontrada pelo DFAHC, bem como a mediana, média, desvio padrão e máxima melhoria encontrada pelo SFAHC e pelo SHC. Todos os algoritmos conseguiram encontrar variantes menores e viáveis para todos os programas analisados como parte do experimento. As distribuições de frequência da qualidade das soluções encontradas por cada algoritmo são apresentadas na Figura 10.

Tabela 7 Melhoria de tamanho dos programas JavaScript encontrada pelo DFAHC, mediana, média (com desvio padrão) e redução máxima encontrada pelo SFAHC e pelo SHC em dez rodadas de otimização.

	Mediana			Média/Desvio		Máximo	
Programa	**DFAHC**	**SFAHC**	**SHC**	**SFAHC**	**SHC**	**SFAHC**	**SHC**
Browserify	26,60%	1,60%	13,20%	1,7% ± 0,0%	12,8% ± 2,9%	1,70%	16,30%
D3-node	29,30%	29,30%	25,50%	29,3% ± 0,0%	25,5% ± 0,0%	29,30%	25,50%
Decimal	7,10%	7,20%	4,80%	8,8% ± 4,4%	4,4% ± 1,8%	18,10%	6,80%
Esprima	4,00%	4,90%	0,30%	4,8% ± 0,2%	0,4% ± 0,1%	4,90%	0,60%
Exectimer	22,80%	22,80%	20,80%	22,8% ± 0,0%	20,7% ± 0,8%	22,80%	21,40%
Express	3,40%	8,00%	4,30%	7,6% ± 1,5%	4,4% ± 0,7%	8,40%	5,70%
Jquery	58,30%	58,40%	17,30%	58,9% ± 1,5%	17,8% ± 3,3%	62,50%	25,40%
Lodash	2,70%	5,90%	0,60%	5,5% ± 0,9%	0,6% ± 0,1%	6,00%	0,80%
Mathjs	9,80%	27,80%	15,80%	22,1% ± 9,8%	15,8% ± 0,9%	30,40%	17,20%
Minimist	2,80%	2,80%	1,80%	2,7% ± 0,1%	1,8% ± 0,1%	2,80%	1,90%
Moment	7,10%	5,50%	1,40%	5,5% ± 0,0%	1,5% ± 0,2%	5,50%	1,90%
Node-semver	0,20%	3,00%	4,10%	3,2% ± 1,9%	4,1% ± 0,46%	6,80%	5,00%
Plivo-node	33,20%	15,60%	55,50%	16,7% ± 5,0%	5,2% ± 2,5%	23,40%	58,50%
Pug	30,90%	31,40%	23,90%	31,4% ± 0,0%	23,3% ± 1,7%	31,40%	26,10%
Tleaf	73,80%	73,80%	69,50%	73,8% ± 0,0%	69,2% ± 0,6%	73,80%	69,60%
UglifyJs2	12,10%	10,90%	2,70%	10,2% ± 3,9,%	2,7% ± 0,5%	15,30%	3,70%
Underscore	10,00%	5,10%	3,70%	5,1% ± 0,0%	3,8% ± 1,2%	5,10%	6,60%
UUID	24,30%	23,30%	23,20%	22,3% ± 3,3%	23,4% ± 2,0%	24,70%	26,40%
XML2js	5,00%	2,00%	8,60%	2,3% ± 0,9%	9,2% ± 5,0%	4,60%	15,20%

Embora os algoritmos tenham encontrado melhorias consistentes para alguns programas, foram encontradas melhorias substanciais para outros. Por exemplo, é difícil acreditar que 58,3% do código-fonte de um programa JavaScript, como *Jquery*, possa ser removido sem afetar sua funcionalidade. No entanto, essa variante passa em todos os casos de teste desenvolvidos para o programa.

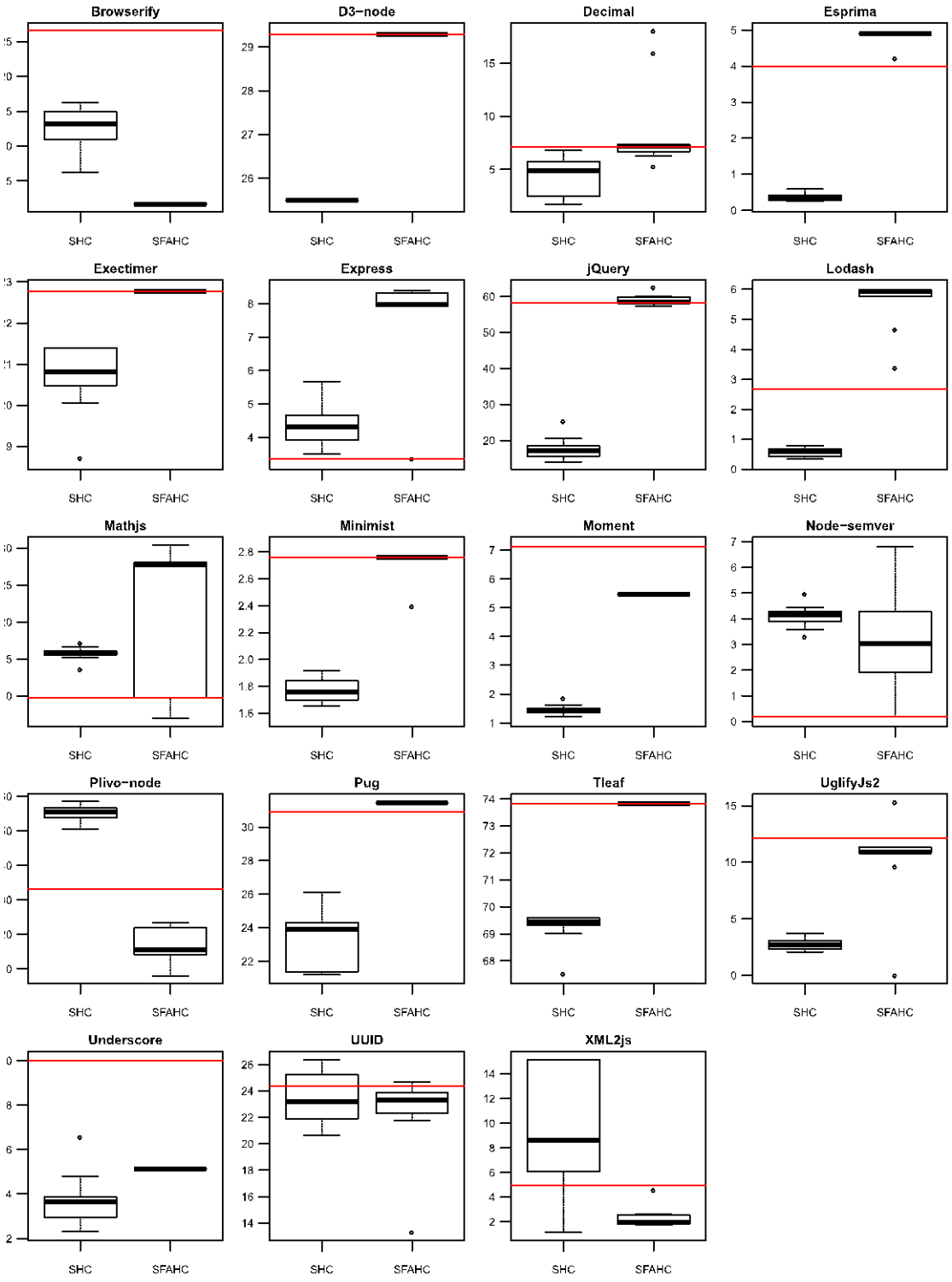

Figura 10 Distribuição das melhorias encontradas por diferentes algoritmos em dez rodadas de otimização. O programa alvo da otimização é representado por zero no eixo y nos gráficos, enquanto a linha horizontal representa a variante otimizada encontrada pelo DFAHC.

Para responder à RQ1, observamos que todas as soluções encontradas pelo SHC representam reduções menores do que as encontradas pelo DFAHC em 13 dos 19 programas.

Por outro lado, todas as soluções encontradas pelo SHC foram superiores (reduções maiores) às encontradas pelo DFAHC para quatro programas. Os *boxplots* que representam as soluções encontradas pelo SHC cruzam a linha que representa a variante encontrada pelo DFAHC duas vezes (para UUID e XML2js). Empregamos um teste de Wilcoxon (alfa = 5%) para determinar se as distribuições para esses casos eram significativamente diferentes da respectiva solução produzida pelo DFAHC e descobrimos que a distribuição não é significativamente diferente para o UUID (*p-value* = 0,160), mas é significativamente diferente para XML2js (*p-value* = 0,027). Concluímos então que o SHC supera o DFAHC em cinco programas, enquanto o oposto ocorre em 13 programas, com um empate. Para as cinco instâncias em que superou o DFAHC, o SHC removeu nós cujos tipos não estão indicados na Tabela 5, incluindo *BreakStatement*, *ThisExpression* e *BlockStatement*.

Para responder à RQ2, observamos que todas as soluções encontradas pelo SFAHC representam reduções menores do que as encontradas pelo DFAHC em 5 dos 19 programas. Todos os resultados favorecem o SFAHC em três programas. Um empate perfeito é observado para três programas (*D3*, *Exectimer* e *Tleaf*), onde encontramos os mesmos resultados em todas as rodadas de otimização tanto para o DFAHC quanto para o SFAHC. Finalmente, os *boxplots* do SFAHC cruzam a linha que representa a solução encontrada pelo DFACH para oito programas, levando a diferenças significativas para o *Express*, *Mathjs*, *Node-semver* e *UUID* (teste de Wilcoxon com nível de confiança de 5%), os três primeiros representando vitória para o SFAHC, enquanto o último representa uma vitória para o DFAHC. Portanto, não encontramos diferença definitiva no desempenho dos algoritmos, cada um superando o outro para seis programas e ambos encontrando resultados semelhantes para sete instâncias.

Depois de atingir um ótimo local, tanto o DFAHC quanto o SFAHC podem reiniciar a otimização se ainda houver orçamento para avaliações. O SFAHC altera aleatoriamente a ordem de avaliação do tipo de instrução ao reiniciar, enquanto o DFAHC segue a mesma ordem usada no primeiro ciclo de otimização. Devido a esse comportamento, o SFAHC conseguiu remover mais instruções do que o DFAHC em alguns casos. A Tabela 8 apresenta os programas nos quais estes resultados foram observados e o número de instruções removidas por cada algoritmo. Enquanto algumas diferenças são pequenas (por exemplo, *Esprima*), alguns programas são positivamente influenciados pela aleatoriedade introduzida no DFAHC (por exemplo, *Node-semver*). A introdução da aleatoriedade leva a perda de previsibilidade das remoções que serão realizadas pelo algoritmo (predeterminada no DFAHC), mas pode ser usada em programas nos quais os resultados determinísticos fornecidos pelo DFAHC são restritos devido a limites de tempo e recursos que seriam necessários para explorar toda a vizinhança.

Tabela 8 Número de instruções removidas pelo DFAHC e pelo SFAHC de cada programa no qual o SFAHC conseguiu encontrar soluções menores do que o DFAHC

Programa	SFAHC	DFAHC
Esprima	187	174
Express	151	54
Lodash	743	111
Mathjs	621	321
Node-Semver	101	6
PUG	108	81

Para responder à RQ3, calculamos a correlação (*Spearman rank-order correlation*) entre a melhoria observada na variante produzida pelo DFAHC e o tamanho do programa sob análise (r_s=−0,36), o tamanho do conjunto de testes (r_s=−0,48) e o percentual de cobertura de linhas de código do conjunto de testes (r_s=−0,49). Todas as correlações são moderadas e negativas (Cohen, 1992), ou seja, quanto maior o programa, o conjunto de testes ou a cobertura, menor a melhoria encontrada pelo algoritmo DFAHC.

Analisamos também se a diferença entre a mediana das melhorias encontradas pelo SFAHC e a melhoria produzida pelo DFAHC está correlacionada com as mesmas medidas (tamanho do programa, tamanho do conjunto de testes e percentual de cobertura). Observamos correlações positivas e moderadas para todas as medidas (Cohen, 1992). A correlação mais forte é com o número de casos de teste (r_s=0,57), enquanto o tamanho do programa e a cobertura dos testes são mais fracamente correlacionados com a diferença entre os resultados dos algoritmos (r_s=0,32 e r_s=0,30, respectivamente). Portanto, para programas com suítes de testes maiores, código-fonte maior e cobertura de testes mais ampla, o componente estocástico que diferencia o SFAHC do DFAHC faz diferença no desempenho do algoritmo.

É difícil determinar o tempo necessário para a otimização porque os programas variam em tamanho e no tempo necessário para executar seus casos de teste. O ambiente de HPC apresenta desafios próprios para a medição do tempo de processamento devido ao *overhead* de comunicação necessário para a paralelização do ciclo de otimização e a competição com outros *jobs* por recursos. Se os algoritmos fossem executados em um único processador do ambiente HPC, os 399 ciclos de otimização realizados como parte deste experimento exigiriam cerca de 600 dias de processamento de dados. Este custo de processamento considerável levou à análise de uma busca local determinística, que poderia ser executada em um momento viável para programas de tamanho médio em computadores convencionais com memória suficiente. Conforme observado nas análises acima, não há perda significativa em manter a busca

determinística para programas de tamanho pequeno e médio ou com tamanho e cobertura de testes limitados.

4.4.2 Análise qualitativa

Concluída a análise quantitativa, examinamos as mudanças encontradas pela busca local para os programas que foram submetidos à otimização. Esta análise tem como objetivo determinar a existência de padrões que possam fornecer orientação para os desenvolvedores sobre como codificar ou testar melhor seus programas JavaScript, bem como instruções para melhorar o Otimizador em si. Agrupamos as observações pelo tipo de instrução removida, conforme a classificação abaixo.

Use strict: a diretiva "*use strict*" foi introduzida no JavaScript 1.8.5 para indicar que o código deve ser executado sob certas restrições, como declarar explicitamente todas as variáveis e usar nomes de parâmetros exclusivos em funções. O Otimizador removeu a diretiva "*use strict*", já que sua ausência não interfere na execução dos casos de teste.

Funções não utilizadas: o Otimizador removeu diversas funções dos programas analisados. Em certas situações, o Otimizador removeu o código inteiro; em outras, deixou apenas a assinatura da função. Por exemplo, as funções *fromURN* e *fromBytes* foram removidas inteiramente do programa *UUID*, enquanto todas as instruções da função *fromBinary* foram removidas, conforme pode ser visto na Figura 11. Os casos de teste não executam as duas primeiras funções. A última função foi executada pelos casos de teste, mas seus valores de retorno não são verificados por assertivas nestes casos de teste. Assim, as duas primeiras funções foram removidas completamente e apenas a assinatura da última foi preservada, de modo que pudesse ser corretamente executada pelos casos de teste. Esse resultado reforça que o Otimizador pode dar suporte ao testador, mostrando maneiras alternativas de testar o programa. Para confirmar se o problema neste cenário é a falta de cobertura das funções pelos casos de teste, escrevemos um novo caso de teste para a função *fromURN* e o submetemos aos desenvolvedores do *UUID* por meio de um pull request[25]. Os autores não aceitaram nem rejeitaram a solicitação *pull* até o presente momento. Porém, o projeto não possui *commits* e interação dos desenvolvedores desde 05 de junho de 2017.

[25] https://github.com/pnegri/uuid-js/pull/20

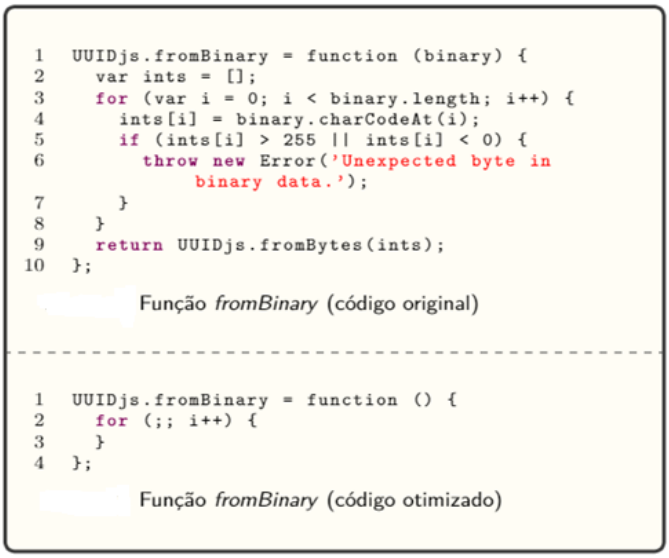

Figura 11 Código da função *fromBinary*

Parâmetros de função: em quase todos os programas, observamos a remoção de parâmetros de funções. Em alguns cenários, os parâmetros são removidos somente da assinatura da função; em outros, eles também são removidos das chamadas de função. Um exemplo pode ser visto na Figura 12, em que o parâmetro *time* foi removido da assinatura da função *fromTime*. A maioria desses parâmetros não é usada pelas funções ou possui um valor *default*, fazendo com que os casos de teste sejam executados corretamente mesmo quando o parâmetro não é passado para a função. Desta forma, o Otimizador remove o parâmetro para reduzir o tamanho do programa. Esse tipo de modificação precisa ser verificado por um desenvolvedor para cada situação. A ausência de cobertura nos casos de teste ou a não verificação dos retornos de funções pelos testes, que muitas vezes não comparam o valor retornado pela função com o valor esperado após a execução da mesma, pode levar a esse tipo de otimização. É importante notar que a ausência de cobertura aqui mencionada não se trata de ausência de cobertura de comandos, pois as funções são invocadas pelo código, mas a ausência de diversidade nos valores dos parâmetros ou no próprio uso destes parâmetros pelos casos de teste.

Chamadas de função: como no grupo de alterações relacionadas aos parâmetros de função, em quase todos os programas observamos que determinadas invocações de função foram removidas, ou seja, a execução de tais funções não interfere nos resultados dos casos de teste. Assim como no caso anterior, esse tipo de mudança realizada pelo Otimizador está relacionado com a falta de cobertura dos casos de teste, levando o Otimizador a remover completamente a declaração de função e suas invocações. Um exemplo pode ser visto na Figura 13, onde temos o código parcial da função *formatUnits* da biblioteca *Mathjs*.

```
1  UUIDjs.fromTime = function (time, last) {
2      last = !last ? false : last;
3      var tf = UUIDjs.getTimeFieldValues(time);
4      var tl = tf.low;
5      var thav = tf.hi & 4095 | 4096;
6      // set version '0001'
7      if (last === false) {
8          return new UUIDjs().fromParts(tl, tf.
               mid, thav, 0, 0, 0);
9      } else {
10         return new UUIDjs().fromParts(tl, tf.
               mid, thav, 128 | UUIDjs.limitUI06,
                UUIDjs.limitUI08 - 1, UUIDjs.
               limitUI48 - 1);
11     }
12 };
```

Função *fromTime* (código original)

```
1  UUIDjs.fromTime = function (last) {
2      var tf = UUIDjs.getTimeFieldValues();
3      var tl = tf.low;
4      var thav = tf.hi & 4095 | 4096;
5      // set version '0001'
6      if (last === false) {
7      } else {
8          return new UUIDjs().fromParts(tl, tf.
               mid, thav, 128 | UUIDjs.limitUI06,
                UUIDjs.limitUI08 - 1, UUIDjs.
               limitUI48 - 1);
9      }
10 };
```

Função *fromTime* (código otimizado)

Figura 12 Código da função *fromTime*

```
1  Unit.prototype.formatUnits = function () {
2      // Lazy evaluation of the unit list
3      this.simplifyUnitListLazy();
4      var strNum = '';
5      var strDen = '';
```

início da função *formatUnits* (código original)

```
1  Unit.prototype.formatUnits = function () {
2      var strNum = '';
3      var strDen = '';
4      var nNum = 0;
5      var nDen = 0;
```

início função *formatUnits* (código otimizado)

Figura 13 Código parcial da função *formatUnits*

Valores *default*: alguns dos programas selecionados para o experimento, como *D3-node*, *TLeaf* e *Pug*, possuem valores *default* que são utilizados no caso de uma propriedade de um objeto ser exigida para sua execução e não estar disponível. Nesses casos, a propriedade solicitada assume

o valor *default*. Usando o *D3-node* como exemplo, quase todas as definições de valores *default* foram removidas pelo Otimizador. Um exemplo pode ser visto na Figura 14, onde as propriedades *selector*, *container* e *styles* foram removidas do objeto. Seus valores *default* eram todos vazios. Isso foi possível porque os casos de teste do *D3-node* **sempre** passam valores para as respectivas propriedades e verificam os resultados das funções que eles chamam. Esses programas não possuem casos de teste para exercer o uso de valores *default*. Esse grupo de modificações pode ser classificado como falta de cobertura específica dos casos de teste: o código é testado, mas os casos de teste não podem impedir que o Otimizador remova as instruções. Novamente, enviamos um *pull request*[26] ao autor do programa com nossa sugestão de melhoria nos testes. O desenvolvedor aceitou os novos casos de teste como parte do conjunto de testes do programa. Isso nos forneceu a primeira evidência de que o Otimizador pode ajudar a melhorar a qualidade dos casos de teste de um programa JavaScript.

```
1  const defaults = {
2      d3Module: require('d3'),
3      // to allow use of d3.v4
4      selector: '',
5      // selects base D3 Element
6      container: '',
7      // markup inserted in body
8      styles: ''     // embedded svg stylesheets
9  };
```

constante *defaults* (código original)

```
1  const defaults = { d3Module: require('d3') };
```

constante *defaults* (código otimizado)

Figura 14 Código da constante *defaults* da biblioteca *D3-node*

Propriedades em objetos: programadores JavaScript utilizam objetos como dicionários de valor-chave para armazenar valores temporários na memória. Nesse caso, uma maneira de liberar memória é remover as chaves não usadas desses objetos, aplicando a instrução *delete*. Por causa da natureza desta instrução, onde uma propriedade ou um objeto são removidos do escopo de uma função, muitos casos de teste não são influenciados por ela (ou seja, não testam a existência do objeto ou propriedade). Esse tipo de código foi removido pelo Otimizador, como pode ser visto nos programas *Moment*, *Lodash* (Figura 14), *Underscore*, *UUID* e *Esprima*. Mesmo que seja uma alteração correta (não limpe a memória), esse tipo de modificação aumenta o consumo de memória e deve ser avaliado por um desenvolvedor. O *pull request* citado na seção de funções não utilizadas inclui sugestões de testes para cobrir essa situação.

[26] https://github.com/d3-node/d3-node/pull/33

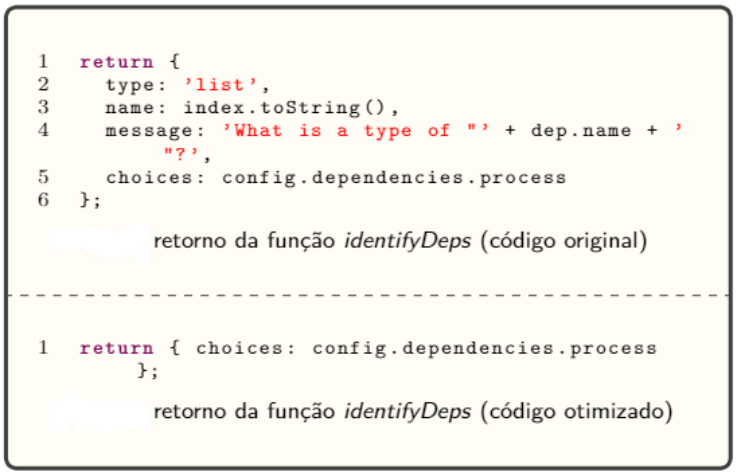

Figura 15 Código de retorno da função *identifyDeps* da biblioteca *Lodash*

Dependências: programas JavaScript escritos para o ambiente *NodeJs* usam a função *require* para incluir outros arquivos ou bibliotecas. Essa função foi removida de alguns programas, como *Mathjs*, *Pug* e *XML2JS*. Por exemplo, a dependência "*pug-strip-comments*" foi removida do *Pug*. Essa dependência manipula comentários dentro dos *templates* antes que a biblioteca analise os mesmos. A remoção da dependência não causa um erro no objeto destinado a receber o *template* sem comentários: a diferença no resultado é que os comentários não são removidos do *template*. Como os casos de testes da biblioteca não verificam se há comentários no *template*, eles são executados sem falhas. No entanto, essa alteração introduz um erro, pois o *template* é deixado com comentários que deveriam ter sido removidos pelo programa. Esse tipo de mudança pode ser considerado uma falta de cobertura que não pode ser capturada pela métrica de cobertura de comandos e, novamente, contribui com uma percepção sobre a baixa qualidade dos testes dos programas selecionados.

Retorno: a remoção de declarações *return* dentro de funções foi observada em quase todos os programas. O exemplo da Figura 16, foi extraído da função *parseNumber* da biblioteca *Mathjs*. Esse tipo de remoção refere-se diretamente à falta de cobertura, em que um código é executado com êxito, mas os testes de biblioteca não verificam o resultado retornado.

```
1                if (!isDigit(c)) {
2                    // this is no legal number, it
                         is just a dot
3                    revert(oldIndex);
4                    return null;
5                }
```

Listing 1. Código parcial da função *parseNumber* (código original)

```
1                if (!isDigit(c)) {
2                    // this is no legal number, it
                         is just a dot
3                    revert(oldIndex);
4                }
```

Listing 2. Código parcial da função *parseNumber* (código otimizado)

Figura 16 Função *parseNumber*

Condicionais: a remoção de comandos condicionais foi observada em quase todos os programas (blocos inteiros de *IF* ou algumas de suas condições). Na maioria dos casos, é possível observar a falta de cobertura para cenários de exceção, pois os casos de teste não verificam os cenários em que o tratamento de erros é ativado. Como exemplo, podemos observar a remoção de um *IF* da biblioteca *Minimist* (Figura 17). A condição removida verifica, na função *isNumber*, se o parâmetro enviado é um número. Esta função não possui casos de teste para cobrir esta condição diretamente: vários casos de teste exercitam seu código, mas nenhum deles usa um parâmetro não numérico, ou seja, a condição de exceção nunca é realmente testada. Para testar nossa hipótese sobre a qualidade dos testes incluímos na biblioteca um novo caso de teste onde passamos um parâmetro não numérico e verificamos se a biblioteca faz o tratamento de erro corretamente. Com esse novo teste, a Otimização não conseguiu remover a condição. A modificação foi submetida ao autor da biblioteca[27], mas não houve nenhum retorno do mesmo em relação ao *pull request*.

[27] https://github.com/substack/minimist/pull/125

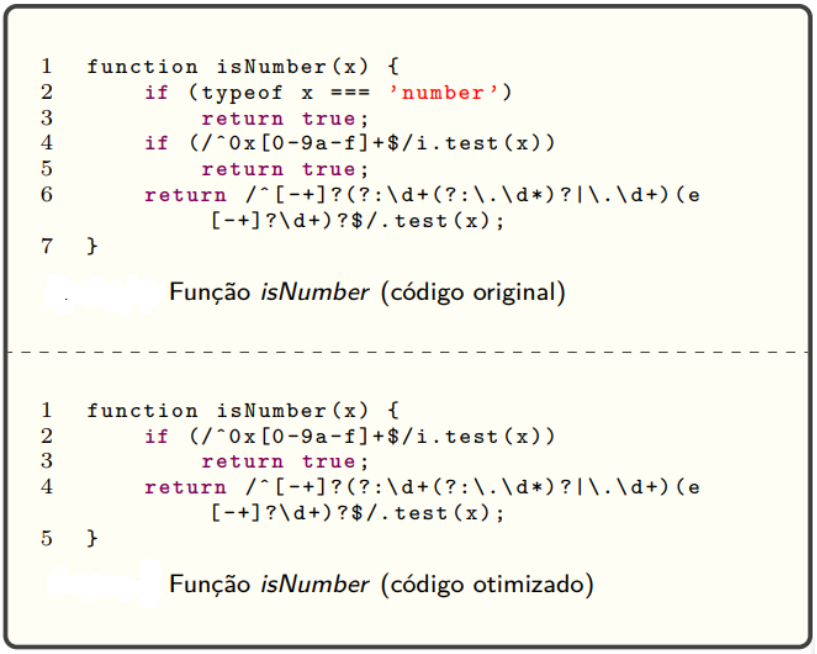

Figura 17 Função *isNumber* da biblioteca *Minimist*

Instruções não cobertas: exceto para *Browserify*, *Esprima*, *Express* e *Minimist*, todos os programas tinham códigos removidos porque os testes não os cobriam. Os desenvolvedores devem sempre verificar remoções decorrentes da falta de cobertura. Afinal, é fácil aceitar que pode haver código correto e útil como parte dessas remoções. No entanto, até mesmo código coberto pelos casos de teste foi removido. Esses trechos de código foram exercidos pelos testes, mas não fizeram nenhuma diferença para a sua execução. Um exemplo pode ser visto no *Exectimer*, no qual o Otimizador removeu um conjunto de instruções que ordenavam os dados dentro da função *median* (Figura 18). Todos os testes que cobriam esta função usaram valores ordenados, isto é, da perspectiva dos testes não houve necessidade de executar o código de ordenação. Como na prática a função pode ser utilizada com dados não ordenados, a alteração pode introduzir erros no programa. Para testar se o código removido estava de fato correto e o problema era a falta de cobertura qualificada, submetemos dois novos casos de teste para o desenvolvedor do programa por meio de *pull request*[28]. O desenvolvedor aceitou os novos casos de teste como parte do conjunto de testes da biblioteca, o que nos forneceu mais uma evidência de que o Otimizador pode ajudar na melhoria da qualidade dos testes de programas JavaScript.

[28] https://github.com/alexandrusavin/exectimer/pull/17/files

```
1  median: function () {
2    if (this.ticks.length > 1) {
3      this.ticks.sort(function (a, b) {
4        return a && b && a.getDiff() - b.getDiff
             () || 0;
5      });
6      const l = this.ticks.length;
7      const half = Math.floor(l / 2);
8      if (l % 2) {
9        return this.ticks[half].getDiff();
10     } else {
11       return (this.ticks[half - 1].getDiff() +
             this.ticks[half].getDiff()) / 2;
12     }
13   } else {
14     return this.ticks[0].getDiff();
15   }
16 }
```

Função *median* (código original)

```
1  median: function () {
2    if (this.ticks.length > 1) {
3      const l = this.ticks.length;
4      const half = Math.floor(l / 2);
5      if (l % 2) {
6        return this.ticks[half].getDiff();
7      } else {
8        return (this.ticks[half - 1].getDiff() +
             this.ticks[half].getDiff()) / 2;
9      }
10   } else {
11     return this.ticks[0].getDiff();
12   }
13 }
```

Função *median* (código otimizado)

Figura 18 Função *median*

4.4.3 Discussão

Uma preocupação que surge dos resultados discutidos nas subseções anteriores é que quanto mais o processo de otimização encontra mudanças, menor é a qualidade percebida do conjunto de casos de teste. Às vezes, essa percepção ocorre devido à falta de cobertura para cenários específicos, mas em muitas circunstâncias observamos cobertura sem qualidade, os resultados da execução de código não sendo comparados com um gabarito pelo caso de teste ou o código sendo executado em situações que não levantam condições específicas que deveriam ser avaliadas pelo caso de teste. Assim, a cobertura de comandos pelos casos de teste não é suficiente para garantir a qualidade do código-fonte do programa submetido à otimização.

Até onde sabemos, nenhuma métrica pode replicar os resultados gerados pelo Otimizador. A contribuição desta pesquisa é a criação de uma técnica para promover a co-evolução de casos de teste e código-fonte. Os resultados do Otimizador expõem a falta de

qualidade nos casos de teste ou mostram modificações que poderiam levar a melhorias no código-fonte de programas escritos em JavaScript, no sentido de produzir um programa menor e com a mesma funcionalidade. Em nossas observações, a primeira evidência que sustenta a afirmativa acima é que os desenvolvedores prontamente aceitaram os testes que modificamos em função das alterações sugeridas pelo Otimizados e que foram submetidos aos seus projetos. Por outro lado, nenhuma métrica conhecida pode capturar efetivamente todas as características funcionais do código e comparar com as funcionalidades avaliadas pelos casos de teste. A co-evolução pode ajudar a expor essas características funcionais, sugerindo melhorias nos casos de teste por deixar claro se estas características funcionais são testadas apropriadamente.

Outra discussão que surge dos resultados é a necessidade de um ambiente de alto poder computacional para otimizar o código JavaScript. Tal necessidade pode dificultar a aplicação prática da abordagem proposta. No entanto, esse ambiente foi utilizado para reduzir o tempo de observação necessário e fornecer os recursos computacionais exigidos para otimizar vários programas grandes em paralelo, particularmente ao analisar o comportamento de algoritmos genéticos, que exigem muita memória. Um ambiente HPC como o Lobo Carneiro nos permitiu testar cenários, melhorar o Otimizador e observar os resultados repetidas vezes. Mesmo considerando o custo de compartilhar recursos em tal ambiente, o seu uso aumentou a velocidade de execução dos experimentos projetados para caracterizar o espaço de busca de programas JavaScript e para avaliar a técnica proposta nesta Tese. No entanto, uma vez determinada qual heurística e configuração produzem os resultados mais robustos, a capacidade de processamento necessária para melhorar um programa JavaScript cai para níveis compatíveis com um bom computador *desktop,* que é frequentemente mais rápido do que uma execução isolada no ambiente HPC. Por exemplo, otimizamos o *Moment* (9.978 LOC e 2.514 casos de teste) usando o DFAHC em um computador convencional (DELL OptiPlex 3040 com um CPU Core i5 e 16GB de RAM, executando Ubuntu Linux). Neste ambiente, um ciclo de otimização demorou 01:27 horas para executar, quase cinco vezes mais rápido do que as 06:51 horas de execução no Lobo Carneiro.

No entanto, esta é uma observação isolada. Se por um lado a otimização parece mais rápida em um computador pessoal, só é possível observar um processo de cada vez nesse tipo de computador. O poder computacional do Lobo Carneiro permitiu realizar 30 execuções simultâneas de cada programa, o que reduziu o tempo total necessário para executar o plano experimental de anos para meses. Como exemplo, os resultados dos últimos experimentos (19 bibliotecas e três heurísticas) contam 159 dias (ou 3.822 horas totais). Esses 159 dias foram reduzidos para 39 no ambiente do Lobo Carneiro, um quarto do tempo que seria necessário para executar os mesmos experimentos em um computador pessoal.

As heurísticas propostas nesta Tese é uma busca local onde apenas uma AST reside na memória de cada vez. Isso reduziu a quantidade de memória necessária para executar a otimização, tornando possível executar o otimizador em computadores convencionais com 8 GB ou 16 GB de RAM. O problema do consumo de memória já havia sido abordado por outros meios na literatura. Por exemplo, Le Goues et. al. (Le Goues, Dewey-Vogt, et al., 2012) propõem o uso de uma representação de *patch* para reduzir o consumo de memória para algoritmos genéticos visando à correção automatizada de *bugs*. Na representação de *patch*, um programa não é representado por sua AST durante o ciclo de otimização, mas apenas pelas linhas de código (ou nós da AST) que foram alteradas em etapas anteriores do ciclo de otimização. Esta troca de representação reduz o consumo de memória necessário para realizar a otimização, especialmente em algoritmos baseados em população. Esse tipo de representação também pode reduzir a memória consumida pela busca local proposta, particularmente conhecendo as propriedades do espaço de busca na otimização de programas JavaScript, como mostrado no Capítulo 3.

Finalmente, temos a comparação de algoritmos genéticos com a busca local. Nos experimentos descritos no Capítulo 3, observamos duas características principais que tornaram o algoritmo genético incapaz de competir com os resultados da busca local: a maioria dos resultados válidos é próximo do código original e o cruzamento tem capacidade limitada de combinar mudanças decorrentes de mutações neste espaço de busca. Para um problema em que as melhores soluções estão próximas da solução atual, os mecanismos de exploração dos algoritmos de busca local são mais eficientes do que os algoritmos genéticos. Com relação ao operador de cruzamento, este se mostrou ineficiente para encontrar alternativas válidas de código devido à distribuição das mutações a serem mescladas. Em alguns dos trabalhos relacionados (Mark Harman, Langdon, et al., 2012) (William B. Langdon & Harman, 2015) (Fogel & Atmar, 1990) (Le Goues, Dewey-Vogt, et al., 2012), os autores observaram ao menos parte deste comportamento e criaram operadores e representações específicas para o algoritmo genético.

4.4.4 Ameaças a validade

A métrica utilizada para guiar a redução de tamanho dos programas JavaScript pode ser vista como uma ameaça interna à validade do estudo experimental. A redução no tamanho dos programas foi medida como o número de caracteres após a minificação do código-fonte, durante a qual comentários são removidos, linhas em branco são eliminadas e algumas transformações estáticas de código são executadas. Pode-se argumentar que o número de caracteres é apenas um *proxy* para o tamanho do arquivo, que efetivamente influencia no tempo de transferência e carga

do mesmo, já que o JavaScript permite o uso de caracteres UTF-8 (alguns dos quais requerem mais de um *byte*). No entanto, o uso de caracteres estendidos não é comum no código-fonte e, portanto, a correlação entre o tamanho do arquivo e o número de caracteres é alta.

Com relação à validade de conclusão, observamos que embora o DFAHC supere o SFAHC para alguns programas analisados, tais resultados são próximos e às vezes superiores para SFAHC. Não é possível concluir que uma abordagem é melhor que a outra analisando os programas selecionados para o experimento apresentado neste capítulo. No entanto, o DFAHC é mais barato que o SFAHC em termos de recursos computacionais, já que ele precisa ser executado uma única vez para produzir os resultados relatados nas seções anteriores. De qualquer forma, seria necessária uma amostra maior de programas para determinar se o SFAHC é significativamente superior ao DFAHC (ou vice-versa).

Ainda com relação à validade de conclusão, é razoável questionar por que não comparamos os resultados produzidos pelo DFAHC com uma técnica que represente o estado da arte na prática dos desenvolvedores de software. Ao longo da realização desta pesquisa, descobrimos que os profissionais têm dedicado mais atenção ao JavaScript do que os acadêmicos e, portanto, ferramentas de aprimoramento de código-fonte JavaScript de última geração são encontradas na prática. Há fortes argumentos sobre qual é a melhor ferramenta para reduzir o tamanho de programas JavaScript, mas a maioria desses argumentos se concentra no minimizador *UglifyJS* e na ferramenta *Google Closure Compiler* (GCC). Bengtsson (Bengtsson, 2016) testou ambas as ferramentas em 90 programas e descobriu que os resultados do *UglifyJS* eram 0,8% menores do que os produzidos pela GCC. Por outro lado, os desenvolvedores do *Facebook* preferem o GCC por sua capacidade de eliminar códigos mortos e por seu funcionamento ser *inline* (Abramov, D., Vaughn, 2017). Comparamos o *UglifyJS* e o GCC para as instâncias usadas em nossos estudos e descobrimos que o GCC, em média, aumenta o tamanho dos programas em 20,71% (variando de -3,60% a 109,8%). Portanto, não comparamos nossa abordagem a uma ferramenta de última geração porque essa ferramenta (neste caso, *UglifyJS*) já é a base para nossas comparações, sendo utilizada para minificar os programas e suas variantes antes de compará-los.

Finalmente, no que diz respeito à validade externa, precisamos de um ser humano para avaliar as modificações encontradas pelas diversas execuções do processo de otimização, onde este ser humano avalia os resultados e aceita ou não as alterações propostas para o código-fonte. Na prática da indústria de software, é comum que a revisão de código seja uma atividade opcional, nem sempre executada pelos desenvolvedores. Esta atividade é um ponto crucial para a aceitação das modificações encontradas pelo processo de otimização. Portanto, temos uma

ameaça em relação ao processo de desenvolvimento de um projeto que não contemple atividades de revisão de código e, consequentemente, não absorva as mudanças propostas pelo Otimizador ou, possivelmente mais arriscado, aceite estas alterações sem críticas. Uma maneira de lidar com esse risco é acoplar o Otimizador ao processo de desenvolvimento do projeto, no qual um servidor de integração contínua poderia executar o Otimizador, colher os resultados e enviar um *patch* automaticamente para avaliação dos desenvolvedores, por exemplo, como um *pull request*.

4.5 Considerações finais

Depois de explorar as configurações da pesquisa local (capítulo 3), observamos o DFAHC e o SFAHC em 19 experimentos controlados. Relatamos os resultados, divididos em quantitativos e qualitativos. Fizemos uma comparação das técnicas utilizadas apresentando exemplos dos grupos de modificações encontrados pelas técnicas. Apresentamos as ameaças a validade e, por fim, concluímos recomendando o DFAHC como sendo a técnica com melhores resultados observados.

Acreditamos que a técnica pode ser incorporada em projetos industriais no processo de Integração Contínua (Elbaum, Rothermel, & Penix, 2014). Como trabalhos futuros, pretendemos explorar outras características não-funcionais, além do tamanho do programa, como consumo de memória, consumo de energia entre outros. Com isso, outros operadores de otimização terão que ser estudados. Além disso, pretendemos executar experimentos em projetos industriais para realizar otimização em seu processo de integração, um desafio de engenharia para fazer a técnica apresentada aqui escalonar para bases de código maiores e permitir sua conexão com outras ferramentas, como enviar solicitações de *pull requests* com *patches* para que um engenheiro humano avalie diretamente modificações.

5 Conclusão e trabalhos futuros

Neste capítulo, apresentamos as conclusões deste trabalho de pesquisa, enfatizando as suas principais contribuições. Adicionalmente, apresentamos algumas limitações e questões abertas não abordadas nesta tese. Finalmente, o caminho a seguir é delineado para mostrar possíveis trabalhos futuros como resultado das realizações atuais.

5.1 Considerações finais

Esta Tese teve como objetivo aplicar técnicas heurísticas de melhoramento de código-fonte sobre programas escritos na linguagem JavaScript para encontrar alterações no código que reduzam o tamanho (em número de caracteres) destes programas. A redução do tamanho dos programas JavaScript tem como objetivo reduzir o tempo de transferência destes programas entre o cliente e o servidor de uma aplicação Web (ou móvel), assim como o tempo de carga e interpretação destes programas no cliente. Indiretamente, a redução do tamanho afeta também o tempo de execução dos programas: quanto menos instruções um código possuir, mais rápido será a interpretação e o processamento do mesmo.

A motivação para diminuir o tamanho de código JavaScript vem em decorrência da intensificação do uso de JavaScript nas últimas décadas, em que seu uso alcançou escala global e se projetor em vários contextos diferentes de uso, conforme discutido no Capítulo 1. Se considerarmos que hoje JavaScript é a base para construção de sites e aplicativos móveis, além de permitir incluir comportamento no lado servidor, uma pequena melhoria em uma biblioteca que tenha uso em escala mundial pode trazer um ganho considerável em tempo de processamento e até mesmo em consumo de energia.

O problema de reduzir o tamanho do código-fonte nesse cenário não é novo e já existem técnicas de redução específicas para JavaScript, como a minificação [10, 11]. A técnica proposta nessa Tese, porém, complementa a minificação além de ter sido projetada para executar sob o processo de integração contínua de um projeto de software, onde a cada nova alteração a otimização seria executada para propor um conjunto de modificações automaticamente.

5.2 Contribuições

Entre as contribuições da Tese, destacam-se:

- Uma técnica heurística de busca para reduzir o tamanho de código-fonte escrito em JavaScript mantendo a sua funcionalidade;

- Um ferramental capaz de executar dois tipos de busca heurística (algoritmos genéticos e busca local) para a otimização de código JavaScript. O próprio ferramental foi desenvolvido em JavaScript e consiste de 16.032 linhas de código [29];

- Um estudo experimental para a caracterização do espaço de busca de soluções para a redução de tamanho de programas escritos em JavaScript, consistindo de um estudo exploratório usando um algoritmo genético e um algoritmo de busca local projetados para trazer conhecimento sobre o espaço de soluções no problema de interesse;

- A coorientação de um Trabalho de Conclusão de Curso de Graduação na Universidade Federal do Estado do Rio de Janeiro (UNIRIO), intitulado *Estudo em Larga Escala sobre a Estrutura do Código-fonte de Pacotes JavaScript* (Silva, D., Sobral, 2017), que teve como objetivo caracterizar programas JavaScript em relação ao seu tamanho, complexidade e ao uso dos diferentes tipos de nós que compõem a especificação da linguagem de programação;

- Um estudo experimental para a observação dos resultados da aplicação da busca local proposta em 19 bibliotecas em JavaScript. Esse estudo, executado durante 159 dias (3822 horas) e com 190 rodadas, traz a discussão, em detalhes, dos resultados encontrados agrupando-os as modificações encontradas, discutindo como a técnica foi capaz de produzi-las, classificando-as quanto a sua corretude e discutindo as ameaças a validade desses resultados.

- O incremento nos testes de duas bibliotecas *open source* (citadas no capítulo 4, seção 4.4.2), para as quais enviamos adições ao código de testes dessas bibliotecas e os mesmos foram incorporados;

5.3 Respostas para as questões de pesquisa

As duas primeiras questões de pesquisa apresentadas no capítulo 1 foram respondidas no capítulo 3 e a última questão de pesquisa foi respondida no capítulo 4. A seguir, apresentamos um resumo das conclusões a que chegamos.

- RQ1: É possível reduzir o tamanho de um código-fonte escrito em JavaScript usando técnicas heurísticas de busca?

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

Conforme visto no capítulo 3 (seção 3.3.2), os dois tipos de busca heurística observados nessa pesquisa encontraram versões reduzidas dos códigos submetidos à otimização. As mudanças encontradas foram catalogadas em grupos e discutidas no capítulo 4. Alguns grupos de mudanças precisam da avaliação de um ser humano envolvido com o projeto para determinar sua corretude; outros grupos de mudanças foram consideradas não corretas e o motivo pelo qual foram encontradas está associado à qualidade limitada dos casos de teste desenvolvidos para os programas em questão. Assim, concluímos que é parcialmente possível utilizar heurísticas para reduzir o tamanho do código-fonte de programas JavaScript.

- RQ2: Quais são as características do espaço de busca por variantes de programas JavaScript onde foram aplicadas operações visando reduções de tamanho?

  No capítulo 3 (seção 3.3.3) foram apresentadas as principais características do espaço de busca do problema de redução de código-fonte em JavaScript. Em resumo, a principal característica observada é que existe um grande número de pequenos *patches* agrupados e distribuídos ao longo do código-fonte das bibliotecas JavaScript. Além disso, foi observado que quanto maior o programa, menores serão os seus *patches* e eles estarão mais próximos um do outro. Adicionalmente, um estudo sobre os tipos de instruções nos *patches* mais frequentes foi conduzido (Capítulo 3, seção 3.3.4) com o objetivo de reduzir o custo de execução da busca por programas que tenham a mesma funcionalidade e tamanho reduzido. Propusemos uma redução no espaço de busca baseado nessa frequência e os resultados estão descritos no capítulo 4.

- RQ3: Que tipos de alterações são realizadas por um procedimento de busca heurística para reduzir o tamanho de programas JavaScript?

  Os tipos de alteração encontrados pela técnica de busca heurística proposta estão descritos no capítulo 4 (seção 4.4) e podem ser classificadas através das principais mudanças em dez grupos pelo tipo de instrução removida pela otimização, à saber: u*se strict*, funções não utilizadas, parâmetros de função, chamadas de função, valores default, propriedades em objetos, dependências, retorno, condicionais e instruções não cobertas. Essas mudanças foram avaliadas de maneira quantitativa e qualitativa, visando entender como a otimização produziu esses resultados e o quão expressivo eles podem ser. Por fim, foi apresentada uma discussão sobre quais desses resultados podem ser incorporados ao código-fonte dos programas originais e qual a relação dos mesmos com a qualidade da suíte de testes dos programas.

5.4 Limitações

A técnica proposta foi observada em um conjunto restrito de bibliotecas, sendo todas *open source* e disponíveis no repositório NPM. Boa parte destas bibliotecas possui documentação técnica incompleta. Já a documentação funcional ou seus requisitos praticamente não existem nos repositórios, podendo ser inferidos apenas por exemplos ou por meio dos seus casos de teste. Isso limitou nosso espectro de conclusões e as observações que puderam ser feitas frente às modificações propostas pela técnica nos experimentos.

Não foi possível obter respostas de todos os desenvolvedores das bibliotecas analisadas sobre as mudanças propostas pelo otimizador. Em alguns casos, como nos *pull requests* enviados com mudanças nos casos de teste, houve retorno dos desenvolvedores e pudemos coletar evidências positivas sobre o uso do otimizador. Porém, em um projeto industrial ou mesmo em um projeto controlado poderíamos obter *feedbacks* sobre as modificações dos próprios desenvolvedores e aumentar a confiança nos resultados discutidos no Capítulo 4.

O operador utilizado na técnica de busca visa exclusivamente a exclusão de instruções. Portanto, o tipo e o número de mudanças que ele pode propor são limitados conforme discutido no Capítulo 4. Em trabalhos relacionados e no campo de melhoramento genético, existe uma maior diversidade de operadores. Tais operadores, se observados sob o mesmo critério não funcional de redução do tamanho do código, poderiam trazer resultados positivos. Isso, além de um trabalho de pesquisa futuro pode ser considerado uma limitação.

5.5 Questões em aberto para trabalhos futuros

No contexto dessa pesquisa, algumas questões permanecem em aberto e podem se tornar tema de futuras pesquisas. Algumas dependem apenas de mais estudos para a sua avaliação; outras surgiram como consequência da evolução da própria pesquisa. A seguir, listamos algumas destas questões em aberto.

É possível otimizar outras características não-funcionais? O foco principal dessa Tese esteve no tamanho do código. Reduzir o tamanho do código mostrou-se possível. Porém, seria também possível explorar outras características não funcionais como consumo de memória ou consumo de energia? O mesmo operador utilizado nessa pesquisa seria suficiente e traria resultados positivos ou seria necessário estudar outros operadores de otimização?

É possível integrar o ferramental a um processo de integração contínua de um projeto industrial? Um desafio de engenharia seria fazer a técnica apresentada aqui escalonar para bases de código maiores e permitir sua conexão com outras ferramentas, como enviar solicitações de *pull requests* com *patches* para que um engenheiro humano avalie diretamente essas modificações. Um exemplo de um processo de integração contínua com o Otimizador inserido pode ser visto na Figura 19, onde a construção de um *release* de um projeto software é representando. O *release* é iniciado por um evento (*commit* de um desenvolvedor, agendamento) onde servidor de integração contínua inicia a execução dos passos de construção configurados (executar a compilação da última versão, executar a instalação automática em um ambiente pré-configurado, executar os testes de unidade, entre outras tarefas). Após a construção completa o Otimizador seria notificado de que uma nova versão pode ser submetida ao processo de redução. O Otimizador então procuraria patches de redução nesse código e submeteria as mudanças ao engenheiro de software responsável pelos releases do projeto. Cabe ao engenheiro acatar ou não as mudanças, podendo absorvê-las apenas em parte. Após a decisão do mesmo, o processo terminaria disponibilizando o novo release otimizado (ou não).

Como avaliar a qualidade dos testes? Um dos resultados aponta para baixa qualidade percebida dos testes das bibliotecas observadas. Devido a isso, alguma das alterações encontradas foram classificadas como inválidas. Seria possível explorar uma co-evolução do código dos testes e o código do software em si? Um estudo para identificar modificações inválidas e quais testes precisam ser melhorados poderia ser um trabalho futuro. Além disso, quais podem ser as métricas de teste para avaliar a qualidade dos mesmos antecipadamente? Isso poderia evitar que algumas modificações fossem encontradas incorretamente.

É possível avaliar o espaço de busca removido? Uma das decisões durante a pesquisa foi a redução do espaço de busca, removendo alguns tipos de instrução que tinham pouco efeito sobre os resultados da otimização. Seria possível medir a perda ao não explorar esse espaço e comparar com os resultados atuais? Um estudo para fazer essa comparação e propor algum outro tipo de redução ou mesmo de técnica que possa explorar melhor esse espaço pode vir a ser conduzidos como trabalho futuro.

É possível aumentar a cobertura dos testes automaticamente? A geração de casos de testes é comum no contexto da Engenharia de Software Baseada em Busca. Porém, este tipo de técnica foi explorado em outras linguagens de programação, mas não em JavaScript. Um trabalho futuro seria explorar uma técnica de geração de casos de testes para aumentar a confiança nos testes e, com isso, avaliar o impacto na qualidade da solução encontrada pela técnica proposta nessa tese.

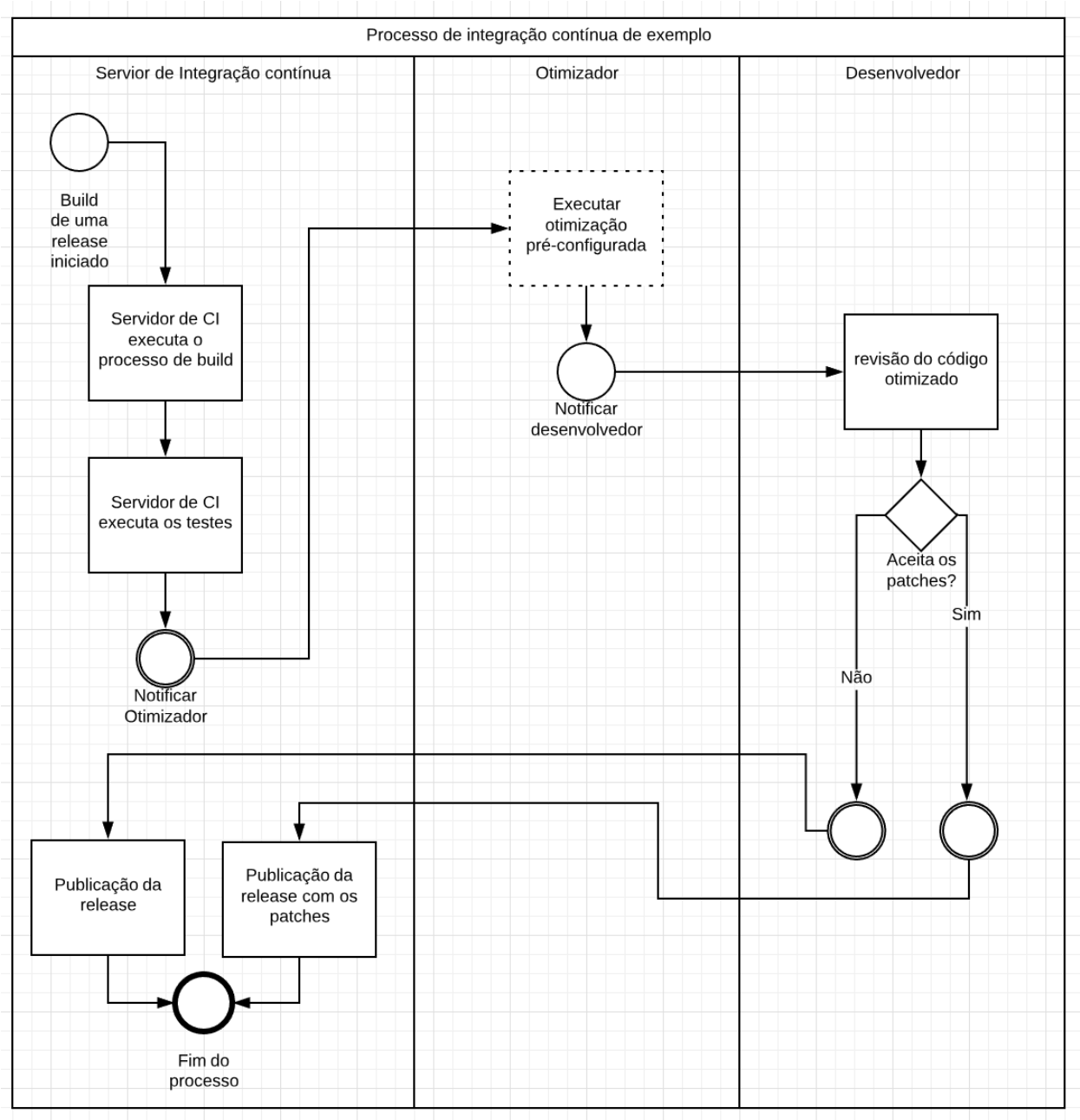

Figura 19 Processo de integração contínua de exemplo.

Referências Bibliográficas

Abramov, D., Vaughn, B. (2017). Behind the Scenes: Improving the Repository Infrastructure.

Arcuri, A. (2008). On the Automation of Fixing Software Bugs. *Icse'08*. https://doi.org/10.1145/1370175.1370223

Arcuri, A., & Yao, X. (2008). A novel co-evolutionary approach to automatic software bug fixing. In *2008 IEEE Congress on Evolutionary Computation, CEC 2008*. https://doi.org/10.1109/CEC.2008.4630793

Bäck, T., & Schwefel, H.-P. (1993). An Overview of Evolutionary Algorithms for Parameter Optimization. *Evolutionary Computation*. https://doi.org/10.1162/evco.1993.1.1.1

Bengtsson, P. (2016). Advanced Closure Compiler vs UglifyJS2.

Brown, W. J., Malveau, R. C., Mowbray, T. J., & Wiley, J. (1998). AntiPatterns: Refactoring Software , Architectures, and Projects in Crisis. *Crisis*.

Brownlee, J. (2011). *Clever Algorithms: Nature Inspired Programming Recipes*. *Search*.

Bruce, B. R., Petke, J., & Harman, M. (2015). Reducing Energy Consumption Using Genetic Improvement. In *GECCO '15: Proceedings of the 2015 on Genetic and Evolutionary Computation Conference*. https://doi.org/doi:10.1145/2739480.2754752

Bruce, B. R., Petke, J., Harman, M., & Barr, E. T. (2018). Approximate Oracles and Synergy in Software Energy Search Spaces. *IEEE Transactions on Software Engineering*. https://doi.org/10.1109/TSE.2018.2827066

Burtscher, M., & Livshits, B. (2010). JSZap: compressing JavaScript code. In *Proceedings of the USENIX Conference on Web Application Development*.

Carson, E. D. (2000). On foundations and outcome evaluation. *Nonprofit and Voluntary Sector Quarterly*. https://doi.org/10.1177/0899764000293008

Cody-Kenny, B., & Barrett, S. (2013). The emergence of useful bias in self-focusing genetic programming for software optimisation. In *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)*. https://doi.org/10.1007/978-3-642-39742-4_29

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

Cohen, J. (1992). A power primer. *Psychological Bulletin*. https://doi.org/10.1037/0033-2909.112.1.155

Elbaum, S., Rothermel, G., & Penix, J. (2014). Techniques for improving regression testing in continuous integration development environments. In *Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering - FSE 2014*. https://doi.org/10.1145/2635868.2635910

Fard, A. M., & Mesbah, A. (2013). JSNOSE: Detecting javascript code smells. In *IEEE 13th International Working Conference on Source Code Analysis and Manipulation, SCAM 2013*. https://doi.org/10.1109/SCAM.2013.6648192

Fogel, D. B., & Atmar, J. W. (1990). Comparing genetic operators with gaussian mutations in simulated evolutionary processes using linear systems. *Biological Cybernetics*. https://doi.org/10.1007/BF00203032

Forrest, S., & Mitchell, M. (1993). Relative building-block fitness and the building-block

hypothesis. *Ann Arbor*. https://doi.org/10.1.1.47.4739

Forrest, S., Nguyen, T., Weimer, W., & Le Goues, C. (2009). A genetic programming approach to automated software repair. In *GECCO 2009: Proceedings of the 11th Annual conference on Genetic and evolutionary computation*. https://doi.org/doi:10.1145/1569901.1570031

Ghannem, A., El Boussaidi, G., & Kessentini, M. (2013). Model refactoring using interactive genetic algorithm. In *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)*. https://doi.org/10.1007/978-3-642-39742-4_9

Glinz, M. (2007). On Non-Functional Requirements. *15th IEEE International Requirements Engineering Conference (RE 2007)*. https://doi.org/10.1109/RE.2007.45

Goldberg, D. E. (1989). *Genetic Algorithms in Search, Optimization, and Machine Learning*. *Addison Wesley*. https://doi.org/10.1007/s10589-009-9261-6

Haraldsson, S. O., Woodward, J. R., Brownlee, A. E. I., Smith, A. V., & Gudnason, V. (2017). Genetic Improvement of Runtime and its Fitness Landscape in a Bioinformatics Application. *GECCO 2017 - Proceedings of the Genetic and Evolutionary Computation Conference Companion*. https://doi.org/10.1145/3067695.3082526

Harman, M. (2007). The current state and future of search based software engineering. In *FoSE 2007: Future of Software Engineering*. https://doi.org/10.1109/FOSE.2007.29

Harman, M., Jia, Y., & Langdon, W. B. (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)*. https://doi.org/10.1007/978-3-319-09940-8_19

Harman, M., & Jones, B. F. (2001). Search-based software engineering. *Information and Software Technology*. https://doi.org/10.1016/S0950-5849(01)00189-6

Harman, M., & Langdon, W. B. (2014). Genetically improved CUDA C++ software. *Genetic Programming*. https://doi.org/10.1007/978-3-662-44303-3

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

Harman, M., Mansouri, S. A., & Zhang, Y. (2012). Search-based Software Engineering: Trends, Techniques and Applications. *ACM Computing Surveys*. https://doi.org/10.1145/2379776.2379787

Jensen, S. H., 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)*. https://doi.org/10.1007/978-3-642-03237-0_17

JSON.org. (2014). Introducing JSON. https://doi.org/10.1017/CBO9781107415324.004

Kautz, H., & Selman, B. (2007). The state of SAT. *Discrete Applied Mathematics*. https://doi.org/10.1016/j.dam.2006.10.004

Koza, J. R. (1994). Genetic programming as a means for programming computers by natural selection. *Statistics and Computing*. https://doi.org/10.1007/BF00175355

Langdon, W. B. (2014). Genetic Improvement of Programs. In *16th International Symposium on Symbolic and Numeric Algorithms for Scientific Computing (SYNASC 2014)*. https://doi.org/doi:10.1109/SYNASC.2014.10

Langdon, W. B. (2015). Genetic improvement of software for multiple objectives. In *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)*. https://doi.org/10.1007/978-3-319-22183-0_2

Langdon, W. B. (2015). Performance of genetic programming optimised Bowtie2 on genome comparison and analytic testing (GCAT) benchmarks. *BioData Mining*. https://doi.org/10.1186/s13040-014-0034-0

Langdon, W. B., & Harman, M. (2015). Grow and Graft a Better CUDA pknotsRG for RNA Pseudoknot Free Energy Calculation. *Proceedings of the 17th Annual Conference on Genetic and Evolutionary Computation - GECCO '15*. https://doi.org/10.1145/2739482.2768418

Langdon, W. B., & Harman, M. (2015). Optimizing existing software with genetic programming. *IEEE Transactions on Evolutionary Computation*. https://doi.org/10.1109/TEVC.2013.2281544

Langdon, W. B., Modat, M., Petke, J., & Harman, M. (2014). Improving {3D} Medical Image Registration {CUDA} Software with Genetic Programming. *GECCO '14: Proceeding of the Sixteenth Annual Conference on Genetic and Evolutionary Computation Conference*. https://doi.org/10.1145/2576768.2598244

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*. https://doi.org/10.1109/ICSE.2012.6227211

Le Goues, C., Nguyen, T. V., Forrest, S., & Weimer, W. (2012). GenProg: A generic method for automatic software repair. *IEEE Transactions on Software Engineering*. https://doi.org/10.1109/TSE.2011.104

Mesbah, A., & Van Deursen, A. (2007). Migrating multi-page web applications to single-page AJAX interfaces. In *Proceedings of the European Conference on Software Maintenance and Reengineering, CSMR*. https://doi.org/10.1109/CSMR.2007.33

Myers, E. W. (1986). An O(ND) difference algorithm and its variations. *Algorithmica*. https://doi.org/10.1007/BF01840446

Ouni, A., Kessentini, M., Bechikh, S., & Sahraoui, H. (2015). Prioritizing code-smells correction tasks using chemical reaction optimization. *Software Quality Journal*. https://doi.org/10.1007/s11219-014-9233-7

Penta, M. Di. (2005). Evolution doctor: A framework to control software system evolution. In *Proceedings of the European Conference on Software Maintenance and Reengineering, CSMR*. https://doi.org/10.1109/CSMR.2005.29

Petke, J., Haraldsson, S. O., Harman, M., Langdon, W. B., White, D. R., & Woodward, J. R. (2018). Genetic Improvement of Software: A Comprehensive Survey. *IEEE Transactions on Evolutionary Computation*. https://doi.org/10.1109/TEVC.2017.2693219

Petke, J., Langdon, W. B., & Harman, M. (2013). Applying Genetic Improvement to {MiniSAT}. In *Symposium on Search-Based Software Engineering*. https://doi.org/doi:10.1007/978-3-642-39742-4_21

Richards, G., Lebresne, S., Burg, B., & Vitek, J. (2010). An analysis of the dynamic behavior of JavaScript programs. In *Proceedings of the 2010 ACM SIGPLAN conference on Programming language design and implementation - PLDI '10*. https://doi.org/10.1145/1806596.1806598

Rosca, J. (1997). *Hierarchical Learning with Procedural Abstraction Mechanisms*. *Ph.D thesis*.

Ryan, C., & Ivan, L. (1999). Automatic Parallelization of Arbitrary Programs. In *Genetic Programming, Proceedings of EuroGP'99*. https://doi.org/10.1007/3-540-48885-5_21

Ryan, C., & Walsh, P. (1995). Automatic conversion of programs from serial to parallel using Genetic Programming - The Paragen System. *Proceedings of ParCo'95*.

Ryan, C., & Walsh, P. (1997). The Evolution of Provable Parallel Programs. In *Genetic Programming 1997: Proceedings of the Second Annual Conference*.

Schulte, E., Weimer, W., & Forrest, S. (2015). Repairing COTS Router Firmware without Access to Source Code or Test Suites: A Case Study in Evolutionary Software Repair. *Proceedings of the Companion Publication of the 2015 on Genetic and Evolutionary Computation Conference - GECCO Companion '15*. https://doi.org/10.1145/1235

Silva, D., Sobral, L. (2017). *Um Estudo em Larga Escala sobre a Estrutura do Código-fonte de Pacotes JavaScript*. Unirio. Rio de Janeiro.

Smith-Miles, K., & Lopes, L. (2012). Measuring instance difficulty for combinatorial optimization problems. *Computers and Operations Research*. https://doi.org/10.1016/j.cor.2011.07.006

Triantafyllis, S., Vachharajani, M., Vachharajani, N., & August, D. I. (2003). Compiler optimization-space exploration. In *International Symposium on Code Generation and Optimization, CGO 2003*. https://doi.org/10.1109/CGO.2003.1191546

Ubl, M., & Eiji, K. (2010). Introducing WebSockets: Bringing Sockets to the Web - HTML5 Rocks. *HTML5rocks*.

Walsh, P., & Ryan, C. (1996). Paragen: A Novel Technique for the Autoparallelisation of Sequential Programs using Genetic Programming. In *Genetic Programming 1996: Proceedings of the First Annual Conference*.

Wasserman, A. I. (2010). Software engineering issues for mobile application development. *Proceedings of the FSE/SDP Workshop on Future of Software Engineering Research. ACM, 2010*. https://doi.org/10.1145/1882362.1882443

White, D. R., Arcuri, A., & Clark, J. A. (2011). Evolutionary improvement of programs. *IEEE Transactions on Evolutionary Computation*. https://doi.org/10.1109/TEVC.2010.2083669

Wilkerson, J. L., Tauritz, D. R., & Bridges, J. M. (2012). Multi-objective coevolutionary automated software correction. In *Proceedings of the fourteenth international conference on Genetic and evolutionary computation conference - GECCO '12*. https://doi.org/10.1145/2330163.2330333

Williams, K. P., & Williams, S. A. (1996). Genetic compilers: {A} new technique for automatic parallelisation. *2nd European School of Parallel Programming Environments (ESPPE'96)*. https://doi.org/10.1.1.49.3499

Wilson, G. C., McIntyre, A., & Heywood, M. I. (2004). Resource review: Three open source systems for evolving programs - Lilgp, ECJ and grammatical evolution. *Genetic Programming and Evolvable Machines*. https://doi.org/10.1023/B:GENP.0000017053.10351.dc

Apêndice A. Como utilizar o Otimizador para reprodução dos experimentos

Para fazer o setup do Otimizador, com o objetivo de executar uma otimização em uma biblioteca JavaScript ou mesmo de reproduzir os experimentos descritos nessa tese, alguns passos prévios e pré-requisitos são necessários, à saber:

1. Instalar o NodeJs;
2. Baixar o código fonte;
3. Realizar a instalação das dependências;
4. Realizar a instalação do projeto alvo do Otimizador;
5. Configurar o Otimizador para esse projeto;
6. Executar o processo;

A.1 Instalar o NodeJS

O Otimizador foi construído na linguaguem TypeScript e, para tal, precisa da infraestrutura do NodeJS para sua execução. Acesse a URL[30] de download do NodeJS e baixe a sua versão mais atual, para a plataforma e sistema operacional desejados. Após a instalação do mesmo no computador, execute o passo A2.

A.2 Baixar o código fonte

Para baixar o código fonte do Otimizador, acesse a URL[31] do repositório no Github e faça o download da última versão disponível. Descompacte o arquivo baixado e, em seguida, execute o passo A3.

A.3 Realizar a instalação das dependências

Na pasta raiz do código fonte (JavaScriptHeuristicOptmizer) abra um terminal de comando e execute o seguinte comando: “npm install”. Esse comando foi instalado e registrado no passo A1 durante a instalação do NodeJS. Ele realizará a instalação de todas as dependências

[30] https://nodejs.org/en/download/

[31] https://github.com/ffarzat/JavaScriptHeuristicOptmizer/releases

necessárias para a execução do Otimizador localmente. Após a instalação das dependências, execute o passo A4.

A.4 Realizar a instalação do projeto alvo do Otimizador

Nesse passo, o projeto que será otimizador tem que ser integralmente instalado e configurado para executar no mesmo computador que o Otimizador. Para realizar a otimização de um código fonte alvo, é preciso que o ambiente de desenvolvimento do mesmo esteja disponível e com acesso ao Otimizador. Escolha uma biblioteca de sua escolha. Para efeito de exemplo, vamos utilizar uma biblioteca disponível no site do NPM[32], chamada D3-node.

Ao acessar o NPM, existe o link[33] para o repositório de código fonte da biblioteca. Faça o download da última versão da biblioteca. Descompacte o arquivo baixado e, em seguida, execute o comando "npm install" no terminal para essa biblioteca. Conforme citado no passo A3, esse comando instala as dependências necessárias para a execução da biblioteca.

Após a instalação com sucesso, execute o comando "npm test". Esse comando vai executar os testes configurados para essa biblioteca (testes de unidade, integração, aceitação ou qualquer outro configurado). Observe o tempo necessário para execução dos testes. Ele será necessário para a configuração do Otimizador.

Abra o arquivo de nome "package.json" no diretório raiz do projeto alvo da otimização. Procure pela propriedade de nome "main" e observe o seu valor. Essa propriedade aponta para o arquivo inicial da biblioteca, ou seja, o seu ponto de entrada. No caso da D3-node o arquivo é o index.js, também encontrado no diretório raiz. Com o tempo necessário para executar os testes medido e com o nome do arquivo de entrada da biblioteca, temos as informações necessárias para configurar o Otimizador.

A.5 Configurar o Otimizador para esse projeto

Volte ao diretório do Otimizador e localize, na pasta raiz, o arquivo de nome "Configuration.json". Esse é o arquivo de configuração do Otimizador. Nele faremos modificações nos valores das seguintes configurações:

- trials: total de execuções desejadas do otimizador na biblioteca alvo;
- logFilePath: caminho para o arquivo de log que será utilizado durante o processo;

[32] https://www.npmjs.com/package/d3-node

[33] https://github.com/d3-node/d3-node

- tmpDirectory: caminho para o diretório temporário necessário para executar a otimização;
- resultsDirectory: caminho para o diretório onde os resultados serão salvos ao final da Otimização;
- heuristics: heurísticas que podem ser utilizadas pelo Otimizador. Os valores possíveis são:
  - RD: *ramdon seach*;
  - GA: *genetic algorithm*;
  - HC: *hill climbing*;
- clientTimeout: coloque aqui o tempo observado de execução dos testes da biblioteca, em segundos, somando ao valor. Exemplo: se a D3-node, na máquina onde a configuração estiver sendo executada, levou 3 segundos para executar os testes, o valor de configuração deverá ser 13 segundos;
- clientsTotal: quantidade de processadores disponíveis para a Otimização. Atenção: sempre deixe ao menos um processador disponível para o sistema operacional;
- memory: quantidade de memória disponível para Otimização consumir durante a execução. Ao chegar nesse limite, o otimizador é derrubado automaticamente pelo NodeJS. Essa configuração é em MB;
- libraries: essa configuração representa uma biblioteca. É necessário configurar o nome, o caminho até o diretório raiz da biblioteca alvo e seu arquivo inicial;

A.6 Executar o processo

Com o ambiente da biblioteca e do Otimizador configurados, execute o comando “npm start” no diretório raiz do Otimizador. Esse comando fará o *build*, executará os testes de unidade do Otimizador e iniciará o processo de otimização. Esse processo pode levar de minutos até horas. Ao final do processo, o Otimizador vai salvar um arquivo chamado “Results.csv” dentro do diretório configurado no passo A5. Além disso, para cada rodada configurada, o Otimizador vai salvar um arquivo com o número da rodada e com extensão “.js”. Por exemplo, “0.js”. Esse arquivo é o código otimizado da biblioteca que foi configurada.