Giovanni Francesco Guarnieri

SiMut: Um Framework Automatizado para Apoiar a Redução do Custo do Teste de Mutação com Base em Similaridade de Programas

Sorocaba, SP

30 de Março de 2022

Giovanni Francesco Guarnieri

SiMut: Um Framework Automatizado para Apoiar a Redução do Custo do Teste de Mutação com Base em Similaridade de Programas

Dissertação de mestrado apresentada ao Programa de Pós-Graduação em Ciência da Computação (PPGCC-So) da Universidade Federal de São Carlos como parte dos requisitos exigidos para a obtenção do título de Mestre em Ciência da Computação. Linha de pesquisa: Sistemas Computacionais.

Universidade Federal de São Carlos – UFSCar

Centro de Ciências em Gestão e Tecnologia – CCGT

Programa de Pós-Graduação em Ciência da Computação – PPGCC-So

Orientador: Prof. Dr. Fabiano Cutigi Ferrari

Sorocaba, SP

30 de Março de 2022

Guarnieri, Giovanni Francesco

SiMut: um framework automatizado para apoiar a redução do custo do teste de mutação com base em similaridade de programas / Giovanni Francesco Guarnieri -- 2022.
103f.

Dissertação (Mestrado) - Universidade Federal de São Carlos, campus Sorocaba, Sorocaba
Orientador (a): Fabiano Cutigi Ferrari
Banca Examinadora: Fabiano Cutigi Ferrari, Márcio de Medeiros Ribeiro, Auri Marcelo Rizzo Vincenzi
Bibliografia

1. Teste de software. 2. Teste de mutação. 3. Mutação seletiva. I. Guarnieri, Giovanni Francesco. II. Título.

Ficha catalográfica desenvolvida pela Secretaria Geral de Informática (SIn)

DADOS FORNECIDOS PELO AUTOR

Bibliotecário responsável: Maria Aparecida de Lourdes Mariano - CRB/8 6979

UNIVERSIDADE FEDERAL DE SÃO CARLOS

Centro de Ciências em Gestão e Tecnologia
Programa de Pós-Graduação em Ciência da Computação

Folha de Aprovação

Defesa de Dissertação de Mestrado do candidato Giovanni Francesco Guarnieri, realizada em 30/03/2022.

Comissão Julgadora:

Prof. Dr. Fabiano Cutigi Ferrari (UFSCar)

Prof. Dr. Márcio de Medeiros Ribeiro (UFAL)

Prof. Dr. Auri Marcelo Rizzo Vincenzi (UFSCar)

O Relatório de Defesa assinado pelos membros da Comissão Julgadora encontra-se arquivado junto ao Programa de Pós-Graduação em Ciência da Computação.

Dedico este trabalho à minha mãe Zilda que sempre me incentivou a estudar e a realizar os meus sonhos.

À memória do meu pai João por sempre ter me apoiado nas minhas decisões.

À minha esposa Flávia pelo amor, carinho e incentivo.

Às minhas irmãs Yacamara e Yaramarta por sempre estarem presentes nos momentos tristes e alegres.

Agradecimentos

Ao meu orientador Prof. Dr. Fabiano Cutigi Ferrari pela paciência, confiança e pelos conhecimentos compartilhados.

Ao meu colega de pesquisa Alessandro Viola Pizzoleto, por sempre acreditar em mim e no projeto.

Aos professores que colaboraram com a minha formação.

À Universidade Federal de São Carlos pelo apoio cedido ao longo do mestrado.

À minha família, que esteve sempre presente me apoiando.

À minha esposa Flávia, por sempre acreditar em mim e nas minhas capacidades.

Por fim, a todos aqueles de alguma forma me ajudaram durante essa jornada.

Muito obrigado.

O presente trabalho foi realizado com apoio da Coordenação de Aperfeiçoamento de Pessoal de Nível Superior - Brasil (CAPES) - Código de Financiamento 001.

"A persistência é o menor caminho do êxito."
(Charles Chaplin)

Resumo

Contexto: O teste de software é fundamental para garantir que o programa realmente atenda corretamente às necessidades do usuário. Cientes da importância desta etapa, pesquisadores propuseram diversas formas para identificar defeitos em programas. Entre elas, o critério Análise de Mutantes, ou teste de mutação, é comprovadamente efetivo mas, por demandar um alto custo de aplicação, é pouco eficiente e ainda não comumente adotado na indústria de software. Sabendo do potencial desse critério, pesquisadores buscaram e apresentaram formas de reduzir o custo do teste de mutação para torná-lo mais viável. Entretanto, como constatado por diversos pesquisadores, resultados das técnicas de redução de custo do teste de mutação aplicadas em um programa são válidos somente para os programas que foram alvos dos experimentos realizados. Recentemente, alguns pesquisadores, em seus estudos, se apoiaram no uso da similaridade entre programas como forma de reaproveitar experiências adquiridas em programas já testados com teste de mutação. Entretanto, essa é uma abordagem que ainda carece de experimentos utilizando variadas formas de similaridade. **Objetivo:** Neste trabalho apresentam-se uma implementação e uma avaliação de um *framework* denominado *SiMut*. Este *framework* foi introduzido em um estudo anterior com o objetivo de ajudar a reduzir o custo para testar um programa baseado em um grupo de programas similares previamente testados com mutação. **Metodologia:** A implementação apresentada neste trabalho lida com programas escritos na linguagem Java e inclui um conjunto de variantes que se relacionam a três tipos de abstrações de programas (código-fonte original, código-fonte ofuscado, e métricas de complexidade interna), três estratégias de cálculos de similaridade (*clustering*, funções de distância entre *strings*, e plagiarismo) e uma abordagem de redução de custo de mutação (inspirada na Mutação Seletiva). A avaliação apresentada, utilizando 35 pequenos programas escritos na linguagem Java, abrange 20 configurações variando-se as técnicas de abstrações de programas e similaridades. **Resultados:** Uma comparação cruzada envolvendo os *clusters* formados e uma comparação com *clusters* formados aleatoriamente aponta para as configurações que tendem a atingir alta efetividade em prever os melhores operadores de mutação para programas não testados com o teste de mutação. **Conclusões:** Considerando-se as configurações selecionadas para os experimentos, os resultados apresentaram diversas combinações que são efetivas para prever os melhores operadores de mutação para programas não testados, onde destacaram-se as combinações que envolveram funções de distância entre *strings*. Em relação às abstrações de programas utilizadas para calcular a similaridade, o código-fonte original parece ser tão relevante quanto o código-fonte ofuscado e as métricas internas de complexidade.

Palavras-chaves: Teste de Software. Teste de Mutação. Técnicas de Redução de Custo. Similaridade de programas. Mutação Seletiva. Agrupamento.

Abstract

Context: Software testing is essential to ensure that the program actually correctly meets the user's needs. Aware of the importance of this step, researchers have proposed several ways to identify program faults. Among the fault identification techniques, there is the mutation analysis criterion, or mutation testing, which has proven to be effective for fault identification, but, as it demands a high application cost, it is little efficient and hence avoided in the software industry. Given the potential of this criterion, researchers sought and presented ways to reduce the cost of mutation testing to make it more viable. However, as verified by several researchers, the results produced by cost reduction techniques for mutation testing applied in a program are valid only for the programs that were targeted in the performed experiments. Recently, some researchers, in their studies, have relied on the use of similarity between programs as a way to reuse experiences acquired in programs already tested with mutation testing. However, this is an approach that still lacks experiments using various forms of similarity. **Objective:** This work presents an implementation and evaluation of a framework called $SiMut$. This framework was introduced in a previous study with the aim of helping to reduce the cost of testing a program based on a group of similar programs previously tested with mutation. **Methodology:** The implementation presented in this work deals with programs written in the Java language and includes a set of variants that relate to three types of program abstractions (original source code, obfuscated source code, and internal complexity metrics), three similarity calculation strategies (clustering, distance functions between strings, and plagiarism) and a mutation cost reduction approach (inspired by Selective Mutation). The presented evaluation, using 35 small programs written in the Java language, covers 20 configurations varying the techniques of abstractions and similarities. **Results:** A cross-comparison involving the formed clusters and a comparison with randomly formed clusters points to configurations that tend to achieve high effectiveness in predicting the best mutation operators for programs not tested with mutation. **Conclusions:** Considering the configurations selected for the experiments, the results presented several combinations that are effective to predict the best mutation operators for untested programs, where the combinations that involved distance functions between strings stood out. Regarding the program abstractions used to calculate similarity, the original source code seems to be as relevant as the obfuscated source code and internal complexity metrics.

Key-words: Software Testing. Mutation Testing. Cost Reduction Techniques. Similarity of programs. Selective Mutation. Clustering.

Lista de ilustrações

Lista de tabelas

Sumário

Lista de abreviaturas e siglas

AMC	*Average Method Complexity*
ARFF	*Attribute-Relation File Format*
BIC	*Bayesin Information Criterion*
Ca	*Afferent Couplings*
CAM	*Cohesion Among Methods of Class*
CBM	*Coupling Between Methods*
CBO	*Coupling Between Objects*
CC	*Cyclomatic Complexity*
Ce	*Efferent Couplings*
CK	*Chidamber e Kemerer*
CSV	*Comma Separated Values*
DAM	*Data Access Metric*
DIT	*Depth of Inheritance Tree*
EM	*Expectation Maximization*
GST	*Greedy String Tiling*
HTML	*HyperText Markup Language*
IC	*Inheritance Coupling*
JAR	*Java ARchive*
LCOM	*Lack of Cohesion in Methods*
LOC	*Lines of Code*
MFA	*Measure of Functional Abstraction*
MOA	*Measure of Aggregation*
MS	*Mutation Score*

NCD	*Normalised Compression Distance*
NOC	*Number of Children*
NPM	*Number of Public Methods*
QMOOD	*Quality Model for Object-Oriented Design*
RFC	*Response For a Class*
STWV	*String To Word Vector*
UML	*Unified Modeling Language*
V&V	Verificação & Validação
wCNN	*Wavelet Convolutional Neural Network*
Weka	*Waikato Environment for Knowledge Analysis*
WMC	*Weighted Methods per Class*
XML	*eXtensible Markup Language*

1 Introdução

O sucesso do desenvolvimento de um software está diretamente relacionado à execução correta de diversas etapas. Dentre essas, pode-se citar o teste de software, que tem como proposta evidenciar defeitos ainda não identificados no projeto (MYERS; BADGETT; SANDLER, 2011). Esses defeitos podem estar relacionados, por exemplo, à falta de informações na análise de requisitos, ou até mesmo ao uso indevido de um operador lógico no código-fonte.

Existe um conjunto de ações, coletivamente chamadas de "Verificação & Validação" (V&V), cuja finalidade é garantir que tanto o modo pelo qual o software está sendo construído quanto o produto em si estejam em conformidade com o especificado (DELAMARO; MALDONADO; JINO, 2007). Dentre essas atividades, o teste de software é um processo, ou uma série de processos, empregado para garantir que o código do programa faça o que foi projetado para fazer e, de maneira análoga, que ele não faça nada inesperado (MYERS; BADGETT; SANDLER, 2011). Deste modo, o software deve ser previsível e consistente.

Myers, Badgett e Sandler (2011) dizem que a atividade de teste é o processo de executar um programa com a intenção de revelar a presença de defeitos, sendo que um bom caso de teste é aquele que possui uma chance maior de revelar um defeito ainda não percebido. Um dos problemas da realização dos testes é a escolha dos casos de teste aplicados no programa. Este problema ocorre, pois, para que o programa seja considerado imune aos defeitos, seria necessário executar todos os valores possíveis de entrada e então observar as saídas. Entretanto, conforme o tamanho do programa, esse procedimento pode ser impraticável por conta de restrições de tempo e custo para a sua realização. Deste modo, a escolha correta dos casos de teste estará diretamente relacionada à identificação de defeitos do projeto.

Uma das abordagens – mais precisamente, um critério de teste – que apresenta um ótimo resultado na identificação de defeitos é o teste de mutação (DEMILLO; LIPTON; SAYWARD, 1978; HAMLET, 1977). A partir da versão original do software, o teste de mutação consiste em gerar versões com pequenas modificações em que cada versão modificada é considerada um *mutante*. Deste modo, o esperado é que o resultado obtido na execução do programa original, em pelo menos um caso de teste, seja diferente de cada versão modificada.

DeMillo, Lipton e Sayward (1978) argumentaram que é fundamental a aplicação de técnicas que indiquem como testar o software, possibilitando que essa atividade possa ser conduzida de forma planejada e sistemática. O teste de mutação, por ser uma avaliação

sistemática, permite o controle da rotina de teste, possibilitando mensurar a qualidade do conjunto de testes aplicado para a identificação de defeitos.

O teste de mutação ainda não é amplamente utilizado na indústria por incorrer em um alto custo de aplicação (JIA; HARMAN, 2011; PAPADAKIS et al., 2015; FERRARI; PIZZOLETO; OFFUTT, 2018). De acordo com Pizzoleto et al. (2019), existem alguns fatores que implicam nesse alto custo. Entre eles, pode-se citar a grande quantidade de mutantes que um software pode gerar, assim como o fato de que a identificação de potenciais mutantes equivalentes depende da análise do testador do software. Diversas pesquisas abordam técnicas para redução de custo sem afetar de maneira considerável a sua eficiência (PIZZOLETO et al., 2019). Entretanto, diversos trabalhos que aplicaram técnicas de redução de custo em teste de mutação afirmam que os resultados da aplicação de técnicas são válidos somente para os programas que foram alvos dos experimentos realizados (OMAR; GHOSH, 2012; SIAMI-NAMIN; ANDREWS; MURDOCH, 2008; LACERDA; FERRARI, 2014).

1.1 Definição do Problema e Motivação

Redução de custo é considerado mandatório para que o teste de mutação seja aceito na indústria de projetos de software. Com isso em mente, pesquisadores propuseram várias técnicas para alcançar esse objetivo. Inicialmente, as técnicas foram classificadas como *do fewer*, *do faster* e *do smarter* (OFFUTT; UNTCH, 2000), e mais recentemente classificados de acordo com a meta principal de redução de custos como, por exemplo, reduzir o número de mutantes, detectar automaticamente mutantes equivalentes, ou evitar a criação de certos mutantes (por exemplo, redundantes e equivalentes) (FERRARI; PIZZOLETO; OFFUTT, 2018; PIZZOLETO et al., 2019).

Mutação seletiva (OFFUTT; ROTHERMEL; ZAPF, 1993), algoritmos evolucionários (baseados em aprendizado de máquina) (TITCHEU CHEKAM et al., 2020) e mutação de ordem superior (*Higher Order Mutation*) (POLO; PIATTINI; GARCÍA-RODRÍGUEZ, 2009) são exemplos de técnicas clássicas ou técnicas que atualmente são amplamente exploradas, ou ambas, para reduzir o custo da mutação. No entanto, essas e muitas outras técnicas são difíceis de generalizar. Por exemplo, Kurtz et al. (2016) encontraram economias de custo significativas ao investigar conjuntos suficientes de operadores de mutação, mas somente se cada programa for analisado individualmente. Segundo os autores, o conjunto ideal suficiente é diferente para cada programa.

Em uma linha de trabalhos recentes (DALLILO; PIZZOLETO; FERRARI, 2019; PIZZOLETO et al., 2020) tem-se explorado a similaridade entre programas como forma de se reutilizar conhecimento obtido com a aplicação de técnicas de redução de custo de teste de mutação em grupos de programas específicos. Nessa direção, Pizzoleto et al.

(2020) apresentaram o *framework SiMut* (*Program **Si**milarity to support Cost Reduction of **Mut**ation*). O objetivo desse *framework* é auxiliar na redução de custo do teste de mutação baseando-se em experimentos anteriores e na similaridade entre programas. Para atingir essa proposta, dado um novo programa não testado u, o *framework* aplica a u as mesmas estratégias de redução de custo aplicadas ao grupo de referência G, que é um grupo formado por programas já testados com o teste de mutação e que são similares a u. Para efeitos experimentais, no final verifica-se a consistência dos resultados em termos de redução de custo e qualidade dos conjuntos de teste.

Para que o *SiMut* chegue aos resultados, o *framework* computa abstrações (por exemplo, métricas internas ou código ofuscado) dos programas envolvidos e, na sequência, essas abstrações são utilizadas como entrada na etapa de similaridade para então formar os grupos de programas mais similares. Na apresentação do *SiMut*, os autores implementaram uma proposta inicial para validar os conceitos e permitir explorar novas ideias. O estudo piloto desenvolvido utilizou métricas internas como abstração e o algoritmo de agrupamento (do inglês, *clustering*) X-Means(PELLEG; MOORE, 2000) na etapa de similaridade. Nos experimentos apresentados, foi possível constatar que essa combinação de abstração e similaridade se mostrou promissora em alguns conjuntos de programas.

Apesar dos estudos citados, a aplicação do teste de mutação continua sendo um desafio para a comunidade de desenvolvimento de software. Pesquisas recentes (KINTIS; MALEVRIS, 2013; TITCHEU CHEKAM et al., 2020; DALLILO; PIZZOLETO; FERRARI, 2019; PIZZOLETO et al., 2020) apresentaram, utilizando similaridade entre programas, novas formas que podem auxiliar na redução do custo do teste de mutação. Na linha do *SiMut*, a possibilidade de realizar estudos empíricos explorando-se de novas combinações de abstrações e similaridades, que podem trazer novos horizontes para a área, motivaram o desenvolvimento deste trabalho, que apresenta uma implementação mais robusta do *framework SiMut* e análises mais aprofundadas de combinações variadas de técnicas de abstrações e similaridades.

1.2 Objetivos

O objetivo deste trabalho é apresentar como a versão mais completa do *framework SiMut* (PIZZOLETO et al., 2020) foi implementada, além de discutir uma análise de resultados de 20 configurações diferentes executadas por meio dele. Seguindo os estudos citados anteriormente que utilizaram a similaridade entre programas para reduzir o custo do teste de mutação (DALLILO; PIZZOLETO; FERRARI, 2019; PIZZOLETO et al., 2020), neste trabalho buscou-se aprofundar os benefícios dessa prática na área de teste de mutação. Os objetivos são detalhados a seguir.

- *Implementação da versão mais robusta do SiMut*: descreve-se neste trabalho como

a versão evoluída do *SiMut* foi implementada na qual diferentes ferramentas foram acopladas para a etapa de abstração (em particular, para ofuscamento de código-fonte e para cálculo de métricas internas), e para a etapa da similaridade (em particular, para executar diferentes algoritmos de *clustering*, detecção de plágio de código-fonte e funções de distância/similaridade entre *strings*).

- *Experimentação com análise e discussão dos resultados*: discutem-se neste trabalho os resultados coletados após a execução de 20 configurações no *framework SiMut* implementado. Para avaliar a efetividade das configurações executadas, a capacidade do grupo de referência G de prever com precisão os parâmetros de redução de custo para a classe não testada u é comparada com os parâmetros calculados para outros grupos de programas menos semelhantes a u. Além disso, nos experimentos, também utilizaram-se grupos de programas formados aleatoriamente como base para comparação de resultados. Para cada experimento executado, 10 grupos aleatórios foram formados com aproximadamente 20% das classes presentes no banco de dados utilizado, e os resultados médios obtidos para os grupos aleatórios são utilizados na comparação com o grupo de referência G. No final do trabalho, na análise dos resultados, são destacados os benefícios de cada técnica utilizada e também quais combinações de abstrações e similaridades podem ser mais promissoras para apoiar a redução de custo do teste de mutação.

Em estudo anterior (DALLILO; PIZZOLETO; FERRARI, 2019), implementou-se uma versão inicial do *framework SiMut*, conceitualmente apresentado em estudo posterior por Pizzoleto et al. (2020). Essa versão desenvolvida anteriormente não foi projetada para receber novas abordagens de abstrações ou similaridades. Deste modo, essa implementação inicial não foi reaproveitada neste trabalho de mestrado, em que se objetivou implementar uma nova versão mais robusta com uma estrutura capaz de facilitar futuras adições de ferramentas, além das que já foram adicionadas a ela.

1.3 Organização

O restante deste trabalho está organizado da seguinte forma: A fundamentação teórica que serviu de embasamento para este trabalho é apresentada no Capítulo 2; no Capítulo 3 apresentam-se os detalhes da implementação do *framework SiMut*; no Capítulo 4 detalha-se como os experimentos envolvendo diferentes formas de abstração e similares foram realizados; no Capítulo 5 são apresentados os resultados e análises em cima dos resultados obtidos; e, por fim, o Capítulo 6 encerra-se este trabalho, apresentando-se os resultados obtidos, os trabalhos relacionados, as contribuições, as limitações e os possíveis trabalhos futuros.

2 Fundamentação Teórica

2.1 Considerações Iniciais

Neste capítulo apresenta-se o embasamento teórico para a realização deste trabalho. Para atingir os objetivos descritos no capítulo anterior, a fundamentação teórica é direcionada para a utilização da similaridade em benefício da redução de custo do teste de mutação. Com isso, apresentam-se os fundamentos do teste de software, o conceito do *SiMut*, framework que foi implementado e utilizado nos experimentos deste trabalho; técnicas de abstrações de programas e técnicas de similaridades entre programas. O restante deste capítulo está organizado da seguinte forma: na seção 2.2 apresentam-se os conceitos fundamentais da área de teste de software, com foco no teste de mutação e nas técnicas de redução de custo; na seção 2.3 o conceito do *SiMut* é abordado, explicando o seu funcionamento e como esse framework contribui para a área de teste de mutação; na seção 2.4 apresentam-se formas de abstrações de programas, envolvendo ofuscador de código e métricas de internas de códigos; por fim, na seção 2.5 apresentam-se as abordagens de similaridades entre programas utilizadas neste trabalho, citando técnicas de plagiarismo, métricas de diversidade de *string* e algoritmos de clusterização.

2.2 Teste de Software

Nesta seção discutem-se os fundamentos sobre teste de software, partindo de assuntos introdutórios sobre a área de teste, até assuntos mais específicos sobre teste de mutação, que está diretamente relacionado a este trabalho. Além disso, também apresentam-se algumas técnicas de redução de custo do teste de mutação investigadas pela comunidade científica.

2.2.1 Definição de Teste de Software

Myers, Badgett e Sandler (2011) definem o teste de software como um processo de executar um programa com a intenção de revelar a presença de defeitos. Delamaro, Maldonado e Jino (2007) mencionam que o intuito do teste de software é verificar se o programa, após receber algumas entradas, tem um comportamento ou resultado de acordo com o esperado, em um cenário no qual o comportamento resultante não seja coerente com o esperado; dessa forma, evidenciou-se a presença de defeitos no software.

O comportamento inesperado de um programa pode ser ocasionado, por exemplo, por erros de programação, como o uso incorreto de operador relacional. A implementação

de um programa baseada em uma interpretação equivocada das análises de requisitos também pode gerar comportamentos inesperados para o usuário.

2.2.2 Defeito, Erro e Falha

Na área de teste de software, os termos defeito, erro e falha possuem significados diferentes. *Defeito* (do inglês, *fault*) pode ser definido como um passo, processo ou definição de dados incorretos; deste modo, caso o trecho defeituoso seja executado, o programa não terá um comportamento esperado. A existência de um defeito pode ocasionar a geração de um *erro* (do inglês, *error*) no programa, que acontece por um estado de execução inconsistente ou inesperado. O estado incorreto de um programa pode então gerar *falhas* (do inglês, *failure*), o que pode fazer com que o resultado produzido na execução seja diferente do resultado esperado, sendo observável externamente (DELAMARO; MALDONADO; JINO, 2007).

Nota-se que um defeito em um programa pode nem sempre ocasionar erros, pois determinados defeitos somente serão executados com um grupo restrito de valores de entradas. Deste modo, o surgimento de um erro, a partir de um defeito, dependerá da entrada de dados que usuário está utilizando durante a execução do programa. Salienta-se ainda que mesmo o surgimento de um erro não é garantia que o software irá falhar, já que de acordo com o erro e com as circunstâncias, o software ainda poderá continuar funcionando dentro das suas especificações.

2.2.3 Técnicas de Teste de Software

Delamaro, Maldonado e Jino (2007) afirmam que a realização de teste de um software é complexa, pois existem diversos fatores que podem colaborar para o surgimento de falhas. Por conta disso, existem várias técnicas de teste de software que podem ser empregadas e, entre elas, pode-se destacar: teste funcional, teste estrutural e teste baseado em defeitos.

2.2.3.1 Teste Funcional

O teste funcional, também conhecido como teste de caixa preta (MYERS; BADGETT; SANDLER, 2011), é uma técnica em que são fornecidas entradas de dados para o software, e na sequência as saídas geradas são avaliadas se estão em conformidade com os objetivos da aplicação. Nessa técnica, os detalhes de implementação não são considerados e o software é avaliado de acordo com a interação do usuário. Um ponto negativo deste teste é dificuldade para se quantificar a atividade de teste, já que não se pode garantir que partes críticas ou essenciais do programa sejam executadas na rotina de testes.

2.2.3.2 Teste Estrutural

Também conhecido como teste de caixa branca (MYERS; BADGETT; SANDLER, 2011), a técnica de teste estrutural estabelece os requisitos de teste com base em uma dada implementação, requerendo a execução de partes ou de componentes elementares do programa. Os caminhos lógicos do software são testados, fornecendo-se casos de teste que põem à prova tanto conjuntos específicos de condições e/ou laços bem como pares de definições e usos de variáveis. Em geral, a aplicação do teste estrutural utiliza uma representação de programa conhecida como *Grafo de Fluxo de Controle* ou *Grafo de Programa*.

2.2.3.3 Teste Baseado em Defeitos

O teste baseado em defeitos procura gerar os casos de teste com base no conhecimento dos defeitos comuns introduzidos pelos desenvolvedores durante a elaboração dos programas (MORELL, 1990). O critério de teste de mutação (DEMILLO; LIPTON; SAYWARD, 1978; HAMLET, 1977), fundamental para a elaboração deste trabalho, cria versões com pequenas modificações sintáticas, nas quais os casos de teste são aplicados, e em seguida observa-se o comportamento dessas versões modificadas, denominadas mutantes.

Para este trabalho foi abordado o teste de mutação. Dessa forma, na próxima seção discutem-se informações específicas sobre esta técnica de teste de software.

2.2.4 Teste de Mutação

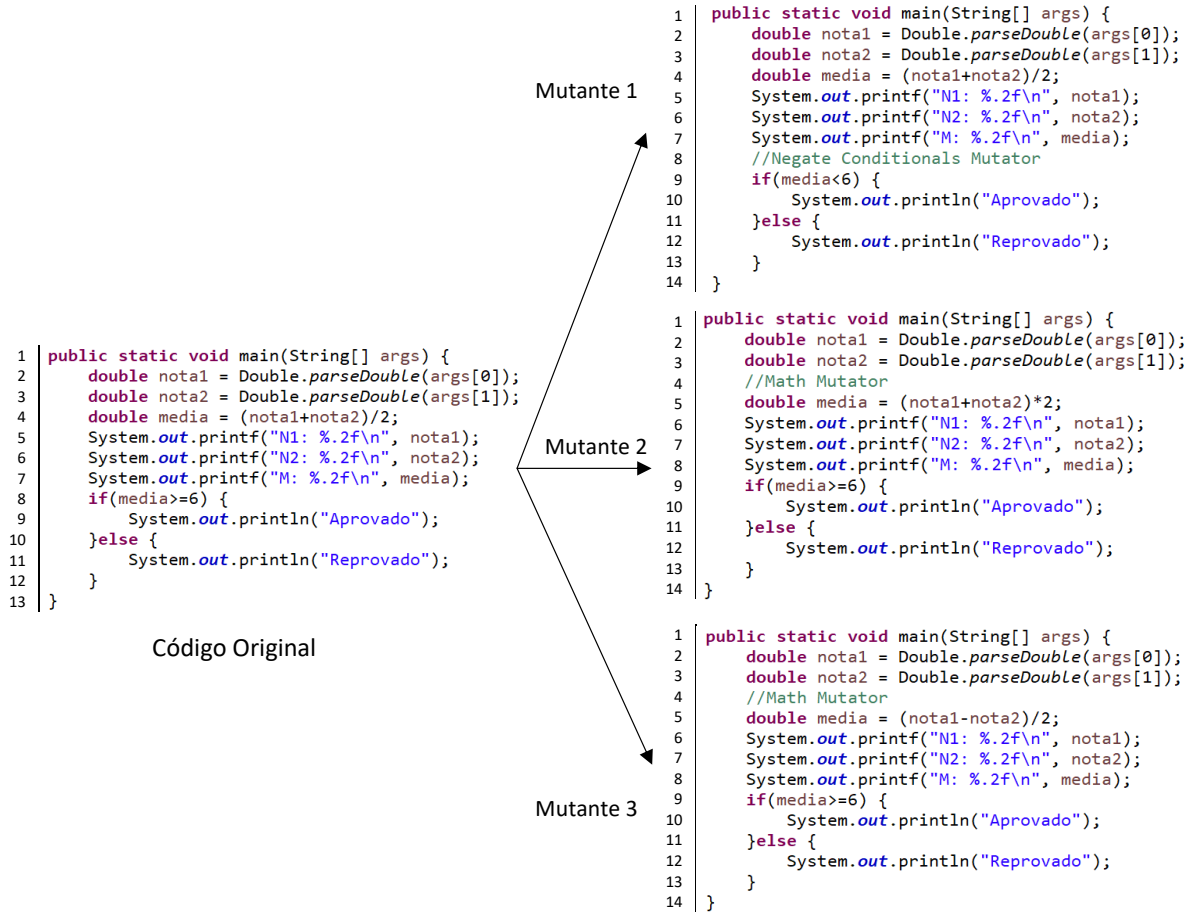
DeMillo, Lipton e Sayward (1978) descreveram as ideias principais sobre a técnica de teste de mutação, apresentando as duas hipóteses que fundamentam a sua utilização: a hipótese do *programador competente* e a hipótese do *efeito de acoplamento*. A hipótese do *Programador Competente* afirma que os programadores experientes escrevem programas corretos ou muito próximos do correto. Já a hipótese do *Efeito de Acoplamento* diz que casos de teste capazes de revelar defeitos simples, também são capazes de revelar defeitos mais complexos, criando um efeito cascata de defeitos acoplados. A diferença entre um defeito simples e um defeito complexo é definido por Offutt (1992) como: *defeito simples é um defeito que pode ser corrigido fazendo uma única alteração em uma instrução de origem, enquanto defeito complexo é um defeito que não pode ser corrigido fazendo uma única alteração em uma instrução de origem.*

Baseando-se nessas hipóteses, para a aplicação do teste de mutação, são geradas diversas versões com pequenas modificações de um programa que está sendo testado, como se defeitos estivessem sendo inseridos no programa original. Essas versões modificadas do programa são chamadas de *mutantes* e contém defeitos típicos do processo de implementação de software como, por exemplo, o uso indevido de um operador matemático (DELAMARO;

MALDONADO; JINO, 2007).

Na Figura 1, como forma de exemplificar o que seriam essas pequenas modificações, apresentam-se três mutantes gerados a partir de um código-fonte. Neste exemplo, a mutação realizada está abaixo do comentário de cada mutante criado. No *Mutante 1*, inverteu-se o operador relacional utilizado na estrutura condicional de $>=$ para $<$. No *Mutante 2* alterou-se o operador matemático de *Divisão* da expressão matemática do cálculo da variável *média* para o operador de *Multiplicação*. Já no *Mutante 3*, inverteu-se o operador matemático de *Soma* utilizado na expressão matemática do cálculo da variável *media* para o operador *Subtração*.

Figura 1 – Exemplo de mutantes gerados.



Fonte: Produzido pelo autor.

2.2.4.1 Escore de Mutação

O escore de mutação (do inglês, *mutation score*, ou *MS*) é um valor entre 0 e 1 que representa a qualidade do conjunto de testes em relação a um determinado grupo de mutantes. Na prática, os mutantes que demonstrarem resultados diferentes do programa em análise — ou seja, os mutantes que forem *mortos* pelos testes — são eliminados e então

é considerado que o programa está imune aos problemas simulados. Existem situações em que o mutante gerará o mesmo resultado que o programa original, independentemente do caso de teste. Quando isso ocorre, o mutante é chamado de *equivalente*. Ao final do teste é calculado o *MS* (Equação 2.1), que é a relação do número de mutantes mortos *(K)* com o total gerado *(M)*, ignorando-se os equivalentes *(E)*. Quanto mais próximo de 1 for o *MS*, melhor o conjunto de testes. Caso M seja igual a E, é considerado que nenhum mutante "matável" foi gerado.

$$MS = \left(\frac{K}{M - E}\right) \tag{2.1}$$

No exemplo da Figura 1, três mutantes foram gerados a partir do código-fonte original. Deste modo, a expectativa do testador é que em ao menos um caso de teste, o resultado obtido para cada mutante seja diferente do programa original. Caso o resultado para o mutante seja diferente, como explicado anteriormente, considera-se que o programa está imune ao defeito simulado e o mutante é dado como morto.

Na Figura 2 apresenta-se um exemplo de mutante equivalente. Do lado esquerdo está o código-fonte do programa original, enquanto que do lado direito está o código-fonte de um mutante equivalente. Nota-se que no mutante foi alterado somente o operador relacional da estrutura de repetição de *maior* para *diferente*. Observa-se que não haverá qualquer diferença entre os resultados gerados por estes dois programas, tendo em vista que a estrutura de repetição continuará executando a mesma quantidade de vezes, sem impactar o comportamento entre eles, já que tanto a expressão lógica com $i > 0$, quanto com i *!= 0* somente será negativa quando i decrementar para 0. Deste modo, como comentado anteriormente, quando o mutante gera sempre mesmo resultado que o programa original, independentemente dos valores de entrada, ele é chamado de mutante equivalente.

Figura 2 – Exemplo de mutante equivalente.

Código Original

```
public static void main(String[] args) {
    int fim = Integer.parseInt(args[0]);
    for (int i = fim; i > 0; i--) {
        int resto = i % 2;
        if(resto == 0) {
            System.out.printf("%d é par\n", i);
        }else {
            System.out.printf("%d é ímpar\n", i);
        }
    }
}
```

Mutante Equivalente

```
public static void main(String[] args) {
    int fim = Integer.parseInt(args[0]);
    for (int i = fim; i != 0; i--) {
        int resto = i % 2;
        if(resto == 0) {
            System.out.printf("%d é par\n", i);
        }else {
            System.out.printf("%d é ímpar\n", i);
        }
    }
}
```

Fonte: Produzido pelo autor.

2.2.4.2 Técnicas de Redução de Custo do Teste de Mutação

Nas últimas décadas, o teste de mutação foi extensivamente investigado (PIZZOLETO et al., 2020; PAPADAKIS et al., 2019). Porém, por ainda ser custoso, demanda-se reduzir o custo de sua aplicação para que o teste de mutação seja mais atraente para o mercado. Com esse objetivo, diversas técnicas foram criadas. Offutt e Untch (2000), em uma classificação de trabalhos nessa linha, agruparam as técnicas de redução de custo em três categorias: *do fewer*, *do smarter* e *do faster*.

Fazer menos (do inglês, *do fewer*) tem como objetivo reduzir o número de mutantes a serem executados sem que ocorram perdas significativas de informações, como perda de efetividade do conjunto de testes. Mutação seletiva (OFFUTT; ROTHERMEL; ZAPF, 1993), presente nessa categoria, objetiva evitar a aplicação de operadores de mutação que produzem mutantes difíceis de serem eliminados, ou que produzem uma maior quantidade de mutantes. Também nessa categoria podem-se citar a técnica de operadores essenciais (BARBOSA; MALDONADO; VINCENZI, 2001), que visa utilizar somente um subconjunto relevante de operadores de mutação, assim como a técnica *One-Op* (UNTCH, 2009), na qual é utilizado apenas o operador mais relevante que pode ser capaz de demandar testes tão poderosos quanto a um grupo de vários operadores.

Fazer de forma mais inteligente (do inglês, *do smarter*) propõe gerenciar o esforço computacional externo ou interno. Mutação fraca (HOWDEN, 1982) se enquadra nessa categoria, na qual o estado do mutante é comparado com o estado do programa original imediatamente após o ponto de execução da instrução mutada, em vez de verificar os resultados após a execução completa do programa e do mutante.

Fazer de forma mais rápida (do inglês, *do faster*) tem como objetivo executar mutantes na forma mais rápida possível. Para essa técnica, pode-se citar meta-mutantes (UNTCH, 1992), que consiste em embutir vários mutantes em apenas um.

Para este trabalho, foi utilizada uma abordagem *do fewer*, sendo ela a técnica de Mutação Seletiva (OFFUTT; ROTHERMEL; ZAPF, 1993), na qual adotou-se como estratégia um ranqueamento dos mutantes pelo escore de mutação para identificar os mutantes mais relevantes, ou seja, os com maior escore de mutação. Deste modo, por característica das técnicas *do fewer*, essa abordagem propõe reduzir o custo computacional por meio da escolha de operadores mais relevantes.

Nota-se que a aplicação de uma técnica de redução de custo em um programa pode não ter a mesma eficiência se aplicada em algum outro programa (SIAMI-NAMIN; ANDREWS; MURDOCH, 2008; OMAR; GHOSH, 2012; LACERDA; FERRARI, 2014). Deste modo, faz-se necessário a identificação da melhor estratégia de redução de custo do teste de mutação quando aplicada em um novo programa, sendo que a abordagem encapsulada no *framework SiMut*, que é elemento central deste trabalho, auxilia na

definição desta estratégia.

Ressalta-se que recentemente, em uma revisão sistemática da literatura que envolveu uma análise de 175 estudos relacionados à redução do custo de teste de mutação, Pizzoleto et al. (2019) propuseram dois novos sistemas de classificação. A primeira se refere aos objetivos primários de redução de custo pretendidos pelos estudos. Entre os objetivos primários, está a redução do número de mutantes a serem executados e detectar automaticamente mutantes equivalentes. A segunda classificação se refere às técnicas empregadas para atingir os objetivos primários. Dentre essas, estão aquelas que são de uso tradicional para análise de programas, assim como técnicas que são específicas ao teste de mutação como, por exemplo, a mutação seletiva.

Em trabalhos anteriores (DALLILO; PIZZOLETO; FERRARI, 2019; PIZZOLETO et al., 2020), constatou-se que o resultado de uma comparação entre programas testados e não testados pode servir como recurso primário para viabilizar a aplicação do teste de mutação a custo reduzido. Nessa direção, Pizzoleto et al. (2020) apresentaram o *framework* $SiMut$ que objetiva auxiliar na redução de custo do teste de mutação baseando-se em experimentos anteriores e na similaridade entre programas. Por se tratar de um *framework* que foi implementado e utilizado neste trabalho, na próxima seção apresentam-se informações fundamentais sobre o $SiMut$.

2.3 Conceito do $SiMut$

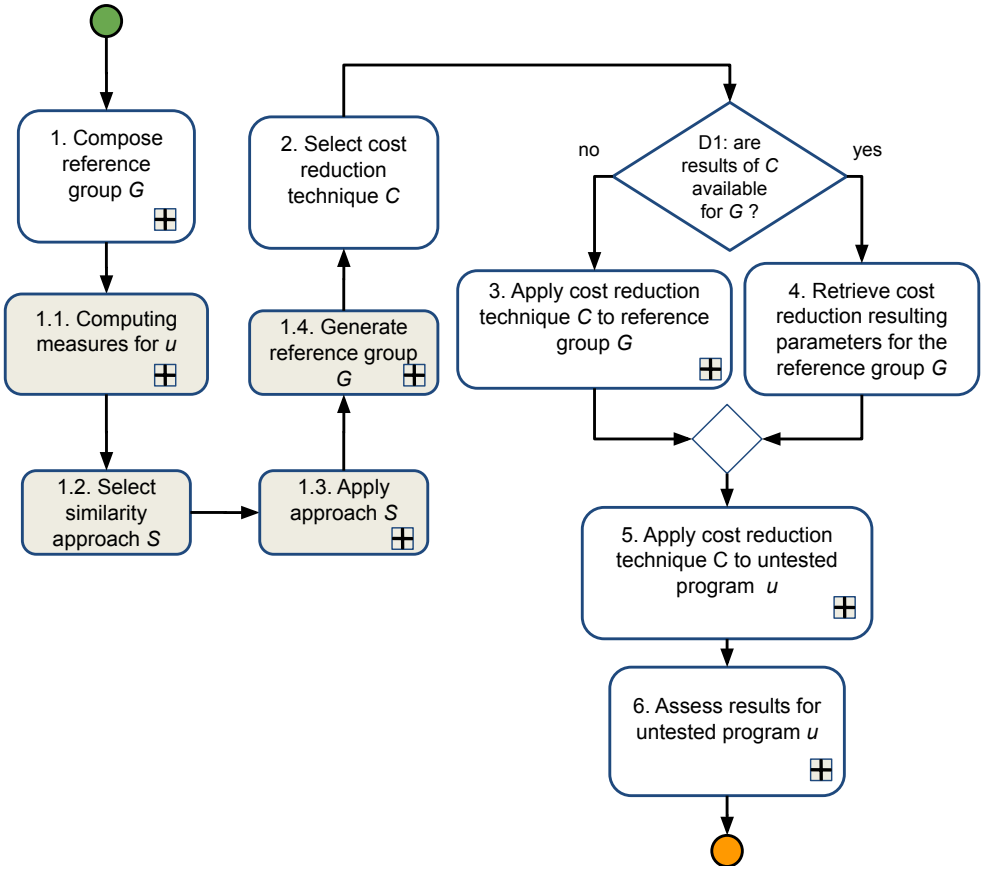
Conceitualmente, o *framework* $SiMut$, para ser executado, requer um conjunto de programas P que já foram testados com mutação; ou seja, P inclui apenas programas para os quais artefatos relacionados a mutações, como casos de teste, mutantes e resultados de execução de mutações, estão disponíveis. Dado P, um programa não testado u, e uma abstração de programa particular a abordagem embutida no $SiMut$ (por exemplo, métricas internas ou código-fonte processado) consiste em identificar o grupo de programas que mais se assemelha a u. Chama-se esse grupo de *grupo de referência* G. $SiMut$ então calcula os parâmetros de redução de custo de mutação para G e sugere que os mesmos parâmetros sejam aplicados a u.

2.3.1 Etapas do $SiMut$

Em resumo, o $SiMut$ compreende um processo geral e um subprocesso chave que é responsável por compor grupos de programas semelhantes. Na Figura 3[1] retratam-se esses processos, na qual as caixas com o fundo branco representam o processo geral e englobam os seis subprocessos ou atividades descritas a seguir:

[1] Para manter a consistência entre documentos, a figura foi mantida na língua inglesa, retratando exatamente a figura publicada anteriormente (GUARNIERI; PIZZOLETO; FERRARI, 2022).

Figura 3 – Visão simplificada do processo do *SiMut*.



Fonte: Guarnieri, Pizzoleto e Ferrari (2022) .

- 1 - *Compose reference group G* (*Compor grupo de referência G*) - São selecionados todos os programas para a execução do *framework* e, após as etapas de abstração e similaridade, o grupo de referência G é formado, contendo os programas mais similares ao programa u.
- 2 - *Select cost reduction technique C* (*Selecionar a técnica de redução de custo C*) - Nesta etapa, é selecionada a técnica de redução de custo que será aplicada ao grupo de referência G, para identificar os melhores resultados para o programa não testado u;
- 3 - *Apply cost reduction technique C to reference group G* (*Aplicar a técnica de redução de custo C ao grupo de referência G*) - essa etapa somente é executada se a técnica de redução de custo C ainda tiver sido computada para o grupo de referência G;
- 4 - *Retrieve cost reduction resulting parameters for the reference group G* (*Recuperar parâmetros resultantes da redução de custos para G*) - esta etapa somente é executada se a técnica de redução de custo C já tiver sido computada para o grupo de referência G e os resultados estiverem armazenados para novas execuções;
- 5 - *Apply cost reduction technique C to untested program u* (*Aplicar a técnica de redução de custo C ao programa não testado u*) - nesta etapa, usando os parâmetros de redução

de custo calculados ou recuperados, a técnica de redução de custo C é aplicada ao programa não testado u. Conjunto suficiente de operadores de mutação ou um algoritmo de aprendizado de máquina treinado são exemplos de parâmetros de redução de custos. Esses parâmetros são aplicados ao programa não testado u.

- 6 - *Assess results for untested program u* (*Avaliar os resultados do programa não testado u*) - essa etapa existe para fins de estudos de efetividade dos resultados coletados para o programa não testado, sendo opcional para as demais utilizações. Na prática, essa etapa foi criada para validar a proposta do *framework*.

A etapa *Compor grupo de referência G* tem um papel fundamental no *framework* como um todo, pois é ela que indica os programas mais similares ao programa não testado u. Esse grupo serve para identificar os melhores parâmetros de redução de custo para u, baseando-se nos programas mais similares. Essa etapa é dividida em quatro subprocessos ou atividades:

- 1.1 - *Computing measures for u* (*Computar medidas para o programa não testado u*) - esta etapa computa as abstrações para o programa não testado u. Nota-se que nesta etapa assume-se que já foram computadas as abstrações para os demais programas da execução.

- 1.2 - *Select similarity approach S* (*Selecionar a abordagem de similaridade S*) - nesta etapa seleciona-se a abordagem de similaridade que será aplicada aos programas envolvidos na execução.

- 1.3 - *Apply approach S* - (*Aplicar a abordagem S*) - Nesta etapa, utilizando como entrada as abstrações computadas para os programas envolvidos na execução, formam-se os grupos de programas mais similares.

- 1.4 - *Generate reference group G* (*Gerar o grupo de referência G*) - nesta etapa, é gerado o grupo de referência G, que contém um grupo de programas mais similares ao programa não testado u. É com base nesses programas similares que são selecionados os parâmetros de redução de custo para o programa não testado u.

2.3.2 Abstrações e Similaridade de Programas

Medidas podem ser computadas de variadas formas de abstrações. Como exemplos de abstrações de programas, podem-se citar métricas internas de programas, um código simplificado, ou uma especificação de alto nível de um software, como um diagrama de sequência do UML (*Unified Modeling Language*). A etapa de abstração é usada para simplificar o processo de análise de programas complexos.

É importante citar que as medidas também devem estar disponíveis para os programas que compõem a base de dados em uso, pois servem de entrada para a etapa de cálculo de similaridade. Portanto, as medidas para os programas do banco de dados devem ser computadas de forma antecipada (e estarem armazenadas no banco de dados), ou, então, serem computadas em tempo de execução do *framework* $SiMut$.

Por fim, a *abordagem de similaridade*, utilizando como entrada as abstrações computadas para os programas envolvidos na execução, é destinada para aplicar um ou mais algoritmos de cálculo de similaridade e produzir o grupo de referência G, ou seja, programas que são similares ao programa u.

2.4 Técnicas de Abstrações

Como explicado na seção anterior, seguindo o conceito do $SiMut$ (PIZZOLETO et al., 2020), para formar os grupos de referência G na etapa de similaridade, os programas passam por um processo de abstração, que cria uma representação do programa original. Essa etapa é importante, pois pode simplificar o processo de análise de programas complexos. Na prática essa técnica cria uma relação entre o programa original e um programa abstrato (COUSOT; COUSOT, 1977), que pode ser utilizado na etapa da similaridade do *framework*.

Nesta seção apresentam-se as abstrações utilizadas neste trabalho: métricas internas e código-fonte ofuscado.

2.4.1 Métricas Internas

Li (2000) diz que métricas de softwares são medidas de software e a principal proposta delas é ajudar a prever e planejar o desenvolvimento de um software. Além disso, o autor fala que o cômputo de métricas internas também auxilia no controle da qualidade do software e no esforço de desenvolvimento. Os autores Chidamber e Kemerer (1994) falam que as métricas internas podem servir de apoio para melhorar a qualidade do software, além de também auxiliarem a estimar o custo e o cronograma de projetos de software.

A escolha de métricas internas como uma forma de abstração para este trabalho foi baseada em estudos recentes (CRUZ; ELER, 2017; DALLILO; PIZZOLETO; FERRARI, 2019; PIZZOLETO et al., 2020) em que se utilizaram métricas internas como recurso para auxiliar no teste de software, assim como em métricas implementadas em uma ferramenta empregada neste trabalho (mais detalhes na subseção 3.3.1). Ao longo dos anos, diversas métricas foram apresentadas, e as consideradas neste trabalho são descritas nas próximas subseções.

2.4.1.1 Métricas de Tamanho

Métricas internas de tamanho objetiva quantificar o tamanho do programa. Entre essas métricas, as utilizadas neste trabalho são listadas a seguir.

- *Lines of Code*(LOC) - computa a quantidade de linhas de código que o programa possui.
- *Number of Classes* - computa a quantidade de classes que o programa possui.
- *Number of Public Methods*(NPM) - computa a quantidade de métodos públicos de cada classe.

2.4.1.2 Métricas de Acoplamento

Martin (1994) propôs várias métricas de acoplamento que podem ser usadas para medir a qualidade de um projeto orientado a objetos em termos da interdependência entre os subsistemas desse projeto. Essas métricas são descritas a seguir:

- *Afferent Couplings* (Ca) - computa a quantidade de classes diferentes que referem-se à classe em análise, através de chamadas de métodos, tipos de retorno, argumentos e exceções.
- *Efferent Couplings* (Ce) - computa a quantidade de classes diferentes que a classe em análise faz referência, através de chamadas de métodos, tipos de retorno, argumentos e exceções.
- *Instability* (I) - computa, baseando-se nos valores das métricas Ca e Ce, o grau de instabilidade da classe em análise.

2.4.1.3 Métricas CK

Chidamber e Kemerer (1994) propuseram seis métricas, chamadas de métricas CK (de Chidamber-Kemerer), relacionadas ao desenvolvimento de programas orientados a objetos. Abaixo tem-se a descrição de cada uma dessas métricas CK:

- *Weighted Methods Per Class* (WMC) - computa a complexidade dos métodos de uma classe. Na prática, está métrica pode representar a soma da complexidade ciclomática de todos os métodos de uma classe. Chidamber e Kemerer (1994) também destacam uma outra abordagem para essa métrica, na qual pode-se considerar valor um para cada método presente na classe, de modo que essa métrica resulte na quantidade total de métodos da classe.

- *Depth of Inheritance Tree* (DIT) - computa a profundidade da árvore de herança para cada classe. Em Java todas as classes herdam Object, portanto o valor mínimo de DIT é um.

- *Number of Children* (NOC) - computa o número de filhos de cada classe, medindo o número de descendentes imediatos de cada classe.

- *Coupling Between Objects* (CBO) - computa o número de classes acopladas a uma outra determinada classe (acoplamento eferentes e acoplamentos aferentes). Esses acoplamentos podem ocorrer por meio de chamadas de métodos, herança, tipos de retorno e etc.

- *Response For a Class* (RFC) - computa o número de métodos diferentes que podem ser executados através do objeto da classe em análise, incluindo todos os métodos acessados dentro da hierarquia da classe.

- *Lack of Cohesion in Methods* (LCOM) - computa a falta de coesão de uma classe. Na prática, essa métrica conta os conjuntos de métodos na classe que não estão relacionados com alguns atributos da mesma classe.

2.4.1.4 Métrica de Henderson-Sellers

Henderson-Sellers (1995) propôs a métrica LCOM3, uma versão alternativa para a métrica LCOM, que computa um valor entre 0 e 2 através da Equação 2.2.

$$LCOM3 = \frac{(\frac{1}{a}\sum_{j=1}^{a}\mu(A_j)) - m)}{1 - m} \tag{2.2}$$

Na equação, a representa o número de atributos na classe, m é o número de métodos na classe e $\mu(A)$ representa o número de métodos que acessam o atributo da classe.

2.4.1.5 Métricas de QMOOD

Bansiya (1999), Bansiya e Davis (2002) propuseram algumas métricas de *Quality Model for Object-Oriented Design* (QMOOD) para avaliar o nível de coesão de métodos, encapsulamento, agregação e abstração. Abaixo listam-se essas métricas:

- *Data Access Metric* (DAM) - computa a razão entre o número de atributos privados (protegidos) e o número total de atributos declarados na classe. O valor da métrica varia entre 0 e 1, sendo desejado o valor máximo 1.

- *Measure of Aggregation* (MOA) - computa o número de atributos da classe cujos os tipos são classes definidas pelo usuário.

- *Measure of Functional Abstraction* (MFA) - computa a razão entre o número de métodos herdados por uma classe e o número total de métodos acessíveis pelos métodos membros da classe. Os construtores e o java.lang.Object(como pai) são ignorados.

- *Cohesion Among Methods of Class* (CAM) - computa o relacionamento entre os métodos de uma classe com base na lista de parâmetros dos métodos. A métrica é calculada usando a soma do número de diferentes tipos de parâmetros de método em cada método dividido por uma multiplicação do número de diferentes tipos de parâmetros do método em toda a classe e número de métodos.

2.4.1.6 Métricas de Tang

Tang, Kao e Chen (1999), em um estudo empírico sobre métricas de orientação a objetos, propuseram outras 3 métricas internas:

- *Inheritance Coupling* (IC) - computa o número de classes pai às quais uma determinada classe está acoplada. Uma classe é acoplada à sua classe pai se um de seus métodos herdados depender funcionalmente dos métodos novos ou redefinidos na classe. Na prática, uma classe é acoplada à sua classe pai se acontecer uma das seguintes situações: um de seus métodos herdados usa um atributo definido em um método novo/redefinido; um de seus métodos herdados chama um método redefinido; um de seus métodos herdados é chamado por um método redefinido e usa um parâmetro definido no método redefinido.

- *Coupling Between Methods* (CBM) - computa o número total de métodos novos/redefinidos aos quais todos os métodos herdados estão acoplados. Acontece um acoplamento quando pelo menos uma das condições de definição da métrica IC é mantida.

- *Average Method Complexity* (AMC) - computa a quantidade média de linhas dos métodos de uma determinada classe.

2.4.1.7 Métrica de Complexidade Ciclomática

A métrica complexidade ciclomática (do inglês, *Cyclomatic Complexity*)(CC), proposta por McCabe (1976), representa a quantidade de caminhos independentes que um programa, ou método de programa, pode seguir. Essa métrica é utilizada em testes de caixa branca, pois auxilia na identificação da quantidade mínima de casos de testes que serão necessários para percorrer todos os caminhos de um determinado programa(MYERS; BADGETT; SANDLER, 2011).

2.4.2 Código-Fonte Ofuscado

Uma outra forma de criar uma abstração de um programa, e empregada neste trabalho, é utilizando um ofuscador de código-fonte. Segundo Linn e Debray (2003), a principal proposta de um ofuscador de código-fonte é dificultar a engenharia reversa de programas, que é uma técnica que possibilita a recriação do código-fonte de um determinado programa a partir do código de máquina, ou seja, do executável do programa. Ainda segundo os autores, a grande preocupação da engenharia reversa para os desenvolvedores está relacionada com encontros de brechas dos programas que possibilitem modificações ou acessos não autorizados, ou então, roubo de propriedade intelectual.

Além de dificultar a engenharia reversa de programas, algumas ferramentas de ofuscamento, como ProGuard[2], também possibilitam a redução significativa dos tamanhos dos executáveis dos programas. Essa redução de tamanho ocorre por meio de recursos como encurtamento de nomes de variáveis, métodos e classes, e também utilizando-se otimização de trechos de códigos, de modo que não comprometa o funcionamento do programa em sua proposta original. Essas otimizações e encurtamentos ocorrem durante o processo de ofuscamento do programa.

Ofuscador de código-fonte tem como característica modificar o programa, mas sem comprometer o objetivo original do programa (ANANTH et al., 2014). Essa técnica também foi utilizada como abstração de programas para este trabalho, pois ela permite a geração de um programa modificado com tamanho reduzido, mas que mantém o seu funcionamento fiel ao programa original. Deste modo, investigou-se se essa forma de abstração pode impactar de forma positiva ou negativa dentro da proposta do *framework* $SiMut$.

2.5 Técnicas de Similaridades

Para atingir os objetivos deste trabalho, utilizaram-se as duas abordagens de abstrações citadas da seção anterior: métricas internas e código ofuscado. Seguindo a proposta do *framework* $SiMut$ implementado, essas abstrações computadas servem de entrada para a etapa de similaridade. Deste modo, nesta seção, apresentam-se os fundamentos sobre similaridade entre programas, além de descrever as abordagens que estão envolvidas diretamente neste trabalho.

2.5.1 Conceitos sobre Similaridade entre Programas

Walenstein et al. (2007) conceituaram o que seria a similaridade dentro do contexto de comparação entre programas. Dividiram em dois tipos: visão sintática e visão semântica,

[2] <https://www.guardsquare.com/proguard> - acessado em Janeiro de 2022.

sendo ambas relacionadas a uma visão tácita do que seria um "Programa". Sintática, ou representacional, de maneira geral, é voltada para uma comparação de estrutura e não de significado. Nesse tipo, a comparação lida com vários níveis de abstrações, como declaração e blocos de códigos. Por outro lado, a similaridade comportamental, ou semântica, refere-se a uma comparação de significado e não de estrutura.

A técnica de comparação de código-fonte de programas é utilizada em diversos contextos como, por exemplo na remoção de redundâncias para compressão, detecção de plágio em códigos, e na diferenciação de programas (*diff*) (WALENSTEIN et al., 2007).

Para este trabalho, as duas formas de similaridade (sintática e semântica) foram utilizadas. Para a identificação de similaridade representacional utilizaram-se métricas de distância de *strings* e algoritmos de *clustering*. Já para a identificação de similaridade semântica, utilizou-se técnica de detecção de plágio de código-fonte. Nas próximas subseções, cada uma dessas técnicas é apresentada, citando os motivos pelas quais elas foram selecionadas.

2.5.2 Detecção de Plágio de Código-Fonte

Técnicas de detecção de plágio de código-fonte foram utilizadas neste trabalho como forma de encontrar similaridade entre programas. O plágio não ocorre apenas em textos acadêmicos ou imagens; é comum também, na maioria dos casos em contexto acadêmico, ocorrerem plágios de código-fonte. O processo de identificação de plágio manualmente, pode ser complexo e demorado de acordo com a quantidade de códigos-fontes em que seja analisar. Por conta disso, pesquisadores (SCHLEIMER; WILKERSON; AIKEN, 2003; PRECHELT et al., 2002) buscam formas para auxiliar na identificação de plágio entre códigos-fontes.

Deste modo, por ser um processo que pode ser demorado, diversas ferramentas buscam automatizar o processo de identificação de indícios de plágio entre códigos-fontes. Nessas ferramentas, nos casos em que a incidência de plágio identificado entre os códigos é alta, torna-se necessário uma análise humana para confirmar a constatação. Entretanto, essa análise humana não é necessária nos casos em que a incidência de plágio é baixa, justificando assim o uso dessas ferramentas.

Para auxiliar nesse processo manual, na JPlag (PRECHELT et al., 2002), ferramenta empregada neste trabalho (mais detalhes na subseção 3.4.3), geram-se diversos arquivos em formato web, destacando as similaridades identificadas entre cada um dos programas. Essa ferramenta foi testada em um contexto acadêmico e avaliada comparando-se vários códigos escritos na linguagem Java criados por estudantes. Como resultado final, os autores constataram aproximadamente 90% de sucesso na identificação de plágio entre os códigos. De maneira geral, ao término da execução dessas ferramentas, computa-se um percentual

de similaridade entre os programas envolvidos na execução.

Para este trabalho, detecção de plágio de código-fonte também foi uma das abordagens escolhidas, por parecer promissora no processo de identificar similaridades entre programas.

2.5.3 Métricas de Distância entre Strings

Métrica de distância entre *strings* é um valor numérico que quantifica a distância entre duas *strings*. Para este trabalho, utilizou-se essa abordagem de similaridade com o objetivo de identificar a similaridade entre códigos-fonte (original ou ofuscado) de programas. Nas próximas subseções apresentam-se as métricas de distância entre *strings* que foram foram utilizadas neste trabalho. A escolha dessas métricas se deu por serem conhecidas, e também por serem computadas através de uma ferramenta empregada neste trabalho (mais detalhes na subseção 3.4.2).

2.5.3.1 Jaccard

Proposto por Jaccard (1901), a métrica de distância Jaccard, dada pela Equação 2.3, retorna um valor entre 0 e 1 que representa a distância entre dois conjuntos de *strings* A e B, na qual quanto mais próximo de 0, mais similares são os dois conjuntos e, quanto mais próximo de 1, mais distantes são os dois conjuntos.

$$d_J(A, B) = \frac{|A \cup B| - |A \cap B|}{|A \cup B|} \tag{2.3}$$

Na Figura 4 apresenta-se um pequeno exemplo de como o cálculo de métrica de distância Jaccard é realizado para dois conjuntos de comandos. Observa-se que a interseção entre os dois conjuntos é igual a 2, enquanto que a junção dos elementos dos dois conjuntos é igual 5. Aplicando a Equação 2.3, o resultado final é de 0.6, deste modo, a distância entre os dois conjuntos de comandos, em uma escala de 0 a 1, é de 0.6.

Na prática, cada código-fonte de programa é representado por um conjunto de *strings* de símbolos ou comandos do código-fonte. Aplicando-se a equação da métrica de distância Jaccard, é então computada a distância entre dois programas.

2.5.3.2 Levenshtein

A métrica de distância Levenshtein, proposta por Levenshtein et al. (1966), computa o número mínimo de modificações (inserções, remoções e substituições) necessárias para transformar uma *string* em uma outra *string*. Na prática, quanto menor o valor da métrica de distância Levenshtein, mais similares são as duas *strings* e, quanto maior esse valor, mais diferentes essas *strings* são.

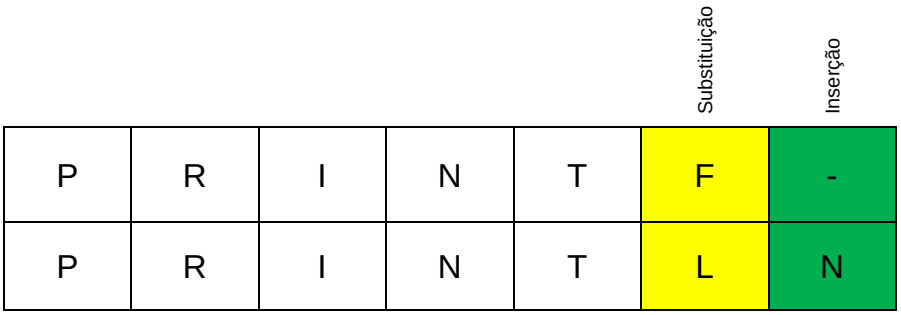
Figura 4 – Exemplo de cálculo de métrica de distância Jaccard.

$$
\begin{aligned}
A &= (\text{“}printf\text{”}, \text{“}int\text{”}, \text{“}double\text{”}, \text{“}for\text{”}) \\
B &= (\text{“}double\text{”}, \text{“}for\text{”}, \text{“}if\text{”}) \\
A \cup B &= (\text{“}double\text{”}, \text{“}for\text{”}, \text{“}if\text{”}, \text{“}int\text{”}, \text{“}printf\text{”}) \\
A \cap B &= (\text{“}double\text{”}, \text{“}for\text{”}) \\
d_J(A,B) &= \frac{|A \cup B| - |A \cap B|}{|A \cup B|} \\
d_J(A,B) &= \frac{5-2}{5} = \frac{3}{5} = 0.6
\end{aligned}
$$

Fonte: Produzido pelo autor.

Na Figura 5 exemplifica-se como é calculada a métrica de distância Levenshtein para entre as palavras “printf” e “println”. Observa-se que houve uma substituição da letra “F” para a letra “L”, e também foi necessário a inserção da letra “N”, resultando em um total de duas modificações necessárias para as duas palavras se tornarem iguais.

Figura 5 – Exemplo de cálculo de métrica de distância Levenshtein.



Fonte: Produzido pelo autor.

No caso, para comparar a distância entre códigos-fontes (original ou ofuscado) de programas, observa-se a quantidade de modificações necessárias para que os dois códigos-fonte fiquem iguais. Na prática, quanto menor for a distância Levenshtein, mais similares são os códigos-fontes e, quanto maior esse valor, mais distintos são os códigos-fontes comparados.

2.5.3.3 Normalised Compression Distance

Normalised Compression Distance (NCD)(CILIBRASI; VITÁNYI, 2005; FELDT et al., 2016), é uma métrica de distância dada pela Equação 2.4, na qual x e y são duas *strings*, $C(x)$ é o tamanho da *string* x comprimida, $C(y)$ é o tamanho da *string* y comprimida, e $C(xy)$ é o tamanho da concatenação das *strings* x e y após o processo de compressão. A função *min* retorna o menor tamanho entre as *strings* comprimidas. Por fim, a função *max* retorna o maior tamanho entre as *strings* comprimidas.

$$NCD(x,y) = \frac{C(xy) - min\{C(x), C(y)\}}{max\{C(x), C(y)\}} \tag{2.4}$$

O resultado da métrica de distância NCD é um valor entre 0 e 1, onde quanto mais próximo de 0, mais parecidas são as *strings* e, quanto mais próximo de 1, mais diferentes são as *strings*. Para este trabalho, escolheu-se utilizar o compressor bzip2[3] e cada uma dessas *strings* representou um código-fonte (original ou ofuscado) de um programa.

2.5.4 Algoritmos de Agrupamento

Também como forma de identificar os grupos de programas mais similares, para este trabalho utilizaram-se algoritmos de agrupamento (do inglês, *clustering*). *Clustering* tem como objetivo, dada uma entrada de várias instâncias, formar grupos de instâncias mais similares de forma automática. Na implementação do *framework* $SiMut$ e nos experimentos, cada instância foi representada como: (i) um conjunto de resultados de métricas internas computadas para cada programa e (ii) um conjunto de contagem de ocorrências de cada palavra presente no código-fonte de cada programa.

Para este trabalho, os algoritmos de *clustering* X-Means e Expectation Maximization foram empregados. Ambos são utilizados em aprendizagem de máquina (do inglês, *machine learning*) no contexto de aprendizado não supervisionado. Aprendizado de máquina não supervisionando, resumidamente, é aquele não necessita da interferência humana durante a sua execução, ou seja, o processo é realizado automaticamente após o inicio da execução. O X-Means é uma versão aprimorada de K-Means e, nas próximas subseções, apresentam-se esses algoritmos de *clustering*, citando-se os motivos pelas quais eles foram selecionados.

2.5.4.1 K-Means

O algoritmo de clustering K-Means é um método iterativo para particionar um conjunto de instâncias em uma quantidade específica K de grupos de instâncias similares. Esse particionamento ocorre através de alguns passos que são descritos a seguir:

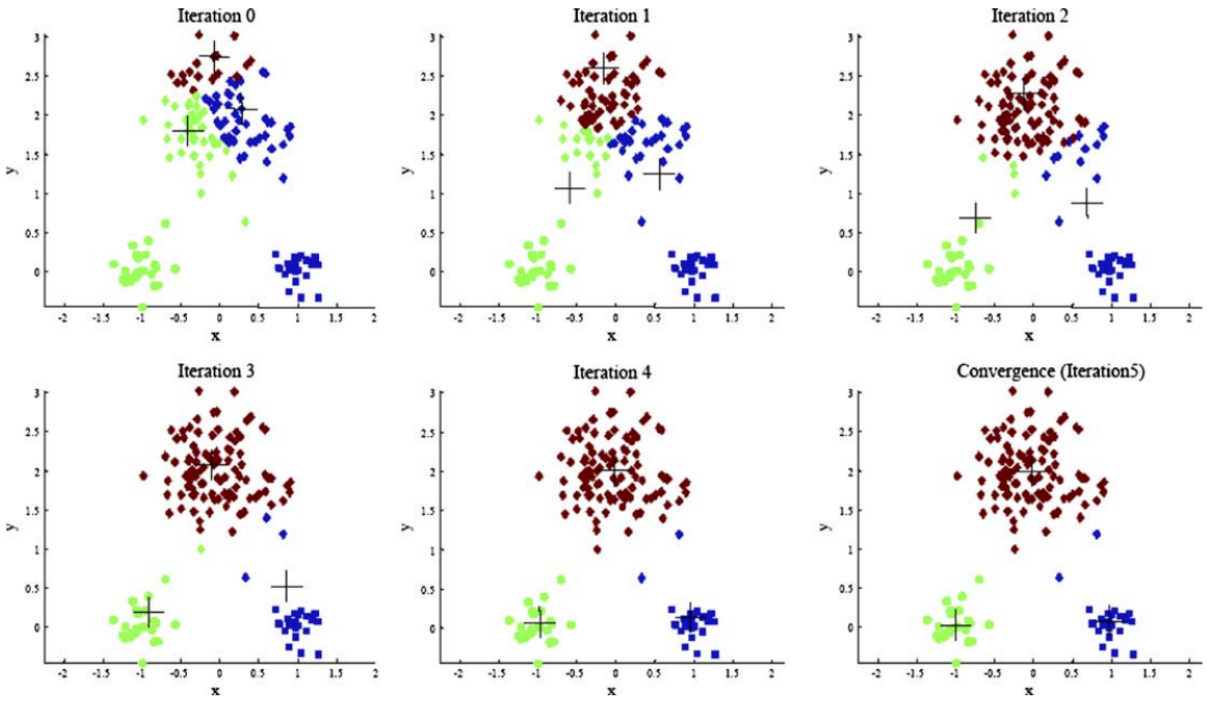
- Definição da quantidade K desejada, na qual K representa quantos *clusters* se deseja dividir as instâncias ao final da execução do algoritmo.
- Para cada *cluster* definido, é posicionado aleatoriamente um centroide em relação ao conjunto de instâncias. Esses centroides servem como referência para a formação dos *clusters* ao final da execução do algoritmo.
- Nesta etapa, cada instância é alocada no *cluster* com centroide mais próximo dela. Nas subseções subseção 2.5.4.2.1 e subseção 2.5.4.2.2 apresentam-se informações sobre as distâncias utilizadas neste trabalho para essa etapa.

[3] <http://www.bzip.org/> Acessado em Janeiro de 2022.

- Cada centroide é reposicionado no centro do *cluster* correspondente, com base nas instâncias alocadas no mesmo.
- As duas últimas etapas descritas acima são executadas de forma iterativa enquanto houver modificações de alocação de instância na execução. Caso não ocorra novas modificações, então o algoritmo de *clustering* K-Means é dado como concluído.

Na Figura 6 apresenta-se um exemplo do comportamento durante 6 iterações de uma execução do algoritmo de *clustering* K-Means. Neste exemplo, K vale 3 e cada + representa um centroide. Nota-se que a cada iteração, cada instância é alocada no *cluster* onde o centroide é mais próximo dela. Observa-se também que na última iteração (*Iteration 5*) ocorre a convergência, sinalizando que nenhuma instância sofreu modificação com relação à iteração anterior (*Iteration 4*).

Figura 6 – Exemplo de iterações do algoritmo K-Means.



Fonte: Wu et al. (2008)

Na próxima subseção apresenta-se uma versão melhorada do algoritmo K-Means, denominada X-Means, que traz algumas otimizações em relação ao K-Means.

2.5.4.2 X-Means

Embora o algoritmo de *clustering* K-Means seja amplamente utilizado, ele apresenta alguns problemas de alocação das instâncias, onde este alocamento pode não ser a ideal para cada instância. O primeiro problema é o posicionamento inicial do centroide, que é feito de forma aleatória. Isso ocorre devido ao fato de não existir algum padrão ou abordagem

para determinar o ponto central de cada *cluster*. Deste modo, a posição inicial estabelecida aleatoriamente pode não ser a ideal para a formação dos *clusters*. Outro problema presente no K-Means, como descrito por Pelleg e Moore (2000), é o algoritmo *clustering* ficar confinado a executar considerando desde o começo K *clusters* simultaneamente ao invés de ir adicionando novos *clusters* dinamicamente. Tal comportamento pode dificultar a escolha ideal de alocação das instâncias envolvidas na execução do algoritmo.

O algoritmo de *clustering* X-Means (PELLEG; MOORE, 2000) é uma variação do algoritmo K-Means que tenta suprir parte dos problemas supracitados.. Na prática, o algoritmo X-Means inicia a execução com a menor quantidade de *clusters* K com base em um intervalo determinado. O algoritmo X-Means adiciona novos centroides na medida em que eles são necessários, até chegar na quantidade máxima definida no intervalo. Durante toda a execução do algoritmo, o conjunto de centroides que atinge a melhor pontuação é registrado, e este é o que finalmente é utilizado. Durante esse processo, a cada variação de quantidade de K, duas operações são executadas: *melhorar os parâmetros* e *melhorar a estrutura*.

Melhorar os parâmetros consiste na execução do algoritmo de K-Means original até a convergência, ou seja, até quando não ocorrer nenhuma nova modificação de alocação de instâncias. Na operação *melhorar estrutura* o objetivo é se, e onde os novos centroides devem aparecer, esse processo ocorre dividindo-se os *clusters* em dois novos *subclusters* através do K-Means. Cada divisão efetuada é avaliada e as divisões que geraram as melhores partições são mantidas, de forma a não gerar mais que a quantidade máxima de grupos do intervalo definido. Em um cenário no qual nenhum *subcluster* é mantido, divide-se uma proporção dos *clusters* formados inicialmente, geralmente na metade, a partir das divisões que resultem nas melhores avaliações. Deste modo, mantém-se tanto os *clusters* bem distribuídos, quanto os novos possíveis *subclusters* com boa cobertura de instâncias não tão bem representadas pelo *cluster* inicial. Para avaliar a permanência ou não dos *clusters* e dos *subclusters* formados durante a execução do X-Means, utiliza-se o cálculo de *Bayesin Information Criterion* (BIC), da área da estatística, que indica o melhor modelo entre um conjunto limitado de modelos.

Nas duas próximas subseções descrevem-se as duas distâncias empregadas neste trabalho para a etapa de cálculo de distância entre as instâncias e os centroides de cada *cluster*.

2.5.4.2.1 Distância Euclidiana

A distância euclidiana, dada pela Equação 2.5, é calculada através da raiz quadrada da somatória das diferenças quadradas entre os vetores. Na equação, x e y representam dois vetores de valores e n representa a quantidade de valores (dimensões) existentes em cada vetor.

$$D_E(x, y) = \sqrt{\sum_{i=1}^{n}(x_i - y_i)^2} \tag{2.5}$$

Essa distância foi utilizada neste trabalho por ser a distância mais utilizada nos algoritmos de *clustering* K-Means e X-Means, inclusive sendo a distância padrão nas principais ferramentas que aplicam estes algoritmos, como WEKA[4] e scikit-learn[5].

2.5.4.2.2 Distância Manhattan

A distância de Manhattan, dada pela Equação 2.6, é calculada através da somatória das diferenças absolutas entre os dois vetores. Na equação, x e y representam dois vetores de valores e n representa a quantidade de valores (dimensões) existentes em cada vetor.

$$D_M(x, y) = \sum_{i=1}^{n}|x_i - y_i| \tag{2.6}$$

Essa distância foi utilizada pois, além de estar disponível na ferramenta utilizada neste trabalho (mais detalhes na subseção 3.4.1), desejou-se analisar, por empiricismo, se os resultados utilizando essa distância trariam algum resultado melhorado com relação à distância Euclidiana, amplamente utilizada em algoritmos de *clustering*.

2.5.4.3 Expectation Maximization

Expectation Maximization (EM), apresentado por Dempster, Laird e Rubin (1977), é um algoritmo de *clustering* que, através de um processo iterativo, permite a estimação de parâmetros em modelos probabilísticos com variáveis latentes (variáveis que não podem ser observadas diretamente, ou que são incompletas). Esse processo iterativo consiste de duas etapas: *Expectativa* (do inglês, *Expectation*) (E) e *Maximização* (do inglês, *Maximization*) (M).

De maneira geral, após o início da execução do algoritmo com uma estimativa de parâmetro inicial, o algoritmo executa de forma iterativa duas etapas até ocorrer uma convergência. Na etapa *Expectation*, utilizando-se as variáveis latentes, estimam-se os dados que faltam para completar a mostra de dados incompleta. Na etapa *Maximization*, por sua vez, aplica-se um algoritmo com os dados estimados na etapa *Expectation*, de modo que esses novos parâmetros substituem os parâmetros anteriores para próxima execução.

Essas duas etapas continuam em execução até ocorrer a convergência. Neste momento, a execução do algoritmo é então concluída e como resultado identificam-se os *clusters* com os elementos mais similares.

[4] <https://www.cs.waikato.ac.nz/ml/weka/> - Acessado em Janeiro de 2022.
[5] <https://scikit-learn.org/> - Acessado em Janeiro de 2022.

2.6 Considerações Finais

Neste capítulo apresentou-se a fundamentação teórica para este trabalho. De maneira geral, discutiram-se assuntos básicos de teste de software, focando no teste de mutação. Também apresentaram-se os conceitos do *framework SiMut*, citando a sua proposta e a descrição das etapas envolvidas em seu funcionamento. Por fim, também apresentaram-se as abordagens de abstrações e similaridades de programas, citando as técnicas envolvidas neste trabalho. No próximo capítulo apresentam-se os detalhes da implementação do *framework SiMut*.

3 Implementação

3.1 Considerações Iniciais

Para a realização dos experimentos, foi implementado o *framework SiMut*, idealizado por Pizzoleto et al. (2020). Neste capítulo apresenta-se como o *framework* foi implementado, descrevendo cada ferramenta acoplada e o seu funcionamento como um todo. O restante deste capítulo está organizado da seguinte forma: na seção 3.2 apresentam-se as características gerais da implementação, descrevendo as linguagens de programação utilizadas no *framework* e como ele foi arquitetado; na seção 3.3 apresentam-se as ferramentas que foram acopladas ao *framework* para computar abstrações de programas; na seção 3.4 apresentam-se as ferramentas que foram inseridas no *framework* para a etapa de similaridade; na seção 3.5 discutem-se as combinações possíveis com base nas ferramentas de abstrações e similaridades implementadas no *framework*; na seção 3.6 apresenta-se e descreve o padrão do arquivo de execução para o *framework*; por fim, na seção 3.7 apresenta-se como a técnica de redução de custo, utilizando a abordagem inspirada na Mutação Seletiva (OFFUTT; ROTHERMEL; ZAPF, 1993), é aplicada.

3.2 Características Gerais da Implementação

O *SiMut* foi desenvolvido, em sua maior parte, em Java, com o objetivo de funcionar em diferentes sistemas operacionais. Como a proposta do *framework* é envolver diferentes ferramentas e possibilitar futuras evoluções, o gerenciador de projeto Maven[1] foi escolhido, facilitando assim a integração do projeto com diferentes dependências. A versão implementada aceita neste primeiro momento somente programas escritos na linguagem de programação Java.

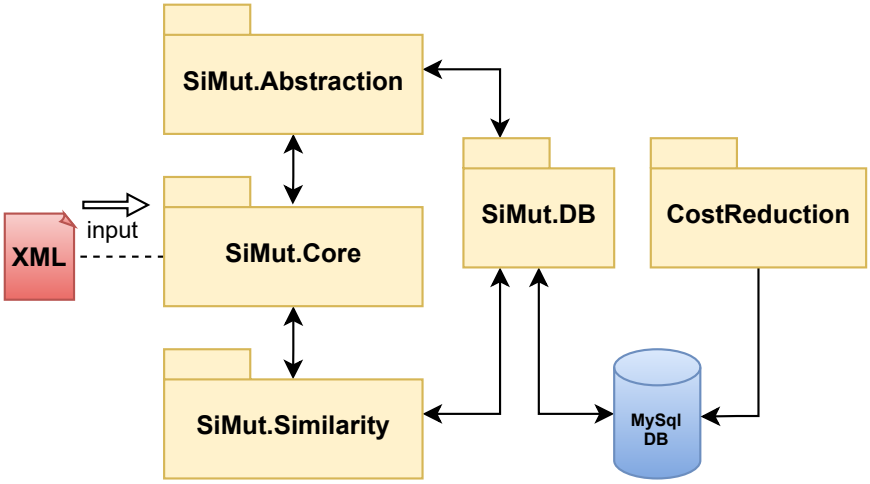
Para executar o *framework SiMut*, é necessário indicar por parâmetro o caminho do arquivo de execução. Esse arquivo é em formato XML *(eXtensible Markup Language)* e, de maneira geral, nele é necessário informar o grupo de classes testadas T que serão usadas para compor o grupo de referência G, a classe não testada u, a abstração A, e a abordagem de similaridade S desejada. Também é possível indicar a quantidade de *clusters* que o usuário quer como saída. Mais informações sobre o arquivo de execução são apresentadas na seção 3.6.

Ao final da execução, o grupo de referência G fica armazenado em um banco de dados relacional MySQL. Além disso, como forma de otimizar outras execuções, todos os

[1] <https://maven.apache.org/index.html> - acessado em Janeiro de 2022

programas e resultados de abstrações são armazenados; deste modo, o *SiMut* não processa novamente abstrações de classes já utilizadas em execuções anteriores. Depois da formação do grupo de referência G, um módulo desenvolvido em C# por um outro pesquisador envolvido no *SiMut* computa a redução de custo. Essa etapa é descrita em detalhes na seção 3.7. A Figura 7 apresenta a arquitetura do *framework*.

Figura 7 – Arquitetura do *SiMut*.



Fonte: Guarnieri, Pizzoleto e Ferrari (2022)

O módulo SiMut.Core contém as classes fundamentais para todo o processo do SiMut. Este módulo é responsável por receber e interpretar o XML de execução, identificando o grupo de classes testadas T selecionadas, a classe não testada u, a técnica de abstração que será aplicada, a técnica de similaridade que será utilizada e a quantidade de clusters escolhida.

O módulo SiMut.Abstraction possui as classes responsáveis por integrar cada abstração acoplada ao *framework*. De acordo com a abstração selecionada, a classe correspondente verifica na base de dados, por meio de classes específicas do módulo `SiMut.DB`, se já existem as abstrações das classes selecionadas. Caso existam abstrações processadas para as classes, então os resultados dessas abstrações são resgatados do banco de dados. Caso não seja encontrado abstração para alguma classe, então a abstração é computada para a classe e o resultado é armazenado na base dados para futuras execuções.

O módulo SiMut.Similarity possui as classes responsáveis por integrar cada abordagem de similaridade. Todos os resultados gerados na etapa de similaridade, também por meio de classes específicas do módulo SiMut.DB, são armazenados no banco de dados. Com isso, é possível visualizar todos os resultados das execuções posteriormente.

O módulo CostReduction, utilizando como entrada os grupos de referência G armazenados no banco de dados, é responsável por aplicar a redução de custo. Essa redução de custo ocorre por meio da identificação do escore de mutação de cada operador

para todos os grupos de programas envolvidos na execução. Mais informações sobre esse módulo são apresentadas na seção 3.7.

Na Figura 8 apresenta-se o modelo físico do banco de dados utilizado pelo *framework* *SiMut*, a qual contém a estrutura das tabelas envolvidas no armazenamento dos programas, das classes, das abstrações computadas e dos grupos de programas mais similares formados. O armazenamento dos programas e das suas respectivas classes é feito por meio das tabelas program e class. A tabela abstraction_result é responsável por armazenar os resultados de abstrações computadas para classes. As tabelas similarity_cluster_result, similarity_cluster e similarity_result são responsáveis por armazenarem as formações dos grupos de programas mais similares. As demais tabelas servem de apoio para estes e demais processos executados pelo *framework* *SiMut*.

Todo as informações de execução são registradas por meio do utilitário Log4j 2[2], que está sob *Licença Apache 2.0*. Caso exista alguma inconsistência no arquivo de execução e isso resulte em alguma falha durante a execução do *framework*, ou por qualquer outro motivo, é possível checar o que ocasionou a falha de execução através do arquivo de *log* gerado pelo utilitário.

Algumas das abstrações acopladas ao *framework* utilizam o *bytecode* da aplicação como entrada. Com isso, para utilizar um programa no *SiMut* é um pré-requisito que todos os programas envolvidos na execução estejam em formato Maven. Deste modo, o *SiMut* compila em tempo de execução todos os programas selecionados no arquivo de execução e gera o *bytecode*, caso necessário, para realizar as abstrações.

Na Tabela 1 lista-se para cada pacote do projeto a quantidade de classes e linhas de código. A última linha da tabela representa a soma dos valores de cada coluna, onde é observado que o desenvolvimento do *framework* envolveu no total 67 classes e 6.231 linhas de código-fonte. Todos os módulos listados nessa tabela foram desenvolvidos inteiramente neste trabalho, objetivando-se a flexibilização de adição de novas ferramentas de abstrações e similaridades. Vale notar que essa contagem não considera o módulo CostReduction, visto que este foi desenvolvido por outro membro do grupo de pesquisa.

Tabela 1 – Métricas do framework desenvolvido.

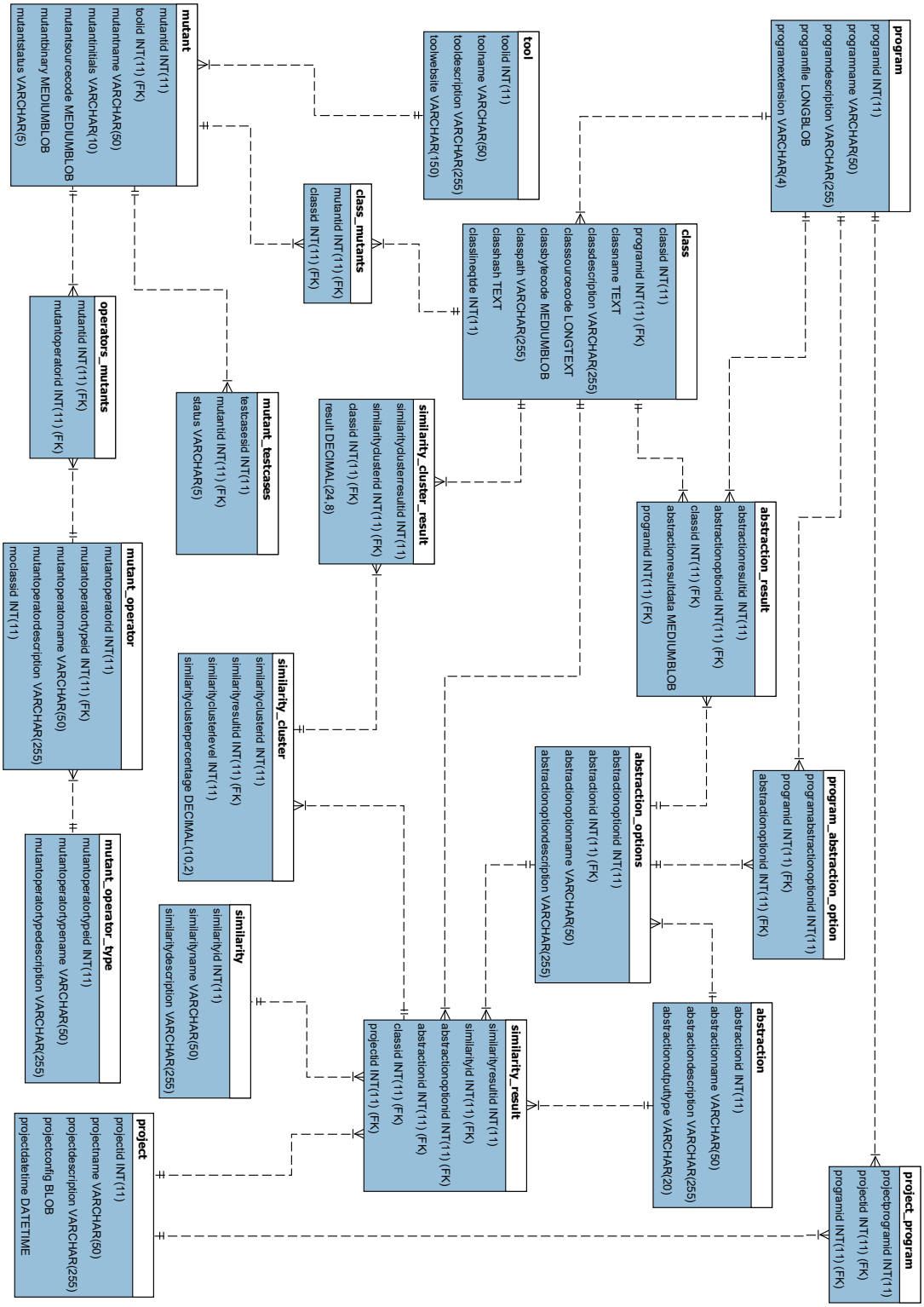
Pacote	**# Classes**	**# Linhas de Código**
simut.core	24	2.001
simut.abstraction	11	1.032
simut.similarity	15	1.486
simut.db	17	1.712
Total	**67**	**6.231**

Fonte: Produzido pelo autor.

Na Figura 3.2 apresenta-se a linha de comando necessária para a execução de

[2] <https://logging.apache.org/log4j/2.x/index.html> - acessado em Dezembro de 2021.

Figura 8 – Modelagem do banco de dados utilizado pelo *framework SiMut*.



Fonte: Produzido pelo autor.

alguma configuração utilizando-se o *SiMut*. Para executar o *SiMut* é necessário indicar o endereço do arquivo de execução. Para a execução correta do *framework*, o arquivo precisa estar em formato XML e ajustado para a forma como o *framework* interpreta as informações.

Figura 9 – Linha de comando para executar o *framework SiMut*.

```
java -jar  .\SiMut.jar <arquivo de execução XML>
```

Fonte: Produzido pelo autor.

Para executar o *framework SiMut*, é necessário que exista um arquivo `simut.config` no mesmo diretório do executável, ou seja, no mesmo endereço que o executável do *SiMut* está localizado. Na Figura 3.2 ilustra-se esse arquivo, o qual contém definições de informações básicas para o funcionamento do *framework*: tamanho máximo em *megabyte* que cada programa selecionado pode ter, informações sobre a conexão com o banco de dados MySQL e também o endereço de onde o Maven está instalado. Este último é necessário para que o *framework* consiga compilar os programas em tempo de execução, utilizando-se o próprio Maven.

Figura 10 – Arquivo de configuração do *SiMut*.

```
#Tamanho máximo (em MB) de cada programa
MAX_FILE_SIZE = 32
#Dados de conexão com o banco de dados MySQL
DB_HOST = 127.0.0.1
DB_NAME = simut
DB_USERNAME = root
DB_PASSWORD = ppgccs_2022
#Diretório
MAVEN_DIR = C:/apache-maven-3.6.3
```

Fonte: Produzido pelo autor.

3.3 Ferramentas para Computar Abstrações

No módulo *SiMut.Abstraction* foram acopladas duas ferramentas capazes de computar abstrações de programas: CKJM extended[3], uma ferramenta que computa métricas internas de programas, e ProGuard[4], um ofuscador de código-fonte. A escolha dessas duas ferramentas se deu pela facilidade de aplicá-las no *framework* implementado, tendo em

[3] <http://gromit.iiar.pwr.wroc.pl/p_inf/ckjm/> - acessado em Janeiro de 2022.
[4] <https://www.guardsquare.com/proguard> - acessado em Janeiro de 2022.

vista que elas também foram desenvolvidas em Java, e também por não estarem obsoletas. Cada abstração computada é armazenada no banco de dados, podendo na sequência ser utilizada na etapa de similaridade do *SiMut*. No arquivo de execução XML o usuário pode indicar qual ferramenta de abstração deverá ser utilizada na execução. Mais detalhes dessas duas ferramentas acopladas são apresentadas nas próximas subseções.

3.3.1 Ferramenta CKJM Extended

CKJM extended é uma versão extendida da versão original da CKJM proposta por Jureczko e Spinellis (2010). Na prática, ela é uma ferramenta capaz de computar 19 métricas internas utilizando-se o *bytecode* de classes escritas em Java. A ferramenta computa métricas apresentadas por diferentes autores, como as métricas de orientação a objetos de Chidamber e Kemerer (1994), as métricas de *Quality Model for Object-Oriented Design* (QMOOD) propostas por Bansiya e Davis (2002), e também a métrica de complexidade ciclomática proposta por McCabe (1976).

No arquivo de execução do *SiMut*, é possível escolher quais métricas deverão ser computadas para os programas selecionados. A Tabela 2[5] apresenta todas as métricas computadas pela ferramenta CKJM extended e que podem ser utilizadas pelo *SiMut*.

Tabela 2 – Lista de métricas computadas pela CKJM extended.

Sigla	**Métrica**
WMC	Weighted methods per class
DIT	Depth of Inheritance Tree
NOC	Number of Children
CBO	Coupling between object classes
RFC	Response for a Class
LCOM	Lack of cohesion in methods
Ca	Afferent coupling
Ce	Efferent coupling
NPM	Number of Public Methods for a class
LCOM3	Lack of cohesion in methods Henderson-Sellers version
LOC	Lines of Code
DAM	Data Access Metric
MOA	Measure of Aggregation
MFA	Measure of Functional Abstraction
CAM	Cohesion Among Methods of Class
IC	Inheritance Coupling
CBM	Coupling Between Methods
AMC	Average Method Complexity
CC	McCabe's Cyclomatic Complexity

Fonte: Tabela elaborada pelo autor com base nas informações disponíveis na página oficial da CKJM extended <http://gromit.iiar.pwr.wroc.pl/p_inf/ckjm/metric.html> - Acessado em Janeiro de 2022.

[5] Para manter a consistência com a ferramenta e outros trabalhos que também abordaram métricas internas, manteve-se o nome de cada métrica na língua inglesa.

Para este trabalho, foi acoplada a versão 2.2 da CKJM extended, a qual está disponível sob a licença *Atribuição - Não Comercial - Sem Derivações 2.5*. Essa integração aconteceu por meio da adição da dependência da ferramenta ao *SiMut*. Com isso, utilizando as classes do pacote `gr.spinellis.ckjm`, o *SiMut* indica para a ferramenta o .JAR (Java Archive), ou seja, o executável Java do programa que deseja-se computar as métricas e na sequência a ferramenta retorna com os resultados computados para cada classe presente no .JAR.

3.3.2 Ferramenta ProGuard

ProGuard é uma ferramenta de código-fonte aberto que otimiza, encurta e ofusca o *bytecode* de aplicações desenvolvidas em Java. Com isso, é reduzido consideravelmente o tamanho das classes, removendo partes redundantes ou trechos de código-fonte não utilizados. Para este trabalho, foi utilizado a ProGuard na versão 6.2.2, que está disponível sob a *Licença Pública Geral GNU v2*.

Para acoplar essa ferramenta ao *SiMut*, foi adicionada a dependência dela ao *framework*. Com isso, utilizando-se as classes do pacote `proguard`, tornou-se possível ofuscar arquivos .JAR diretamente pelo *SiMut*. O *SiMut* possibilita escolher quais ações devem ser executadas por meio da ProGuard; no caso, é possível escolher até três das seguintes opções: ofuscamento, otimização e encurtamento. Ao final da execução da ProGuard, é gerado um novo arquivo .JAR ofuscado. A partir disso, *SiMut* extrai os arquivos desse .JAR e decompila, por meio da ferramenta Java Decompiler[6], todos os arquivos .class (ou seja, os *bytecodes* das classes compiladas pelo compilador Java) encontrados. Para este trabalho, foi utilizada a versão 1.1.3 da Java Decompiler que está disponível sob a *Licença Pública Geral GNU v3*.

A ProGuard utiliza um arquivo de configuração para ser executada. Esse arquivo é gerado em tempo de execução utilizando-se o *SiMut*, no qual se definem quais técnicas serão aplicadas, seguindo o que foi inserido no arquivo de execução do *framework*. De maneira geral, a ProGuard pode aplicar três ações supracitadas em cada *bytecode*, que são descritas a seguir.

- *obfuscate* (ofuscar): esse recurso reduz a quantidade de caracteres dos nomes de classes, variáveis e métodos. As opções de encurtamento de nome de classe e nome de método executadas pela ProGuard foram removidas; isso foi necessário para que a classe processada não perdesse a relação com a classe original, pois, como descrito na seção 3.2, o *SiMut* armazena todas as abstrações computadas para utilizar na etapa de similaridade.
- *shrink* (encolher): esse recurso encolhe o *bytecode* removendo variáveis ou métodos que não estão sendo utilizados pelo programa;

[6] <http://java-decompiler.github.io/> - acessado em Janeiro de 2022.

- *optimize* (otimizar): esse recurso otimiza o *bytecode*, podendo remover alguns trechos, como estrutura condicional ou de repetição que logicamente nunca serão executados pelo programa.

3.4 Ferramentas para Similaridade

Depois de computar as abstrações, o *SiMut* realiza o processo de formação do grupo de referência G. Para essa etapa foram utilizadas as seguintes ferramentas de similaridade: Weka(*Waikato Environment for Knowledge Analysis*) (WITTEN; FRANK, 2002), mdist[7] e JPlag.[8]. Até o momento da implementação do framework *SiMut* descrito nesta dissertação, essas ferramentas continuavam recebendo atualizações e, por serem de fácil implementação, foram selecionadas para o módulo de similaridade. As subseções seguintes descrevem cada uma dessas ferramentas, explicando a versão utilizada e alguns detalhes fundamentais de implementação.

3.4.1 Ferramenta Weka

Weka é uma biblioteca de código-fonte aberto que inclui algoritmos de aprendizado de máquina para tarefas de mineração de dados. Com isso, ela fornece recursos para classificação automática, regressão e agrupamento (do inglês, *clustering*). Através da Weka, o *SiMut* permite o agrupamento de classes usando os algoritmos X-Means (PELLEG; MOORE, 2000) ou *Expectation Maximization* (EM) (DEMPSTER; LAIRD; RUBIN, 1977). Ambos são algoritmos de aprendizado de máquina não supervisionado que geram K *clusters* de um conjunto de dados.

Por ser escrita em Java, o acoplamento da Weka no *SiMut* foi realizada através da adição da dependência dela ao *framework*. Com isso, através das classes do pacote `weka`, o *SiMut* executa a etapa de similaridade, formando os grupos de classes mais similares. Para este trabalho, foi utilizada a versão 6.8 da Weka que está sob a *Licença Pública Geral GNU v2*.

Para realizar o processo de clustering, a Weka utiliza como entrada um arquivo em formato ARFF (*Attribute-Relation File Format*). Esse arquivo é dividido em duas partes: informações de cabeçalho e os dados. Na prática, ele descreve uma lista de instâncias que compartilham de um mesmo conjunto de atributos. Para realizar a etapa de similaridade, o *SiMut* gera o arquivo ARFF em tempo de execução. Esse arquivo ARFF pode ser visualizado pelo testador ao final da execução do *framework SiMut*.

Na Figura 11 apresenta-se um ARFF gerado automaticamente pelo *SiMut*, na qual é possível identificar cinco atributos, sendo o primeiro do tipo texto e os demais do

[7] <https://hub.docker.com/r/robertfeldt/mdist> - Acessado em Janeiro de 2022.
[8] <https://jplag.ipd.kit.edu/> - Acessado em Janeiro de 2022.

tipo numérico. Após a linha @*data*, são listadas todas as instâncias (as classes envolvidas na execução), que são utilizadas no agrupamento, onde cada instância fica em uma linha. Ao gerar o arquivo ARFF, para depois identificar os resultados de cada classe envolvida na execução do *framework* $SiMut$, o valor do primeiro atributo é composto por três valores separados pelo caractere *underline*: código do programa, código da classe e nome da classe. Os valores dos atributos de cada instância são separados por vírgula e segue a ordem em que foi definido cada atributo. No exemplo da Figura 11, a classe *Bisect* pertence ao programa código 1 e o seu código também é 1, os atributos *cbo* e rfc têm respectivamente os seguintes valores: 3.0 e 5.0. Já os atributos *dit* e *noc* não pontuaram e ficaram com resultado igual a zero.

Figura 11 – Exemplo de ARFF gerado pelo *framework*.

```
@relation CKJM_EXTENDED_ProjectID_21_1_4_cal
@attribute className string
@attribute dit numeric
@attribute noc numeric
@attribute cbo numeric
@attribute rfc numeric
@data
1_1_Bisect,0.0,0.0,3.0,5.0
1_2_BoundedQueue,0.0,0.0,7.0,14.0
1_3_BubCorrecto,0.0,0.0,5.0,16.0
1_4_cal,0.0,0.0,13.0,15.0
1_5_Calculation,0.0,0.0,8.0,20.0
1_6_checkIt,0.0,0.0,6.0,6.0
1_7_CheckPalindrome,0.0,0.0,8.0,13.0
1_8_countPositive,0.0,0.0,8.0,10.0
1_9_DigitReverser,0.0,0.0,7.0,12.0
1_10_Find,0.0,0.0,2.0,5.0
```

Fonte: Produzido pelo autor.

Utilizando-se o arquivo de execução do $SiMut$, é possível escolher qual distância deverá ser adotada pelo algoritmo de agrupamento X-Means. Foram definidas duas opções: distância Euclidiana e distância de Manhattan. A escolha dessas duas opções se deu pelo fato de estarem, por padrão, disponíveis na biblioteca Weka.

Caso a Weka utilize como entrada uma abstração que gerou um resultado em formato textual, por meio do $SiMut$, a Weka aplica o processo de *String To Word Vector* (STWV), que converte esse código-fonte em um conjunto de atributos numéricos que representam a quantidade de ocorrência de cada palavra presente no código-fonte.

3.4.2 Ferramenta mdist

Capaz de calcular distâncias e similaridades entre arquivos, mdist é uma ferramenta que implementa MultiDistances.jl,[9] uma biblioteca desenvolvida em linguagem Julia. Para

[9] <https://github.com/robertfeldt/MultiDistances.jl> - Acessado em Janeiro de 2022.

utilizar a mdist no *SiMut*, foi necessário integrar o Docker,[10] que é uma ferramenta de conteinerização de código aberto. Deste modo, quando o *SiMut* utiliza recursos da mdist, automaticamente por meio do Docker é verificado se a versão mais recente da mdist está disponível no computador; caso não esteja, automaticamente uma imagem da versão mais recente da mdist é baixada e utilizada pelo *SiMut*.

Existem duas formas para executar a mdist. A primeira forma é passando por parâmetro o algoritmo de distância desejado e o caminho de dois arquivos para serem comparados. A segunda forma, que foi a implementada no *SiMut*, é indicando o algoritmo de distância desejado e o endereço da pasta que contém todos os arquivos de código-fonte que serão comparados. Nessa última forma, ao final da execução, um arquivo de formato CSV é gerado contendo uma matriz com todos os resultados de distância entre todos os arquivos da pasta indicada.

No arquivo de execução do *SiMut* é possível indicar o algoritmo de distância desejado e, após a execução da mdist, o *SiMut* automaticamente lê o arquivo CSV gerado e identifica o grupo de programas mais similares. Ao final da execução, é possível acessar o arquivo de resultados gerado pela mdist. Na Figura 3.4.2 é apresentado o comando utilizado para executar a mdist. No parâmetro "output" é indicado o endereço do diretório que receberá o CSV com os resultados, no parâmetro "-d" representa-se o algoritmo de distância que deverá ser aplicado, e por fim em "folder" é indicado o endereço da pasta que contém os arquivos que serão comparados.

Figura 12 – Linha de comando para executar a ferramenta mdist.

```
docker run -it -v "ouput":/data robertfeldt/mdist mdist -d jaccard
distances "folder"
```

Fonte: Produzido pelo autor com base nas informações disponíveis no site <https://hub.docker.com/r/robertfeldt/mdist> - acessado em Janeiro de 2022.

Para este trabalho foi acoplada a mdist na versão 0.1.7 que está disponível sob a *Licença MIT*. Entre os diversos algoritmos de distância disponíveis na mdist, para as execuções realizadas neste trabalho foram utilizados os seguintes algoritmos: Levenshtein (LEVENSHTEIN et al., 1966), Jaccard (JACCARD, 1901) e *Normalised Compression Distance* (NCD) (FELDT et al., 2008; FELDT et al., 2016).

3.4.3 Ferramenta JPlag

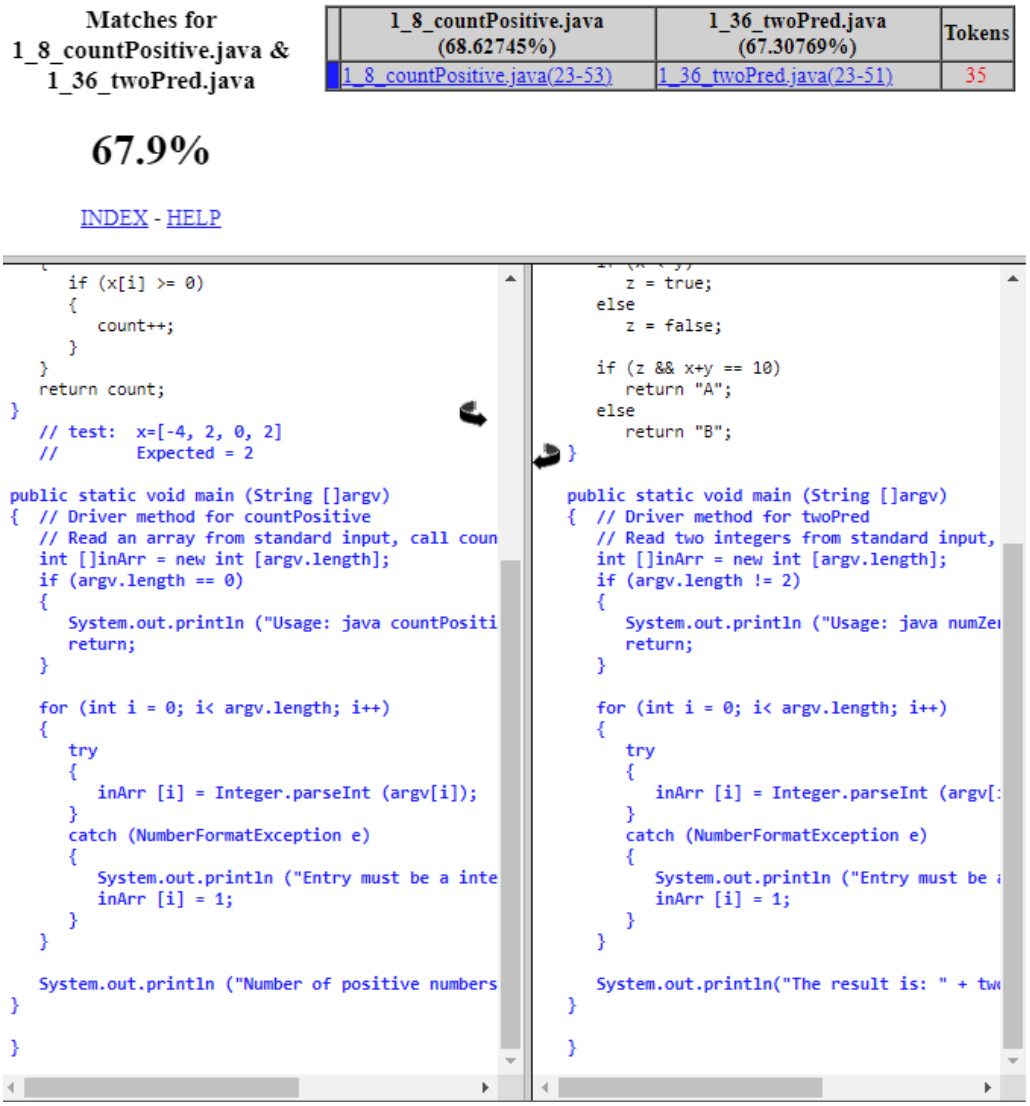
Também como forma de identificar similaridade entre programas, foi utilizada a JPlag, uma ferramenta capaz de detectar plágio entre códigos-fontes. Essa ferramenta não

[10] <https://www.docker.com/> - Acessado em Janeiro de 2022.

apenas compara o texto do código, mas como também compara a sintaxe da linguagem de programação dos arquivos selecionados. Desta forma, seja o mesmo código ou não, se a lógica é a mesma, a JPlag é capaz de identificar similaridade (PRECHELT et al., 2002).

No *SiMut*, foi acoplada a ferramenta JPlag na versão 2.12.1 que está disponível sob a *Licença Pública Geral GNU v3*. Por ser uma ferramenta desenvolvida em Java, para acoplá-la no *SiMut*, foi necessário adicionar sua dependência ao projeto. Assim, utilizando as classes do pacote `jplag`, o *SiMut* indica o endereço da pasta que contém todos os programas que serão comparados na execução planejada e na sequência é executado o processo de detecção de plágio. Ao final da execução, a JPlag gera um arquivo em formato *CSV* contendo uma matriz de comparação entre todos os arquivos. Além disso, são também gerados vários arquivos em formato HTML (*HyperText Markup Language*) mostrando todas as similaridades encontradas entre os arquivos da pasta indicada.

Figura 13 – Exemplo de comparação gerado pela JPlag entre códigos Java.



Fonte: Produzido pelo autor.

Na Figura 13 apresenta-se um exemplo de um desses arquivos HTML gerados pela JPlag. Neste exemplo, são apresentadas as similaridades identificadas entre duas classes: *countPositive* e *twoPred*. É possível notar que foi identificado uma similaridade de 67,9% entre elas. Na cor azul, são destacados os trechos de código similares entre os códigos. Para possíveis análises futuras pelo testador, todos os arquivos gerados durante a execução da JPlag, utilizando-se o *SiMut*, ficam disponíveis para acesso ao término da execução do *framework SiMut*.

Após a execução da JPlag, o *SiMut* lê os dados contidos no CSV e então são formados os grupos de programas mais similares. Na sequência, esses grupos de programas similares são armazenados no banco de dados para que então possam ser utilizados na etapa de redução de custo.

3.5 Combinações Possíveis

O *framework* implementado, por meio das ferramentas citadas nas seções anteriores, possibilita diversas combinações de execuções. Na Figura 14 apresentam-se as possibilidades de combinações do *SiMut*. Caso sejam utilizadas métricas internas computadas pela CKJM extended, então é possível, na etapa de similaridade, aplicar os algoritmos de *clustering* disponíveis na biblioteca Weka. No caso de utilizar um código processado pela ProGuard, além dos algoritmos de *clustering* da Weka, também é possível formar grupos de programas mais similares usando as funções de distância da mdist ou então JPlag.

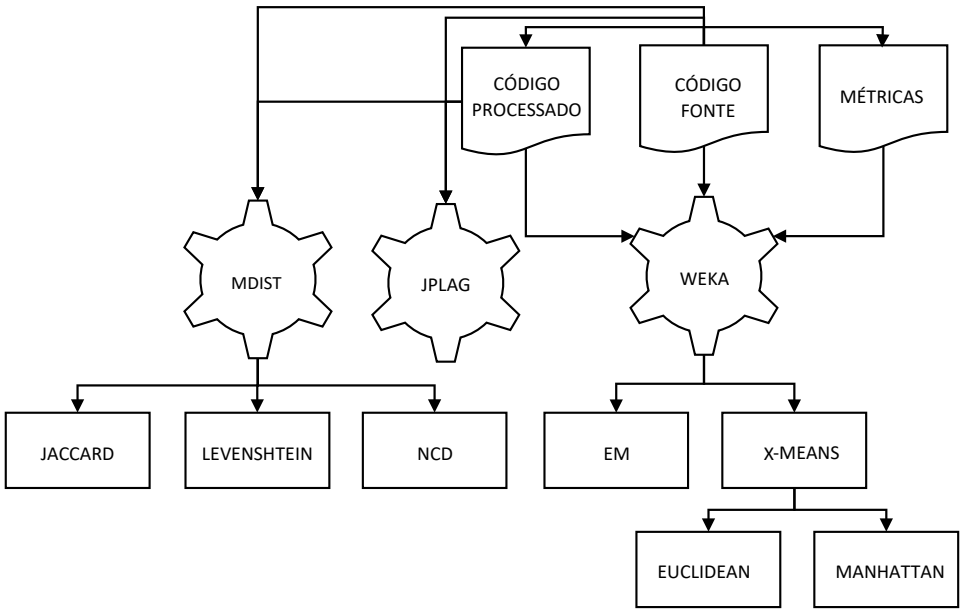
Para comparar a eficácia da configuração que aplica uma forma de representação de código-fonte na etapa de abstração, como métricas internas ou código ofuscado, também foi implementado no *SiMut* a opção de se utilizar o código-fonte original; neste caso, as mesmas técnicas de similaridades que são aplicadas a um código processado também podem ser aplicadas ao código original.

3.6 Arquivo de Execução

Para executar todo o processo do *SiMut*, é necessário indicar por parâmetro um arquivo de execução em formato XML. A Figura 15 apresenta um exemplo desse arquivo. Dentro do elemento *programs* são definidos os nomes e endereços dos programas já testados e do programa ainda não testado. Como algumas ferramentas de abstração utilizam o *bytecode* como entrada, todos os programas envolvidos na execução planejada precisam estar no padrão Maven. Deste modo, o *framework* compila, caso necessário, todos os programas, gerando assim o *bytecode* em tempo de execução.

Em *abstraction* são definidas as abstrações que o testador deseja utilizar na execução planejada. De acordo com a abstração selecionada, é possível definir algumas opções, como

Figura 14 – Árvore de possibilidades.



Fonte: Versão adaptada de Guarnieri, Pizzoleto e Ferrari (2022)

no exemplo da Figura 15, em que foram selecionadas cinco métricas internas utilizando-se a ferramenta CKJM extended. Na Tabela 2, na subseção 3.3.1, listam-se todas as métricas computadas pela CKJM extended e que podem ser selecionadas no arquivo de execução. Caso seja selecionado a ProGuard na etapa de abstração, então é possível selecionar até três das seguintes opções: *obfuscate*, *shrink* e *optimize*. Estas três opções oferecidas pela ProGuard são descritos na subseção 3.3.2.

Em *similarities* é necessário indicar o atributo *groups*, na qual é definida a quantidade desejada de grupos ao final da execução. Em *sources* são indicados quais abstrações serão utilizadas na etapa de similaridade; no exemplo, é indicado que será utilizado CKJM extended. Em *nprograms* são indicados os programas e as classes que ainda não foram testadas com o teste de mutação para os quais se deseja computar os parâmetros de redução de custo (por exemplo, operadores de mutação com maiores escores de mutação). O programa não testado também precisa ser indicado no elemento *programs*, como no exemplo. Ao final do arquivo de execução, é necessário indicar o nome da ferramenta que será utilizada para realizar o processo de similaridade. O nome da ferramenta é definido no atributo *name* e o algoritmo que deverá ser utilizado pela ferramenta precisa ser definido no atributo *algorithm*. Dentro desse elemento é então indicado o *level* de cada grupo, no qual quanto menor o *level*, maior será a similaridade entre os programas já testados e o programa ainda não testado. É necessário que a quantidade de grupos definidos dentro de *similarity* seja exatamente igual ao atributo *groups* do elemento *similarities*.

Além do atributo *level*, de acordo com a ferramenta de similaridade selecionada, é obrigatório indicar o percentual desejado de elementos de cada grupo. Caso seja utilizado a

Figura 15 – Exemplo de arquivo de execução.

```
<?xml version="1.0" encoding="UTF-8" standalone="no"?>
<project description="Example" name="Example">
    <programs>
        <program name="testedProgram1" path="c:/testedProgram1" />
        <program name="testedProgram2" path="c:/testedProgram2" />
        <program name="untestedProgram" path="c:/untestedProgram" />
    </programs>
    <abstractions>
        <abstraction name="CKJM-Extended">
            <options>
                <option name="dit"/>
                <option name="noc"/>
                <option name="cbo"/>
                <option name="rfc"/>
                <option name="lcom"/>
            </options>
        </abstraction>
    </abstractions>
    <similarities groups="3">
        <sources>
            <source name="CKJM-Extended"/>
        </sources>
        <nprograms>
            <nprogram name="untestedProgram">
                <class name="untestedClass"/>
            </nprogram>
        </nprograms>
        <similarity algorithm="xmeans" name="weka">
            <group level="0"/>
            <group level="1"/>
            <group level="2"/>
        </similarity>
    </similarities>
</project>
```

Fonte: Guarnieri, Pizzoleto e Ferrari (2022)

mdist ou JPlag, é obrigatório definir o percentual de classes que cada *cluster* deverá ter. No entanto, se algum algoritmo de *clustering* da Weka for selecionado, não é possível definir a porcentagem, pois o próprio algoritmo de *clustering* formará os *clusters*. Nesse caso, através do atributo *recluster* do elemento *similarity*, é possível definir um percentual máximo de classes que cada *cluster* pode ter; com isso, se o *cluster* o ultrapassar, um *recluster* é realizado automaticamente apenas uma vez, gerando um número final de *clusters* dado pela fórmula $(N * 2) - 1$, sendo N o número de grupos definidos no arquivo de execução XML. É importante destacar que o *reclustering* é realizado uma única vez; portanto um novo *recluster* não será executado caso algum *cluster* continue com uma quantidade elevada de classes após o *reclustering*. Na prática, caso seja utilizada mdist ou JPlag para a etapa de similaridade, é possível definir a quantidade de grupos e quantos programas ficarão em cada grupo. Quando a Weka é utilizada, é somente possível definir a quantidade de grupos, com a opção de executar *reclustering* uma única vez caso algum grupo exceda o percentual máximo de classes definido.

Na Figura 17 apresentam-se todas as configurações possíveis do *framework SiMut*.

Nota-se que, de acordo com a abstração selecionada, é necessário indicar as opções desejadas como, por exemplo, na abstração CKJM extended, em que se faz necessário indicar as métricas que deverão ser computadas. Com relação à etapa de similaridade, com exceção da ferramenta Weka, em todas as ferramentas é necessário indicar o percentual de classes que cada *cluster* deverá conter.

3.7 Aplicação da Técnica de Redução de Custo

Após a formação dos grupos de programas mais similares, a próxima etapa é a aplicação da técnica de redução de custo da mutação. Esse módulo, que foi implementado em C# por um colaborador do grupo de pesquisa, é responsável por computar os processos relacionados à redução de custo do teste de mutação. Atualmente, a abordagem utilizada para a redução de custo é inspirada na Mutação Seletiva (OFFUTT; ROTHERMEL; ZAPF, 1993). Em específico, neste trabalho ranqueiam-se os operadores de mutação e selecionam-se somente os operadores com o maior escore de mutação.

Para obter os escores de mutação, foi utilizado o banco de dados desenvolvido por Pizzoleto, Guarnieri e Ferrari (2021). Parte dos artefatos presentes nessa base de dados foram utilizados nos experimentos deste trabalho. Utilizando esse banco de dados, o módulo CostReduction dinamicamente cria a *killing matrix* que abrange mutantes e casos de teste para todas as classes utilizadas na execução do *SiMut*. Entre as informações que podem ser extraídas da *killing matrix*, podem-se destacar as seguintes: escore de mutação para um programa particular, para um grupo de programas similares e para todos os programas do banco de dados. Além disso, também pode-se extrair, de um programa ou grupos de programas, o escore de mutação de um operador de mutação específico.

Todos os resultados gerados pelo módulo são armazenados em arquivos CSV. Deste modo, para cada execução realizada utilizando-se o *SiMut*, um arquivo CSV é gerado. Esse arquivo pode então ser interpretado e processado por outras ferramentas. Para este trabalho foi desenvolvido um *script* em NodeJS que interpreta os resultados gerados pelo módulo *CostReduction*, no qual é possível filtrar os N melhores operadores de cada grupo de programas.

3.8 Considerações Finais

Neste capítulo foi descrito como o *framework SiMut* foi implementado, citando as ferramentas de abstrações e similaridades acopladas, as combinações possíveis, o formato do arquivo de execução e a técnica de redução de custo aplicada. No próximo capítulo apresenta-se a formação dos 20 experimentos utilizando o *framework* implementado, citando-se as questões de pesquisa e os procedimentos para a análise dos resultados.

4 Desenho do Experimento

4.1 Considerações Iniciais

Neste capítulo apresenta-se como foi realizado o experimento utilizando o *framework* implementado. Deste modo, o objetivo deste capítulo é detalhar cada etapa realizada na experimentação desde a definição do objetivo do estudo até os procedimentos para a análise de resultados. O restante desse capítulo está organizado da seguinte forma: na seção 4.2 o objetivo do estudo e as questões de pesquisa são apresentados; na seção 4.3 são apresentados os artefatos utilizados nos experimentos; na seção 4.4 discutem-se as configurações que foram executadas no *framework* para chegar aos resultados; por fim, a seção 4.5 apresenta os procedimentos para a análise dos resultados.

4.2 Objetivo do Estudo e Questões de Pesquisa

Estudos anteriores (DALLILO; PIZZOLETO; FERRARI, 2019; PIZZOLETO et al., 2020) evidenciaram de forma preliminar que o uso de similaridade de programas pode trazer benefícios para a redução do custo do teste de mutação. No experimento descrito nesse capítulo, objetivou-se analisar os benefícios que diferentes técnicas de abstrações e similaridades podem trazer. Para isso, foi explorado uma variedade de configurações de execução no *SiMut* para responder a seguinte questão de pesquisa geral: *O uso de abstrações de software induzem a formação de grupos de programas similares, os quais consequentemente podem trazer benefícios para a redução de custo do teste de mutação?* Essa questão de pesquisa geral foi dividida nas seguintes questões de pesquisa:

- **RQ1** - *O uso da abstração de software, no formato de métricas internas, induz a formação de grupos de programas similares, a qual consequentemente podem trazer benefícios para a redução de custo do teste de mutação?*

  Essa questão de pesquisa visa investigar se configurações que utilizam métricas internas e variados algoritmos de *clustering* podem favorecer na formação de grupos de programas similares, que consequentemente podem auxiliar na redução de custo do teste de mutação. Deste modo, através de análise empírica, é possível dizer se essa abordagem traz de fato vantagem para a redução de custo do teste de mutação como constatado em estudos anteriores.

- **RQ2** - *O uso de abstração de software, no formato de código fonte original ou processado, induz a formação de grupos de programas similares, os quais consequentemente podem trazer benefícios para a redução de custo do teste de mutação?*

Essa questão de pesquisa visa investigar se as configurações nas quais se utiliza código-fonte (original ou processado) como entrada em variadas abordagens de similaridade podem auxiliar na formação do no grupo de referência G, na qual traz benefícios para a redução de custo do teste de mutação.

- **RQ3** - *O uso de uma forma alternativa de código fonte (métricas internas ou código ofuscado) leva a uma formação mais precisa de grupos de programas similares quando comparado ao código-fonte original, o que consequentemente pode trazer benefícios para reduzir o custo do teste de mutação?*

  Essa questão de pesquisa propõe uma comparação da efetividade de grupos de referência G, quando utilizado alguma forma alternativa de código-fonte, como métricas internas ou código ofuscado, com relação ao código-fonte original. Deste modo, será investigado se o processamento de geração dessa representação do código-fonte se faz necessária para alcançar melhores resultados para a redução de custo do teste de mutação.

4.3 Artefatos Utilizados

Para a execução dos experimentos, foram selecionados 35 dos 38 programas utilizados por Dallilo, Pizzoleto e Ferrari (2019). Os outros três programas não foram selecionados por não serem compiláveis, por conta da ausência de dependências não encontradas na internet. De maneira geral, os programas selecionados são pequenos e consistem de apenas uma classe Java. Somente foram selecionados programas compiláveis. Para facilitar a execução no *framework* e também na geração de casos de teste e mutantes, os 35 programas foram combinados em um único projeto no padrão Maven, chamado de *simplePrograms*.

Para este trabalho utilizou-se parte de um banco de dados desenvolvido por Pizzoleto, Guarnieri e Ferrari (2021) que tem como proposta servir de insumo para estudos envolvendo teste de mutação. O banco de dados supracitado contém 5 projetos cadastrados e possui aproximadamente 2.000 classes, 50.000 casos de teste e 195.000 mutantes para esses projetos. Para os experimentos deste trabalho, utilizou-se apenas um projeto, com os seus respectivos casos de teste e mutantes, denominado *simplePrograms*.

Neste banco de dados, os casos de teste foram gerados pela ferramenta EvoSuite[1] na versão 1.0.6. Já os mutantes foram gerados e executados pela ferramenta PIT[2] na versão 1.6.2. As duas ferramentas são amplamente utilizadas para experimentação em pesquisa de teste de software para programas Java. No banco de dados, tem-se a relação completa de todos os mutantes de cada projeto, indicando o estado de cada um deles como: mutante morto, mutante vivo e mutante não coberto pelo caso de teste. Deste modo, o banco de

[1] <https://www.evosuite.org/> - acessado em Janeiro de 2022.
[2] <http://pitest.org/> - acessado em Janeiro de 2022.

dados também fornece informações de escore de mutação por operador de mutação para cada um dos projetos.

Na Tabela 3 mostram-se as métricas descritivas dos programas e casos de teste. A tabela é divida em duas partes: Código da Aplicação, que diz respeito ao código-fonte da aplicação, e EvoSuite/PIT, que diz respeito aos testes e mutantes gerados pela EvoSuite e PIT, respectivamente. Na tabela, TLOC é o número total de linhas do código (desconsiderando-se linhas em branco e comentários), NOM é o número de métodos, CC é a média da complexidade ciclomática (McCabe, 1976) dos métodos da classe, TCs representa o total de casos de teste gerados para a classe, M significa o total de mutantes gerados pela PIT, e MS representa o escore de mutação para a classe. As métricas do código da aplicação foram calculadas usando o *plugin* Eclipse Metrics[3] na versão 1.3.8.

Como apresentado, o número de casos de teste para cada classe varia de 3 a 29, enquanto o número de mutantes varia de 28 a 201. O escore de mutação varia de 16,32% a 95,34% com média 62,66%. Conforme relatado por Pizzoleto, Guarnieri e Ferrari (2021), a EvoSuite foi configurada para aplicar o conjunto completo de critérios de geração de cobertura de teste (ROJAS et al., 2015), enquanto a PIT foi configurada para aplicar o conjunto completo de 28 operadores de mutação. Para este trabalho não foi adicionado nenhum caso de teste criado manualmente, nem identificaram-se mutantes equivalentes.

Do conjunto completo de operadores de mutação da PIT, apenas 17 operadores de mutação produziram mutantes para os programas selecionados para o experimento. Na Tabela 4 listam-se os 17 operadores, apresentando-se o nome e a descrição de cada um deles.

4.4 Escolha das Configurações de Execução

As configurações selecionadas para serem executadas no *SiMut* foram definidas tendo-se como base as questões de pesquisa. Um sumário com relação às escolhas é apresentado a seguir.

RQ1: Na Tabela 5 listam-se todas as métricas computadas pela CKJM extended e que estão envolvidas neste trabalho. Nesta tabela, as sete métricas que estão com o sinal de "*" (WMC, $LCOM$, $LCOM3$, LOC, CAM, AMC e CC) formam um grupo específico de métricas, denominado *métricas selecionadas*, que foi escolhido com base nas características dos programas usados nos experimentos, os quais não continham características de orientação a objetos como, por exemplo, herança e encapsulamento. Para a formação dos grupos de programas mais similares utilizando como entrada métricas internas, os algoritmos de *clustering* X-Means e EM da Weka foram selecionados. Para X-Means foram aplicados os dois algoritmos de distância: Euclidiana e Manhattan. Essas técnicas foram selecionadas

[3] <http://metrics2.sourceforge.net/> acessado em Janeiro de 2022

Tabela 3 – Métricas descritivas dos programas utilizados.

	Código da Aplicação			**EvoSuite/PIT**		
Aplicação / Classe	**TLOC**	**NOM**	**CC**	**# TCs**	**# M**	**MS (%)**
Bisect	30	3	2,000	3	30	73,33
BoundedQueue	63	6	2,000	17	83	89,15
BubCorrecto	51	7	1,571	11	48	64,58
cal	97	3	7,000	15	147	16,32
Calculation	79	9	3,222	38	201	31,84
checkIt	29	2	3,500	9	28	64,28
CheckPalindrome	24	2	2,500	9	38	57,89
countPositive	38	2	3,500	7	34	61,76
DigitReverser	25	2	2,000	7	29	27,58
Find	52	4	3,250	12	84	69,04
findLast	61	3	3,333	9	48	50,00
findVal	58	3	3,333	9	46	47,84
Fourballs	33	2	2,500	10	43	95,34
Gaussian	40	7	1,714	25	96	89,58
Heap	48	8	2,125	26	100	74,00
InversePermutation	28	2	4,000	7	62	56,45
lastZero	37	2	3,500	8	32	59,37
LRS	33	3	2,667	9	52	94,23
MergeSort	37	3	3,333	8	95	48,42
Mid	60	9	1,667	18	51	84,31
numZero	37	2	3,500	8	33	60,60
oddOrPos	38	2	4,000	8	40	57,50
power	45	2	4,000	10	42	50,00
printPrimes	54	3	3,000	9	46	41,30
Queue	63	6	1,833	9	85	65,88
QuickSort	45	4	3,750	4	96	41,66
RecursiveSelectionSort	27	3	2,333	8	55	36,36
Stack	37	7	1,714	18	58	84,48
stats	53	2	2,500	6	89	47,19
sum	35	2	3,000	8	29	55,17
TestPat	34	2	3,500	7	35	91,42
trashAndTakeOut	27	2	2,500	9	33	81,81
Triangle	50	9	2,111	29	139	87,76
Triangulo	88	5	5,000	25	122	85,24
twoPred	38	2	4,000	9	39	51,28
Média	**45,543**	**3,857**	**3,013**	**12,114**	**-**	**62,66**

Fonte: Versão adaptada de Guarnieri, Pizzoleto e Ferrari (2022)

pois já foram utilizadas em experimentos anteriores na área de teste de software (CRUZ; ELER, 2017; DALLILO; PIZZOLETO; FERRARI, 2019). Para verificar se a escolha das métricas baseada nas características das classes traria resultados mais precisos, também foram criadas as mesmas execuções considerando-se *todas as métricas* disponíveis na ferramenta CKJM extended listadas na Tabela 5[4]. A descrição de cada métrica pode ser visualizada na subseção 2.4.1.

RQ2: Os recursos de redução, otimização e ofuscação do ProGuard foram aplicados

[4] Para manter a consistência com a ferramenta e outros trabalhos que também abordaram métricas internas, manteve-se o nome de cada métrica na língua inglesa.

Tabela 4 – Lista de operadores de mutação envolvidos no experimento.

Operador de Mutação	Descrição
ArgumentPropagationMutator	Substitui a chamada de método por um de seus parâmetros de tipo correspondente.
ConditionalsBoundaryMutator	Modifica o limite condicional dos operadores relacionais (<, <=, > e >=) com sua contraparte de limite.
ConstructorCallMutator	Substitui o chamada do método construtor com valor nulo.
IncrementsMutator	Altera o incremento e decremento de variáveis locais.
InlineConstantMutator	Altera constantes inline e substitui valores padrão com base em seu tipo de dados.
InvertNegsMutator	Inverte a negação de variáveis do tipo inteiro e real.
MathMutator	Inverte o operador matemático.
MemberVariableMutator	Modifica classes removendo atribuições a variáveis de membro. Os membros são com o seu valor padrão Java para o tipo específico.
NegateConditionalsMutator	Inverte todos os operadores relacionais(==, !=, >, >=, < e <=).
NonVoidMethodCallMutator	Remove chamadas de métodos com retorno. Seu valor de retorno é substituído pelo valor padrão Java para aquele tipo específico.
RemoveConditionalMutator_EQ_ELSE	Força a verificação de igualdade (== e !=) para ser executado o bloco de instruções do $ELSE$, a expressão é substituída por $false$.
RemoveConditionalMutator_EQ_IF	Força a verificação de igualdade (== e !=) para ser executado o bloco de instruções do IF, a expressão é substituída por $true$.
RemoveConditionalMutator_ORD_ELSE	Força a verificação de operador relacional(>, >=, < e <=) para ser executado o bloco de instruções do $ELSE$, a expressão é substituída por $false$.
RemoveConditionalMutator_ORD_IF	Força a verificação de operador relacional (>, >=, < e <=) para ser executado o bloco de instruções do IF, a expressão é substituída por $true$.
RemoveIncrementsMutator	Remove incrementos de variáveis locais do código fonte.
ReturnValsMutator	Altera os valores de retorno das chamadas de método. Dependendo do tipo de retorno do método, outra mutação é usada.
VoidMethodCallMutator	Remove chamadas de método *void* (sem retorno).

Fonte: Informações retiradas do site oficial da PIT. <https://pitest.org/quickstart/mutators/> - acessado em Janeiro de 2022

ao *bytecode* dos programas. Nas execuções dos experimentos, para formar os *clusters* foi utilizado o código-fonte processado pela ferramenta ProGuard como entrada para as seguintes ferramentas: mdist, JPlag e Weka. Para mdist, aplicaram-se as funções de distância Levenshtein, Jaccard e NCD. Para Weka, aplicaram-se os algoritmos de *clustering* EM e X-Means. Deste modo, para essa questão de pesquisa, aplicaram-se todas as abordagens de similaridade disponíveis no *framework* $SiMut$ que recebem como entrada um código-fonte ofuscado.

RQ3: As mesmas configurações do $SiMut$ que foram usadas para código processado também foram usadas para formarem *clusters* baseando-se no código-fonte original dos programas. Isso possibilitou a comparação da relevância do código processado na abordagem

Tabela 5 – Lista de métricas selecionadas para os experimentos.

Sigla	**Métrica**
*WMC	Weighted methods per class
DIT	Depth of Inheritance Tree
NOC	Number of Children
CBO	Coupling between object classes
RFC	Response for a Class
*LCOM	Lack of cohesion in methods
Ca	Afferent coupling
Ce	Efferent coupling
NPM	Number of Public Methods for a class
*LCOM3	Lack of cohesion in methods Henderson-Sellers version
*LOC	Lines of Code
DAM	Data Access Metric
MOA	Measure of Aggregation
MFA	Measure of Functional Abstraction
*CAM	Cohesion Among Methods of Class
IC	Inheritance Coupling
CBM	Coupling Between Methods
*AMC	Average Method Complexity
*CC	McCabe's Cyclomatic Complexity

Fonte: Tabela elaborada pelo autor com base nas informações disponíveis na página oficial da CKJM extended <http://gromit.iiar.pwr.wroc.pl/p_inf/ckjm/metric.html> - Acessado em Janeiro de 2022.

baseada no *SiMut*.

Para todas as execuções foram definidas a criação de três *clusters*. Para as execuções que utilizaram as técnicas de similaridade mdist ou JPlag, foram definidos os seguintes tamanhos para cada *cluster*: 20% para o *cluster* ao qual a classe não testada u foi alocada; 30% para o cluster intermediário, o qual possui um conjunto de classes que são moderadamente similares a u; e 50% para o *cluster* com as classes mais diferentes de u. Nos experimentos que aplicaram os algoritmos de *clustering* da Weka, na qual não é possível definir um percentual de classes para cada *cluster*, como forma de evitar a criação de *clusters* com muitos elementos, um *reclustering* (isto é, um reagrupamento) foi automaticamente executado caso fosse formado algum *cluster* com mais de 60% das classes. Este *reclustering* ocorre uma única vez; portanto, caso ocorresse a criação de um novo *cluster* com mais de 60% dos elementos, um novo *reclustering* não seria executado.

No total, 20 configurações diferentes foram executadas utilizando-se o *framework*. A Tabela 6 lista todas as configurações selecionadas para os experimentos, indicando a abstração e a similaridade utilizada em cada uma. As configurações que utilizaram abstração de métricas internas serviram de apoio para responder RQ1 e RQ3; as demais configurações foram executadas para analisar a efetividade de se utilizar código processado e código original como abstração. Essas outras configurações serviram de apoio para responder as questões de pesquisa RQ2 e RQ3. Em resumo, todas as ferramentas, com todas as possíveis configurações, disponíveis no *framework SiMut* foram utilizadas.

Tabela 6 – Lista de configurações selecionadas para os experimentos.

RQ	**Abstração**	**Similaridade**
RQ1, RQ3	Todas as métricas	EM
RQ1, RQ3	Todas as métricas	X-Means - Euclidean
RQ1, RQ3	Todas as métricas	X-Means - Manhattan
RQ1, RQ3	Métricas selecionadas	EM
RQ1, RQ3	Métricas selecionadas	X-Means - Euclidean
RQ1, RQ3	Métricas selecionadas	X-Means - Manhattan
RQ2, RQ3	Código processado	mdist - Jaccard
RQ2, RQ3	Código processado	mdist - Levenshtein
RQ2, RQ3	Código processado	mdist - NCD
RQ2, RQ3	Código processado	Weka - X-Means - Euclidean
RQ2, RQ3	Código processado	Weka - X-Means - Manhattan
RQ2, RQ3	Código processado	Weka - EM
RQ2, RQ3	Código processado	JPlag
RQ2, RQ3	Código original	mdist - Jaccard
RQ2, RQ3	Código original	mdist - Levenshtein
RQ2, RQ3	Código original	mdist - NCD
RQ2, RQ3	Código original	Weka - X-Means - Euclidean
RQ2, RQ3	Código original	Weka - X-Means - Manhattan
RQ2, RQ3	Código original	Weka - EM
RQ2, RQ3	Código original	JPlag

Fonte: Produzido pelo autor.

A Figura 16 é um dos arquivos XML de execução utilizados nos experimentos.[5] Este código especificamente é correspondente à configuração *Código processado / mdist - Jaccard*. Em todos os arquivos XML de execução foi selecionado apenas o projeto *simplePrograms* para a execução. É possível observar que para a etapa de abstração foi selecionado o código processado pelo ProGuard, aplicando ofuscamento, encurtador e otimizador de código. No XML também foi definido que todas as classes de *simplePrograms* foram consideradas como não testadas, essa decisão é explicada na seção 4.3. Para a etapa de similaridade, foi selecionada a função de distância Jaccard da ferramenta mdist. Na sequência, através dos elementos *groups* que estão dentro de *similarity*, são definidos os percentuais e os níveis de cada grupo. Ficou definido que para o grupo de nível 0, que é o grupo de referência G, terá 20% das classes.

Para fornecer uma base para comparação da efetividade da abordagem baseada no $SiMut$, foram formados aleatoriamente 10 *clusters* para cada um dos 35 programas utilizados no experimento; resultou-se assim, em um total de 350 *clusters* aleatoriamente formados. Esses *clusters* aleatórios contém aproximadamente 20% dos programas e são usados como base de comparação nas 20 configurações do $SiMut$. A próxima seção aborda como esses resultados são interpretados.

[5] Os demais arquivos de execução podem ser acessados através do seguinte endereço: <https://tinyurl.com/guarnieri2022>

Figura 16 – Arquivo de execução para o experimento: Código processado / mdist - Jaccard.

```
<?xml version="1.0" encoding="UTF-8" standalone="no"?>
<project description="Abstraction: ProGuard / Similarity: mdist - jaccard" name="simplePrograms_UFSCar">
  <programs>
    <program id="1" name="simplePrograms"/>
  </programs>
  <abstractions>
    <abstraction name="ProGuard">
      <options>
        <option name="obfuscate;optimize;shrink"/>
      </options>
    </abstraction>
  </abstractions>
  <similarities groups="3">
    <sources>
      <source name="ProGuard"/>
    </sources>
    <nprograms>
      <nprogram name="simplePrograms">
        <class name="Stack"/>
        <class name="LRS"/>
        <class name="Heap"/>
        <class name="Gaussian"/>
        <class name="Fourballs"/>
        <class name="twoPred"/>
        <class name="BubCorrecto"/>
        <class name="Calculation"/>
        <class name="countPositive"/>
        <class name="DigitReverser"/>
        <class name="Find"/>
        <class name="findVal"/>
        <class name="InversePermutation"/>
        <class name="lastZero"/>
        <class name="MergeSort"/>
        <class name="numZero"/>
        <class name="printPrimes"/>
        <class name="Queue"/>
        <class name="stats"/>
        <class name="TestPat"/>
        <class name="trashAndTakeOut"/>
        <class name="Triangulo"/>
        <class name="Bisect"/>
        <class name="BoundedQueue"/>
        <class name="cal"/>
        <class name="CheckPalindrome"/>
        <class name="findLast"/>
        <class name="Mid"/>
        <class name="oddOrPos"/>
        <class name="power"/>
        <class name="QuickSort"/>
        <class name="RecursiveSelectionSort"/>
        <class name="sum"/>
        <class name="Triangle"/>
        <class name="checkIt"/>
      </nprogram>
    </nprograms>
    <similarity algorithm="jaccard" name="mdist">
      <group level="0" percentage="20"/>
      <group level="1" percentage="30"/>
      <group level="2" percentage="50"/>
    </similarity>
  </similarities>
</project>
```

Fonte: Produzido pelo autor.

4.5 Procedimentos para a Análise de Resultados

Para fins experimentais, cada programa do banco de dados foi considerado um programa não testado u. Com isso, foi possível simular um cenário de caso real em que uma nova classe é alimentada ao $SiMut$, e o *framework* ajuda o testador a aplicar testes de mutação a um custo reduzido. Como no banco de dados existem os artefatos de mutação

Tabela 7 – Trecho dos resultados detalhados.

Configuração - Código original / mdist - Levenshtein	
u	numZero
O_u	ConstructorCallMutator, RemoveConditionalMutator_ORDER_ELSE, ReturnValsMutator, IncrementsMutator, RemoveIncrementsMutator
OC_u	ConstructorCallMutator, ReturnValsMutator, IncrementsMutator, RemoveIncrementsMutator, ConditionalsBoundaryMutator
OP_u	ConstructorCallMutator, IncrementsMutator, RemoveIncrementsMutator, NonVoidMethodCallMutator
OC_1	MathMutator, ConstructorCallMutator, IncrementsMutator, ReturnValsMutator
OC_2	ConstructorCallMutator, IncrementsMutator, RemoveIncrementsMutator, NonVoidMethodCallMutator
R1	V: $IC_u(4)$ >m($IC_1(3)$, $IC_2(3)$)
R2	V: $IC_u(4)$ >$IP_u(3)$
$ORan_1$	ConstructorCallMutator, IncrementsMutator, RemoveIncrementsMutator, MathMutator
$ORan_2$	ConstructorCallMutator, NonVoidMethodCallMutator, NegateConditionalsMutator, IncrementsMutator
$ORan_3$	ConstructorCallMutator, MemberVariableMutator, ReturnValsMutator, NonVoidMethodCallMutator
$ORan_4$	ConstructorCallMutator, IncrementsMutator, RemoveIncrementsMutator, MemberVariableMutator
$ORan_5$	ConstructorCallMutator, RemoveIncrementsMutator, IncrementsMutator, MathMutator
$ORan_6$	ConstructorCallMutator, IncrementsMutator, MathMutator, ReturnValsMutator
$ORan_7$	ConstructorCallMutator, IncrementsMutator, ReturnValsMutator, RemoveIncrementsMutator
$ORan_8$	ConstructorCallMutator, IncrementsMutator, RemoveIncrementsMutator, ReturnValsMutator
$ORan_9$	ConstructorCallMutator, RemoveIncrementsMutator, IncrementsMutator, RemoveConditionalMutator_EQUAL_IF
$ORan_{10}$	ConstructorCallMutator, RemoveConditionalMutator_ORDER_ELSE, IncrementsMutator, MathMutator
$IRanAvg_u$	3
R3	V: $IC_u(4)$ >$IRanAvg_u(3)$

Fonte: Guarnieri, Pizzoleto e Ferrari (2022)

relacionados a u, pode-se avaliar os resultados caso os mesmos parâmetros de redução de custo sejam aplicados a u.

Para analisar os resultados obtidos e identificar as configurações que trouxeram mais benefícios, foi adaptado o modelo de interpretação usado por Dallilo, Pizzoleto e Ferrari (2019). Para cada programa e para cada configuração do $SiMut$, uma tabela de resultado foi criada para armazenar e listar os melhores operadores de mutação. A Tabela 7 exemplifica os resultados de uma das iterações do experimento. Os elementos mostrados são definidos da seguinte forma:

- u: representa o programa considerado não testado em uma iteração.
- C: representa um *cluster*. Em particular,
  - C_u é o *cluster* a qual u está associada (exceto u); e
  - C_i (i=1..4) é um dos *clusters* restantes.
- P_u: representa o conjunto completo de programas considerados para o experimento, exceto u.
- Ran_i: representa os *clusters* formados aleatoriamente, exceto u.

- O: representa uma lista dos melhores operadores de mutação. Em particular,
  - O_u é a lista para u.
  - OC_u é a lista para C_u.
  - OC_i é a lista para C_i.
  - $ORan_i$ (i=1..10) é a lista para Ran_i.
  - e OP_u é a lista para P_u
- I: representa o tamanho da interseção entre uma lista de melhores operadores e uma lista de melhores operadores para u. Em particular,
  - IC_u é o tamanho da interseção entre O_u e OC_u
  - IP_u é o tamanho da interseção entre O_u e OP_u
  - $IRanAvg_u$ é a média do tamanho da interseção de O_u com todos os $ORan_i$(i = 1..10)
  - IC_i (i=1..4) é o tamanho da interseção entre O_u e um OC_i particular.
- m: representa uma função que retorna o maior valor entre um grupo de valores.

O valor numérico entre parênteses é o tamanho da interseção. $R1$, $R2$ e $R3$, definidos na sequência, podem ter um resultado positivo (V), neutro (N), ou negativo (X).

- $R1$: compara os resultados obtidos para C_u com os outros *clusters* formados. O resultado é positivo se IC_u for maior que todos os outros resultados para IC_i; negativo se IC_u for menor que qualquer outro resultado para IC_i; ou neutro se o resultado for o mesmo em todos os casos.
- $R2$: parecido com o $R1$, $R2$ compara os resultados obtidos para C_u com o conjunto completo de programas, exceto u. Resultado positivo, neutro e negativo são igualmente computados como feito para $R1$.
- $R3$: similarmente a $R1$ e $R2$, $R3$ compara os resultados obtidos para C_u com a média dos resultados obtidos para os *clusters* aleatórios.

Na Tabela 7 listam-se os resultados para uma iteração de uma configuração do $SiMut$ em que consideraram-se os quatro melhores operadores, ou mais em caso de empate de escore entre eles, para cada grupo de programas. Para o programa `numZero`, os resultados são:

- O_u tem 5 operadores: *Constructor Call Mutator*, *Remove Conditional Mutator _ORDER _ELSE*, *Return Vals Mutator*, *Increments Mutator* e *Remove Increments Mutator*

- OC_u tem 5 operadores: *Constructor Call Mutator*, *Return Vals Mutator*, *Increments Mutator*, *Remove Increments Mutator* e *Conditionals Boundary Mutator*
- OP_u tem 4 operadores: *Constructor Call Mutator*, *Increments Mutator*, *Remove Increments Mutator* e *Non Void Method Call Mutator*
- $R1$: tem resultado positivo, pois IC_u é maior que m(IC_1, IC_2).
- $R2$: tem resultado positivo, pois IC_u é maior que IP_u.
- $IRanAvg_u$: é igual 3 pois esse é o resultado da média de todas as interseções entre O_u com todos os $ORan_i$(i=1..10).
- $R3$: tem um resultado positivo, pois IC_u é maior que $IRanAvg_u$.

Este processo foi executado para todas as classes em todas as configurações mencionadas na seção anterior. No próximo capítulo discutem-se os resultados alcançados.

4.6 Considerações Finais

Neste capítulo apresentou-se como o experimento foi realizado usando-se o *framework* implementado. No capítulo foram descritos os artefatos utilizados, as questões de pesquisa, as combinações selecionadas e a abordagem empregada para a etapa de análise de resultado. No próximo capítulo apresentam-se os resultados obtidos após a execuções de todos os experimentos, destacando a efetividade de cada configuração selecionada.

5 Resultados e Análise

Neste capítulo discutem-se os resultados obtidos e as análises após a experimentação executada através do *framework* implementado. Na seção 5.1 apresentam-se algumas informações iniciais sobre como a análise dos resultados foi realizada; na seção 5.2 apresentam-se os resultados obtidos considerando duas abordagens variando-se a quantidade de melhores operadores para cada grupo de programas; na seção 5.3 apresentam-se as análises dos resultados obtidos considerando um grupo maior de operadores, e também é feito um ranqueamento das configurações com base nos resultados obtidos para R1, R2 e R3; na seção 5.4, com base nos resultados obtidos, apresentam-se as respostas para as questões de pesquisa listadas na seção 4.2; por fim, na seção 5.5 apresentam-se as ameaças à validade dos experimentos realizados neste trabalho.

5.1 Considerações Iniciais

Para a geração dos resultados, a execução do experimento foi dividida em duas partes. A primeira considera, para cada configuração do $SiMut$, somente o melhor operador de mutação, na qual buscou-se analisar os resultados utilizando a técnica de redução de custo *One-Op* (UNTCH, 2009) . Já a segunda parte considera um grupo com os quatro melhores operadores de mutação, estratégia inspirada na Mutação Seletiva (OFFUTT; ROTHERMEL; ZAPF, 1993). A escolha dessas duas abordagens se deu pelos resultados inconclusivos considerando-se apenas o melhor operador para cada grupo de programas. Na próxima seção detalha-se o motivo pela qual ampliou-se a quantidade de melhores operadores de cada grupo de programas, e também apresentam-se os resultados obtidos nas duas abordagens.

5.2 Resultados

A proposta inicial para a análise dos resultados foi considerar e comparar apenas o melhor operador de mutação para cada grupo de programas considerado. Entretanto, após a coleta dos resultados nessa abordagem, constatou-se que a grande maioria das interseções (ou seja, o conjunto de operadores em comum entre os grupos de programas) foram vazias ou de tamanho mínimo. Após uma análise aprofundada, identificou-se um operador de mutação (chamado de *Constructor Call Mutator*) predominante na maioria grupos de programas. Esse naturalmente acabou sendo o único operador presente em OC_u e OP_u na maioria das ocasiões.

Tabela 8 – Trecho de resultados de duas classes considerando apenas o melhor operador.

Configuração - Código original / mdist - Levenshtein	
u	numZero
O_u	ConstructorCallMutator
OC_u	ConstructorCallMutator
OP_u	ConstructorCallMutator
OC_1	MathMutator
OC_2	ConstructorCallMutator
R1	V: $IC_u(1) >= \text{m}(IC_1(0)\ ,\ IC_2(1))$
R2	N: $IC_u(1) = IP_u(1)$
$ORan_1$	ConstructorCallMutator
$ORan_2$	ConstructorCallMutator
$ORan_3$	ConstructorCallMutator
$ORan_4$	ConstructorCallMutator
$ORan_5$	ConstructorCallMutator
$ORan_6$	ConstructorCallMutator
$ORan_7$	ConstructorCallMutator
$ORan_8$	ConstructorCallMutator
$ORan_9$	ConstructorCallMutator
$ORan_{10}$	ConstructorCallMutator
R3	N: $IC_u(1) = IRanAvg_u(1)$
Configuração - Métricas selecionadas / X-Means - Manhattan	
u	Find
O_u	ReturnValsMutator
OC_u	ConstructorCallMutator
OP_u	ConstructorCallMutator
OC_1	ConstructorCallMutator
OC_2	ConstructorCallMutator
R1	N: $IC_u(0) = \text{m}(IC_1(0)\ ,\ IC_2(0))$
R2	N: $IC_u(0) = IPu(0)$
$ORan_1$	ConstructorCallMutator
$ORan_2$	ConstructorCallMutator
$ORan_3$	ConstructorCallMutator
$ORan_4$	ConstructorCallMutator
$ORan_5$	IncrementsMutator
$ORan_6$	ConstructorCallMutator
$ORan_7$	IncrementsMutator
$ORan_8$	ConstructorCallMutator
$ORan_9$	ConstructorCallMutator
$ORan_{10}$	ConstructorCallMutator
R3	N: $IC_u(0) = IRanAvg_u(0)$

Fonte: Produzido pelo autor.

Para exemplificar essa constatação, na Tabela 8 listam-se parte dos resultados obtidos, na qual apresentam-se os melhores operadores para cada grupo de programas considerando apenas o melhor operador de mutação.[1] Nesta tabela listam-se resultados de duas iterações para configurações diferentes: *(i) Código original / mdist - Levenshtein* e *(ii) Métricas selecionadas / X-Means - Manhattan*. Os resultados listados são para duas classes: *(i) numZero* e *(ii) Find*. Além da tabela conter os melhores operadores, também listam-se os resultados para R1, R2 e R3 para as duas classes em suas respectivas configurações.

Nos resultados para a classe *numZero* na configuração *Código original / mdist - Levenshtein*, como apresentado na Tabela 8, constatou-se que o operador *Constructor Call Mutator* foi predominante na maioria dos grupos de programas, na qual somente em OC_2

[1] Todos os resultados detalhados podem ser acessados por meio do endereço: <https://tinyurl.com/guarnieri2022>

o operador foi diferente, sendo o *Math Mutator*. Neste primeiro experimento, o resultado para R1 foi positivo, enquanto para R2 e R3 os resultados foram neutros. Os resultados neutros ocorreram por conta do tamanho da interseção dos grupos de operadores entre os grupos de programas, na qual se mostrou igual a um na maioria das vezes.

Ainda sobre a Tabela 8, no experimento em que considerou-se *Find* como classe não testada u, também é notório que na maioria das ocasiões, somente o operador *Constructor Call Mutator* prevaleceu. Para essa execução, os resultados para R1, R2 e R3 foram neutros. Nota-se que o melhor operador para u foi *Return Vals Mutator*, o que diferiu dos melhores operadores para todos os grupos de programa, gerando interseção igual a zero para a maioria dos resultados.

Como forma de evitar resultados neutros para então viabilizar conclusões em cima dos resultados obtidos, também foram coletados resultados considerando um grupo maior de operadores. Na Tabela 9 listam-se os melhores operadores e os resultados para as mesmas classes e configurações da Tabela 8, porém considerando-se os quatro melhores operadores de mutação para cada grupo de programas, e não apenas o melhor operador. Note que a quantidade de operadores pode ser maior que quatro, pois pode acontecer empate de escore de mutação entre os operadores com melhor ranqueamento em cada grupo de programas.

Considerando os quatro melhores operadores para cada grupo de programas, observou-se uma melhor distribuição de operadores para as duas classes nas duas configurações. Constatou-se que para a classe *numZero*, os resultados para R1, R2 e R3 foram positivos. Já para a classe *Find*, os resultados obtidos para R1 e R3 foram negativos, enquanto para R2 o resultado foi neutro. Deste modo, o aumento na quantidade de melhores operadores fez com que a maioria dos resultados que antes estavam como neutros, mudassem para positivos ou negativos, viabilizando assim conclusões em cima dos resultados obtidos por meio dos experimentos.

Essa distribuição de resultados também foi positiva na maioria das classes e configurações selecionadas para o experimento. As duas próximas subseções apresentam os resultados sumarizados para as duas abordagens em que variou-se a quantidade de melhores operadores para cada grupo de programas.

5.2.1 Resultados considerando apenas o melhor operador:

A primeira parte do experimento considerou o melhor operador para cada grupo de programas: o programa u, o *cluster* C_u, os *clusters* C_i, o conjunto completo de programas P e os 10 *clusters* formados aleatoriamente Ran_i (i = 1..10). Na Tabela 10 apresenta-se o resumo dos resultados utilizando essa abordagem. É importante destacar que os grupos de programas não consideram u, que é tratado como um programa não testado nos

Tabela 9 – Resultados para duas classes considerando grupo de quatro melhores operadores.

Configuração - Código original / mdist - Levenshtein	
u	numZero
O_u	ConstructorCallMutator, RemoveConditionalMutator_ORDER_ELSE, ReturnValsMutator, IncrementsMutator, RemoveIncrementsMutator
OC_u	ConstructorCallMutator, ReturnValsMutator, IncrementsMutator, RemoveIncrementsMutator, ConditionalsBoundaryMutator
OP_u	ConstructorCallMutator, IncrementsMutator, RemoveIncrementsMutator, NonVoidMethodCallMutator
OC_1	MathMutator, ConstructorCallMutator, IncrementsMutator, ReturnValsMutator
OC_2	ConstructorCallMutator, IncrementsMutator, RemoveIncrementsMutator, NonVoidMethodCallMutator
R1	V: $IC_u(4) > \mathrm{m}(IC_1(3)\ ,\ IC_2(3))$
R2	V: $IC_u(4) > IP_u(3)$
$ORan_1$	ConstructorCallMutator, IncrementsMutator, RemoveIncrementsMutator, MathMutator
$ORan_2$	ConstructorCallMutator, NonVoidMethodCallMutator, NegateConditionalsMutator, IncrementsMutator
$ORan_3$	ConstructorCallMutator, MemberVariableMutator, ReturnValsMutator, NonVoidMethodCallMutator
$ORan_4$	ConstructorCallMutator, IncrementsMutator, RemoveIncrementsMutator, MemberVariableMutator
$ORan_5$	ConstructorCallMutator, RemoveIncrementsMutator, IncrementsMutator, MathMutator
$ORan_6$	ConstructorCallMutator, IncrementsMutator, MathMutator, ReturnValsMutator
$ORan_7$	ConstructorCallMutator, IncrementsMutator, ReturnValsMutator, RemoveIncrementsMutator
$ORan_8$	ConstructorCallMutator, IncrementsMutator, RemoveIncrementsMutator, ReturnValsMutator
$ORan_9$	ConstructorCallMutator, RemoveIncrementsMutator, IncrementsMutator, RemoveConditionalMutator_EQUAL_IF
$ORan_{10}$	ConstructorCallMutator, RemoveConditionalMutator_ORDER_ELSE, IncrementsMutator, MathMutator
R3	V: $IC_u(4) > IRanAvg_u(3)$
Configuração - Métricas selecionadas / X-Means - Manhattan	
u	Find
O_u	ReturnValsMutator, ConditionalsBoundaryMutator, RemoveConditionalMutator_ORDER_ELSE, IncrementsMutator
OC_u	ConstructorCallMutator, NonVoidMethodCallMutator, IncrementsMutator, NegateConditionalsMutator
OP_u	ConstructorCallMutator, IncrementsMutator, RemoveIncrementsMutator, MemberVariableMutator
OC_1	ConstructorCallMutator, IncrementsMutator, RemoveIncrementsMutator, ReturnValsMutator
OC_2	ConstructorCallMutator, IncrementsMutator, RemoveIncrementsMutator, MathMutator
R1	X: $IC_u(1) < \mathrm{m}(IC_1(2)\ ,\ IC_2(1))$
R2	N: $IC_u(1) = IP_u(1)$
$ORan_1$	ConstructorCallMutator, RemoveIncrementsMutator, IncrementsMutator, MathMutator
$ORan_2$	ConstructorCallMutator, ReturnValsMutator, IncrementsMutator, RemoveIncrementsMutator
$ORan_3$	ConstructorCallMutator, RemoveIncrementsMutator, IncrementsMutator, NonVoidMethodCallMutator
$ORan_4$	ConstructorCallMutator, RemoveIncrementsMutator, IncrementsMutator, ReturnValsMutator
$ORan_5$	IncrementsMutator, ConstructorCallMutator, MemberVariableMutator, RemoveIncrementsMutator
$ORan_6$	ConstructorCallMutator, IncrementsMutator, RemoveIncrementsMutator, NegateConditionalsMutator
$ORan_7$	IncrementsMutator, ConstructorCallMutator, RemoveIncrementsMutator, ReturnValsMutator
$ORan_8$	ConstructorCallMutator, IncrementsMutator, RemoveIncrementsMutator, ReturnValsMutator, MemberVariableMutator
$ORan_9$	ConstructorCallMutator, NonVoidMethodCallMutator, IncrementsMutator, MathMutator
$ORan_{10}$	ConstructorCallMutator, RemoveIncrementsMutator, ReturnValsMutator, IncrementsMutator
R3	X: $IC_u(1) < IRanAvg_u(1.5)$

Fonte: Produzido pelo autor.

experimentos.

Na Tabela 10 observa-se que o resultado neutro é o mais presente na maioria das configurações executadas no *SiMut*, especialmente nos resultados de R2 em que IC_u é

Tabela 10 – Resumo dos resultados considerando somente o melhor operador para cada grupo de programas.

Métricas selecionadas / X-Means - Euclidean		**Código processado / X-Means - Manhattan**	
R1	V(11), N(12), X(12)	R1	V(15), N(14), X(6)
R2	V(2), N(25), X(8)	R2	V(0), N(33), X(2)
R3	V(8), N(18), X(9)	R3	V(10), N(22), X(3)
Todas as métricas / EM		**Todas as métricas / X-Means - Euclidean**	
R1	V(10), N(18), X(7)	R1	V(0), N(32), X(3)
R2	V(1), N(29), X(5)	R2	V(0), N(32), X(3)
R3	V(9), N(20), X(6)	R3	V(10), N(21), X(4)
Todas as métricas / X-Means - Manhattan		**Métricas selecionadas / EM**	
R1	V(0), N(33), X(2)	R1	V(0), N(34), X(1)
R2	V(0), N(33), X(2)	R2	V(0), N(34), X(1)
R3	V(11), N(21), X(3)	R3	V(11), N(22), X(2)
Métricas selecionadas / X-Means - Manhattan		**Código original / JPlag**	
R1	V(0), N(34), X(1)	R1	V(1), N(34), X(0)
R2	V(0), N(34), X(1)	R2	V(0), N(35), X(0)
R3	V(11), N(22), X(2)	R3	V(12), N(22), X(1)
Código original / mdist - Jaccard		**Código original / mdist - Levenshtein**	
R1	V(0), N(33), X(2)	R1	V(7), N(24), X(4)
R2	V(0), N(34), X(1)	R2	V(0), N(32), X(3)
R3	V(11), N(22), X(2)	R3	V(10), N(21), X(4)
Código original / EM		**Código original / X-Means - Euclidean**	
R1	V(17), N(17), X(1)	R1	V(17), N(14), X(4)
R2	V(0), N(35), X(0)	R2	V(0), N(35), X(0)
R3	V(12), N(22), X(1)	R3	V(12), N(22), X(1)
Código original / X-Means - Manhattan		**Código processado / JPlag**	
R1	V(17), N(14), X(4)	R1	V(0), N(35), X(0)
R2	V(0), N(35), X(0)	R2	V(0), N(35), X(0)
R3	V(12), N(22), X(1)	R3	V(12), N(22), X(1)
Código processado / mdist - Jaccard		**Código processado / mdist - Levenshtein**	
R1	V(0), N(35), X(0)	R1	V(10), N(22), X(3)
R2	V(0), N(35), X(0)	R2	V(2), N(30), X(3)
R3	V(12), N(22), X(1)	R3	V(12), N(20), X(3)
Código processado / EM		**Código processado / X-Means - Euclidean**	
R1	V(17), N(13), X(5)	R1	V(16), N(13), X(6)
R2	V(0), N(35), X(0)	R2	V(0), N(34), X(1)
R3	V(12), N(22), X(1)	R3	V(11), N(22), X(2)
Código processado / mdist - NCD		**Código original / mdist - NCD**	
R1	V(3), N(31), X(1)	R1	V(2), N(32), X(1)
R2	V(0), N(35), X(0)	R2	V(0), N(34), X(1)
R3	V(12), N(22), X(1)	R3	V(11), N(22), X(2)

Fonte: Versão adaptada de Guarnieri, Pizzoleto e Ferrari (2022)

comparado com IP_u. Este comportamento também impactou os resultados de R1 e R3. Por exemplo, usando-se código processado pelo ProGuard como abstração e o NCD do mdist como similaridade, em R1 e R3 existiram 31 e 22 resultados neutros, respectivamente.

Na prática, os resultados obtidos considerando-se somente o melhor operador foram inconclusivos para o estudo empírico proposto neste trabalho. Na próxima subseção apresentam-se os resultados sumarizados considerando-se um grupo maior de melhores operadores.

5.2.2 Resultados considerando os quatro melhores operadores

Dados os resultados inconclusivos considerando-se apenas o melhor operador para cada grupo de programas, decidiu-se expandir o grupo dos melhores operadores; mais especificamente, os operadores de mutação foram ranqueados de acordo com os respectivos escores de mutação para os conjuntos de programas considerados (por exemplo, u e C_u), e escolheu-se 25% dos operadores com as pontuações de escore de mutação mais altas.

Dado que 17 operadores de mutação da ferramenta PIT produziram mutantes para os programas alvo, os grupos de melhores operadores tinham quatro, ou mais em caso de empate de escore, operadores cada. O aumento do número dos melhores operadores em cada grupo permitiu um melhor equilíbrio nos resultados, com maiores quantidades de resultados positivos e negativos.

Na Tabela 11 listam-se os resultados sumarizados considerando-se os 25% melhores operadores de mutação para cada grupo de programas. Nessa tabela, nota-se que a quantidade de resultados neutros caiu de forma significativa; por exemplo, na configuração *Código original / JPlag*, R1 teve apenas 5 resultados neutros, R2 ficou com 10 resultados neutros e R3 ficou com apenas 1 resultado neutro. Somente para R2 nas execuções das configurações *Código original / X-Means - Euclidean* e *Código original - X-Means - Manhattan*, o resultado neutro ficou presente em mais de 50% dos programas envolvidos. Foi com base nessa abordagem que se analisaram os resultados obtidos neste trabalho. Na próxima seção apresenta-se a análise dos resultados obtidos.

5.3 Análise dos Resultados

Para a análise dos resultados, ressalta-se que: R1 avalia se C_u (ou seja, o *cluster* ao qual a classe u está associada) é um melhor (ou pelo menos tão bom quanto) preditor do escore de mutação do que os demais clusters C_i computados pelo $SiMut$. Portanto, um resultado neutro pode ser considerado satisfatório para R1. Da mesma forma, um resultado neutro para R2 também é satisfatório, tendo em vista que C_u é um preditor de escore de mutação do operador de mutação tão bom quanto o conjunto completo de programas, já que computar os melhores operadores para um único *cluster* é menos custoso do que para o conjunto de programas como um todo. Para R3, um resultado neutro é claramente insatisfatório. Isso se deve ao baixo custo para formar os *clusters* aleatoriamente sem a necessidade de computar abstrações, medidas e similaridade.

Com base nessa lógica de interpretação e considerando os resultados que levam em conta os quatro melhores operadores de mutação, na Tabela 12 é apresentado um ranqueamento em que os valores de R1, R2 e R3 podem variar de 1 a 35. Os valores das colunas R1 e R2 são a soma dos resultados positivos e neutros, enquanto a coluna R3 totaliza somente os resultados positivos. Na coluna T, por sua vez, apresenta-se a soma de

Tabela 11 – Resumo dos resultados considerando os quatro melhores operadores para cada grupo de programas.

Métricas selecionadas / X-Means - Euclidean		**Código processado / X-Means - Manhattan**	
R1	V(19), N(3), X(13)	R1	V(17), N(1), X(17)
R2	V(16), N(13), X(6)	R2	V(12), N(13), X(10)
R3	V(21), N(3), X(11)	R3	V(15), N(2), X(18)
Todas as métricas / EM		**Todas as métricas / X-Means - Euclidean**	
R1	V(21), N(2), X(12)	R1	V(21), N(1), X(13)
R2	V(19), N(10), X(6)	R2	V(17), N(12), X(6)
R3	V(22), N(3), X(10)	R3	V(22), N(1), X(12)
Todas as métricas / X-Means - Manhattan		**Métricas selecionadas / EM**	
R1	V(21), N(0), X(14)	R1	V(21), N(4), X(10)
R2	V(17), N(12), X(6)	R2	V(17), N(14), X(4)
R3	V(21), N(1), X(13)	R3	V(23), N(3), X(9)
Métricas selecionadas / X-Means - Manhattan		**Código original / JPlag**	
R1	V(20), N(4), X(11)	R1	V(19), N(5), X(11)
R2	V(16), N(15), X(4)	R2	V(17), N(10), X(8)
R3	V(22), N(3), X(10)	R3	V(19), N(1), X(15)
Código original / mdist - Jaccard		**Código original / mdist - Levenshtein**	
R1	V(19), N(7), X(9)	R1	V(21), N(6), X(8)
R2	V(17), N(15), X(3)	R2	V(19), N(12), X(4)
R3	V(21), N(2), X(12)	R3	V(22), N(3), X(10)
Código original / EM		**Código original / X-Means - Euclidean**	
R1	V(21), N(2), X(12)	R1	V(18), N(2), X(15)
R2	V(16), N(11), X(8)	R2	V(11), N(22), X(2)
R3	V(19), N(2), X(14)	R3	V(20), N(4), X(11)
Código original / X-Means - Manhattan		**Código processado / JPlag**	
R1	V(19), N(1), X(15)	R1	V(17), N(5), X(13)
R2	V(13), N(20), X(2)	R2	V(16), N(11), X(8)
R3	V(22), N(3), X(10)	R3	V(17), N(3), X(15)
Código processado / mdist - Jaccard		**Código processado / mdist - Levenshtein**	
R1	V(23), N(6), X(6)	R1	V(22), N(4), X(9)
R2	V(20), N(12), X(3)	R2	V(22), N(7), X(6)
R3	V(27), N(0), X(8)	R3	V(24), N(0), X(11)
Código processado / EM		**Código processado / X-Means - Euclidean**	
R1	V(19), N(0), X(16)	R1	V(17), N(0), X(18)
R2	V(14), N(11), X(10)	R2	V(13), N(12), X(10)
R3	V(15), N(2), X(18)	R3	V(15), N(2), X(18)
Código processado / mdist - NCD		**Código original / mdist - NCD**	
R1	V(19), N(7), X(9)	R1	V(19), N(10), X(6)
R2	V(19), N(11), X(5)	R2	V(19), N(13), X(3)
R3	V(23), N(1), X(11)	R3	V(24), N(3), X(8)

Fonte: Versão adaptada de Guarnieri, Pizzoleto e Ferrari (2022)

R1, R2 e R3. Cada percentual entre parênteses representa a efetividade obtida dentro do intervalo possível.

5.3.1 Análise baseada em R1, R2 e R3, em conjunto:

Considerando os três grupos de resultados resumidos na Tabela 12, nota-se que um substancial grupo de configurações executadas no *SiMut* apresentaram resultados promissores. As configurações *Código processado / mdist - Jaccard* e *Código original /*

Tabela 12 – Classificação dos resultados (R*i* podendo variar de 1 a 35).

#	Configuração	R1	R2	R3	T
1	Código processado / mdist - Jaccard	29 (82,86%)	32 (91,43%)	27 (77,14%)	88 (83,81%)
2	Código original / mdist - NCD	29 (82,86%)	32 (91,43%)	24 (68,57%)	85 (80,95%)
3	Código original / mdist - Levenshtein	27 (77,14%)	31 (88,57%)	22 (62,86%)	80 (76,19%)
4	Métricas selecionadas / EM	25 (71,43%)	31 (88,57%)	23 (65,71%)	79 (75,24%)
5	Código original / mdist - Jaccard	26 (74,29%)	32 (91,43%)	21 (60,00%)	79 (75,24%)
6	Código processado / mdist - Levenshtein	26 (74,29%)	29 (82,86%)	24 (68,57%)	79 (75,24%)
7	Código processado / mdist - NCD	26 (74,29%)	30 (85,71%)	23 (65,71%)	79 (75,24%)
8	Métricas selecionadas / X-Means - Manhattan	24 (68,57%)	31 (88,57%)	22 (62,86%)	77 (73,33%)
9	Código original / X-Means - Manhattan	20 (57,14%)	33 (94,29%)	22 (62,86%)	75 (71,43%)
10	Todas as métricas / EM	23 (65,71%)	29 (82,86%)	22 (62,86%)	74 (70,48%)
11	Todas as métricas / X-Means - Euclidean	22 (62,86%)	29 (82,86%)	22 (62,86%)	73 (69,52%)
12	Código original / X-Means - Euclidean	20 (57,14%)	33 (94,29%)	20 (57,14%)	73 (69,52%)
13	Métricas selecionadas / X-Means - Euclidean	22 (62,86%)	29 (82,86%)	21 (60,00%)	72 (68,57%)
14	Todas as métricas / X-Means - Manhattan	21 (60,00%)	29 (82,86%)	21 (60,00%)	71 (67,62%)
15	Código original / jPlag	24 (68,57%)	27 (77,14%)	19 (54,29%)	70 (66,67%)
16	Código original / EM	23 (65,71%)	27 (77,14%)	19 (54,29%)	69 (65,71%)
17	Código processado / jPlag	22 (62,86%)	27 (77,14%)	17 (48,57%)	66 (62,86%)
18	Código processado / EM	19 (54,29%)	25 (71,43%)	15 (42,86%)	59 (56,19%)
19	Código processado / X-Means - Manhattan	18 (51,43%)	25 (71,43%)	15 (42,86%)	58 (55,24%)
20	Código processado / X-Means - Euclidean	17 (48,57%)	25 (71,43%)	15 (42,86%)	57 (54,29%)

Fonte: Versão adaptada de Guarnieri, Pizzoleto e Ferrari (2022)

mdist - NCD, respectivamente, somaram 88 e 85 resultados positivos de um máximo de 105 possíveis.

Essa constatação inicial sugere que a similaridade pode sim ser uma aliada para reduzir o custo dos testes de mutação. Particularmente, a configuração que combinou código processado pelo ProGuard e similaridade computada com a medida de distância Jaccard implementada pela ferramenta mdist apresentou o melhor desempenho em relação às demais (efetividade de aproximadamente 84% para indicar os melhores operadores de mutação). Destaca-se também as configurações que geraram pelo menos 79 resultados positivos, pois esse valor representa uma efetividade superior a 75%.

5.3.2 Análise baseada em R1, R2 e R3, individualmente:

Com relação ao resultado R1, observou-se que algumas configurações tiveram efetividade acima de 75%, todas incluindo mdist como ferramenta de similaridade (mesmo variando as abstrações e medidas de distância). Eles são (i) *Código processado / mdist - Jaccard*, (ii) *Código original / mdist - NCD*, e (iii) *Código original / mdist - Levenshtein*. Muitas outras configurações tiveram efetividade acima de 60%. Em R2, exceto, pelas últimas três configurações apresentadas na Tabela 12 todas as configurações apresentaram efetividade acima de 75% para melhor prever os melhores operadores de mutação quando comparados a todo o conjunto de programas. Por fim, considerando apenas R3, a efetividade de cinco configurações foram de pelo menos 65%, e para três configurações, isso foi acima de 68%: (i) *Código processado / mdist - Jaccard*, (ii) *Código original / mdist- NCD* e (iii) *Código processado / mdist - Levenshtein*. Mais uma vez, todas as configurações que se destacaram quando comparadas aos *clusters* formados aleatoriamente incluem mdist como

ferramenta de similaridade.

5.4 Revisitando as Questões de Pesquisa

Em relação à RQ1, as configurações que usaram métricas de complexidade interna como abstração de software sobre a qual a similaridade é calculada produziram resultados variados. Entre essas configurações, (i) *Métricas selecionadas / EM* e (ii) *Métricas selecionadas / X-Means - Manhattan* apresentaram os melhores resultados, alcançando, respectivamente 75% e 73% de efetividade. Essas duas melhores configurações usaram apenas sete métricas computadas pela ferramenta CKJM extended. Todas as outras configurações que também utilizaram métricas internas como forma de abstração alcançaram efetividade maior que 67%.

Portanto, em relação à questão de pesquisa RQ1 *conclui-se que o uso de métricas de complexidade interna como forma de abstração, combinada com algoritmos de agrupamento, pode induzir positivamente a formação de grupos de programas similares que contribuem para a redução de custo do teste de mutação.*

Similarmente às configurações que usaram métricas internas, as configurações que usaram código-fonte como abstração de software produziram resultados variados de acordo com a técnica de similaridade aplicada. Em particular, a formação de grupos de programas similares pela ferramenta mdist se mostrou promissora com código processado e código original; em todas as execuções que usaram mdist, a efetividade ficou acima de 75%. Quando os algoritmos de *clustering* da Weka foram utilizados para agrupar os programas, a maior efetividade foi de 71% da configuração *Código original / X-Means - Manhattan*, seguida pela configuração *Código original / X-Means - Euclidean* (efetividade de 70%). Outras configurações que utilizaram algoritmos de *clustering* da Weka para abstrações de código processado não produziram resultados significantes. Quando utilizado a ferramenta de detecção de plágio JPlag, a efetividade foi de aproximadamente 63% a 67%.

Deste modo, com respeito à questão de pesquisa RQ2, conclui-se *que o uso de abstração de código-fonte (original ou processado), para particulares ferramentas/abordagens de similaridades (mdist e Weka com X-Means), pode induzir positivamente a formação de grupos similares que contribuem para a redução de custo do teste de mutação.*

Quando comparado os resultados das configurações que usaram uma forma alternativa de código-fonte, como métricas internas ou código ofuscado, com os que utilizaram código-fonte original, não é possível identificar uma clara vantagem para nenhum lado. De acordo com os resultados obtidos, enquanto o código processado é apresentado como a configuração com maior efetividade, é possível notar na Tabela 12 que configurações que utilizaram código original aparecem imediatamente em seguida em termos de efetividade (note que ambas as abstrações são processadas pela ferramenta mdist). Deste modo, é

observada uma alternância de tipos de abstrações na tabela de classificação dos resultados. Assim, não há uma tendência clara para o uso vantajoso de qualquer apresentação alternativa do código-fonte além do código-fonte original.

Com isso, sobre RQ3, conclui-se que *o código-fonte original, quando utilizado em uma abordagem baseada em similaridade como a definida no* framework *SiMut, pode ser tão efetiva quanto alguma forma alternativa de representação de código-fonte.*

5.5 Ameaças à Validade

Nesta seção discutem-se as ameaças à validade dos experimentos realizados neste trabalho, agrupadas em três categorias Wohlin et al. (2012): validade interna, de construção e externa.

Interna: Ao usar os algoritmos de *clustering* EM e X-Means da Weka não há garantia que a divisão dos clusters seja a ideal em termos de proximidade entre a classe em análise e o elemento central do agrupamento (ou seja, o centroide). Além disso, a qualidade do banco de dados de artefatos (programas, casos de teste e mutantes) utilizados nos experimentos depende da precisão dos procedimentos e ferramentas utilizadas para criá-lo. Para reduzir as ameaças desta categoria, aplicaram-se os mesmos algoritmos de agrupamento utilizados em experimentos anteriores no contexto de teste de software (CRUZ; ELER, 2017; DALLILO; PIZZOLETO; FERRARI, 2019). Além disso, utilizou-se um banco de dados que vem sendo sistematicamente criado com ferramentas amplamente utilizadas para geração de testes e geração e execução de mutantes (PIZZOLETO; GUARNIERI; FERRARI, 2021).

Construção: A escolha do número mínimo de *clusters* para todos os experimentos foi definido pelo autor. A escolha, embora satisfatória de acordo com os resultados obtidos, pode não representar a melhor opção. O percentual de programas para compor cada *cluster*, quando aplicado, também foi uma decisão que pode não representar a melhor configuração. Diferentes números de *clusters* e diferentes tamanhos de *clusters* podem levar a diferentes resultados. Outra ameaça diz respeito ao uso de programas e artefatos associados (ou seja, casos de teste e mutantes) que estão disponíveis no banco de dados utilizado, de modo que conforme gerado pela EvoSuite e PIT, não houve qualquer adição de casos de teste ou detecção de mutantes equivalentes. Essa abordagem traz uma natureza mais "prática" ao trabalho, tornando-o comparável a um cenário em que engenheiros de software adotam um processo totalmente automatizado para aplicar teste de mutação a um novo programa.

Externa: Os experimentos relatados neste trabalho foram realizados com base em 35 programas simples. Portanto, os resultados podem não se traduzir em outras configurações que, por exemplo, empreguem programas-alvos diferentes (por exemplo, estruturas de classe mais complexas que envolvem relações de herança ou polimorfismo), ou mesmo configurações que usam outras abstrações de software e ferramentas de cálculo de similaridade.

5.6 Considerações Finais

Neste capítulo apresentaram-se e analisaram-se os resultados obtidos após a execução, por meio do *framework* implementado, das 20 configurações selecionadas. A princípio foi considerado apenas o melhor operador para cada grupo de programas; entretanto, foi constatado que a grande maioria das interseções entre OC_u e os grupos de programas foram vazias ou de tamanho mínimo. Por esse resultado considerando apenas um operador ser inconclusivo, escolheram-se 25% dos melhores operadores de cada grupo. Esse aumento na quantidade de operadores possibilitou um melhor equilíbrio nos resultados para a realização das análises.

Com base nos resultados obtidos, foi identificado que a efetividade variou consideravelmente de acordo com a configuração utilizada, na qual as funções de distância entre *strings*, da ferramenta mdist, mostraram-se promissoras tanto para código processado quanto para código original. Considerando-se os experimentos realizados, também constatou-se que o uso do código-fonte original como entrada na etapa de similaridade do $SiMut$ pode ser tão efetivo quanto alguma forma alternativa de representação de código-fonte, como métricas internas ou código ofuscado.

No próximo capítulo encerra-se este trabalho apresentando-se as conclusões, os trabalhos relacionados, as contribuições para a área de teste de mutação, as limitações, as possibilidades de trabalhos futuros, e as publicações resultantes deste trabalho.

6 Conclusão

Neste trabalho apresentou-se uma implementação do *framework* *SiMut*. Essa implementação objetivou ajudar a reduzir o custo do teste de mutação baseando-se em resultados disponíveis para grupos de programas similares já testados com mutação. A implementação deste *framework* incluiu diversificadas formas para computar abstrações de programas e para calcular similaridade entre programas, e suporta uma técnica de redução de custo de mutação que identifica os melhores operadores de mutação de acordo com os seus escores.

Nos experimentos utilizaram-se 20 combinações variadas e 35 programas escritos em Java que já possuíam informações relacionadas ao teste de mutação. Os resultados apresentaram diversas combinações que são efetivas para prever os melhores operadores de mutação para classes não testadas. Em relação às abstrações de programa utilizadas para calcular a similaridade, o código-fonte original parece ser tão relevante quanto o código-fonte ofuscado e as métricas internas de complexidade.

6.1 Trabalhos Relacionados

Nesta seção, apresentam-se alguns dos estudos relacionados a este trabalho. Buscou-se discutir estudos que se apoiaram na similaridade, ou informações internas, de programas para auxiliar no teste de mutação.

TITCHEU CHEKAM et al. (2020) usaram informações estruturais para apoiar o teste de mutação. Nos testes realizados, utilizaram informações internas do programa e aprendizado supervisionado para selecionar mutantes que tornam o teste de mutação mais eficiente. O foco do método foi analisar o menor número possível de mutantes "matáveis" e priorizar os mutantes que revelam a maioria dos defeitos. A abordagem, chamada FaRM, aplica aprendizado supervisionado aos mutantes gerados de um grupo de programas defeituosos, e constrói um modelo de aprendizado. Esse modelo é então aplicado a outros mutantes e gera uma classificação de mutantes destrutíveis e reveladores de defeitos. Nos testes realizados, a técnica apresentou um desempenho superior à técnica de mutação seletiva. Embora o foco deste estudo não tenha sido reaproveitar experimentos anteriores de teste mutação em programas ainda não testados, neste estudo, assim como neste trabalho de mestrado, também utilizaram-se informações estruturais para apoiar o teste de mutação.

Também relacionado à utilização de informações estruturais de softwares, Cruz e Eler (2017) desenvolveram duas análises empíricas sobre a influência das métricas CK,

de Chidamber e Kemerer (1994), na testabilidade de software orientado a objetos. Na primeira análise, as métricas foram analisadas por meio da correlação das métricas CK com a cobertura de linha de código e escore de mutação. Na segunda análise empírica, foi realizada uma proposta de clusterização das métricas para tentar identificar grupos de classes com características semelhantes relacionadas à testabilidade. Apesar das limitações dos testes descritas no trabalho, as duas análises empíricas conseguiram demonstrar a importância de cada métrica CK dentro do contexto de testabilidade. Desta forma, no trabalho de Cruz e Eler (2017) foi estabelecida uma correlação entre métricas internas de software e testabilidade de programas. Essa correlação também foi constatada neste trabalho de mestrado, tendo em vista que, como discutido na seção 5.2, os experimentos que utilizaram métricas internas também produziram bons resultados.

Em um outro trabalho, desta vez visando reduzir a quantidade de mutantes equivalentes em um software ainda não testado, Kintis e Malevris (2013) se apoiaram em algoritmos de similaridade de programas. Nesta pesquisa, foram identificados os mutantes equivalentes de uma aplicação e na sequência, após verificar semelhanças deste programa com um outro ainda não testado, foi possível verificar, após um estudo empírico, que é possível reaproveitar experiências anteriores no processo de identificação de mutantes equivalentes. Na etapa de similaridade, comparou-se a sintaxe entre os programas. A técnica utilizada também demonstrou que a similaridade entre programas pode auxiliar na escolha correta de técnicas de redução de custo do teste mutação.

Também como forma de auxiliar no processo de identificação de mutantes equivalentes, Jammalamadaka e Parveen (2022) utilizaram similaridade semântica de programas em uma estratégia chamada de *Hybrid Wavelet Convolutional Rain Optimization* (HWCRO). Na estratégia apresentada, recursos como similaridade e entropia de informações (obtida por meio das informações dos operadores de mutação usados nos programas mutantes) entre códigos e mutantes são extraídos, e então utilizados em um classificador de rede neural convolucional *wavelet* (do inglês, *wavelet convolutional neural network*) (wCNN). Rede neural convolucional, geralmente utilizado em reconhecimento de imagens, é um algoritmo de Aprendizado Profundo (do inglês, *Deep Learning*) no qual, após receber algumas entradas e atribuir importâncias para características identificadas nelas, consegue-se diferenciar uma entrada da outra. A função de *wavelet* tem como papel reduzir a dimensionalidade dos recursos extraídos, visando otimizar o processo proposto pelos autores. Para a etapa de similaridade, a estratégia HWCRO utiliza *Greedy String Tiling* (GST), que na prática percorre os pares de *strings* dos códigos-fontes para identificar similaridade entre eles. Por fim, o classificador proposto pelos autores diz se um o mutante é equivalente (ou não) ao programa original. Nos experimentos realizados, os resultados alcançaram uma precisão de aproximadamente 85%.

Esses dois últimos estudos utilizaram a similaridade de programas como forma

de auxiliar na identificação de mutantes equivalentes. Uma diferença fundamental entre este trabalho de mestrado e os estudos de Kintis e Malevris (2013) e Jammalamadaka e Parveen (2022) é com relação ao objetivo a ser alcançado com a aplicação da similaridade de programas. Enquanto as abordagens apresentadas pelos autores visam comparar mutantes com o programa original e entre si para a identificação de mutantes equivalentes, neste trabalho de mestrado comparam-se programas que já foram testados com teste de mutação com programas ainda não testados com teste de mutação, isso como forma de reaproveitar experiências anteriores de técnicas redução de custo do teste de mutação aplicadas nestes programas já testados.

Dallilo, Pizzoleto e Ferrari (2019) e Pizzoleto et al. (2020) realizaram um estudo para avaliar as características internas do programa como preditores de pontuação nos testes de mutação. Foi utilizado o algoritmo de *clustering* X-Means (PELLEG; MOORE, 2000), no qual a proposta foi identificar um grupo R de programas testados que pudessem ser usados como base para testar um novo programa usando mutação a custo reduzido. Nos experimentos, nos quais foi aplicada a técnica *One-Op* (UNTCH, 2009), os autores utilizaram 38 programas e observaram que aqueles que ficaram no mesmo *cluster* demonstraram ter, na maioria dos casos, os mesmos operadores de impacto no teste de mutação. Deste modo, neste trabalho constatou-se que por meio da similaridade de programas computada com base em características internas, foi possível reaproveitar experiências adquiridas em programas já testados com mutação em programas ainda não testados com mutação. No estudo de Dallilo, Pizzoleto e Ferrari (2019) implementou-se uma versão inicial do *framework* $SiMut$, o mesmo que foi implementado de forma mais robusta neste trabalho de mestrado acoplando-se diferentes técnicas de abstrações e similaridades.

6.2 Contribuições

Comparado com pesquisas anteriores (DALLILO; PIZZOLETO; FERRARI, 2019; PIZZOLETO et al., 2020) e com o estado da arte em teste de mutação, este trabalho traz as seguintes contribuições:

- Apresenta uma implementação do *framework* $SiMut$ que automatiza uma abordagem baseada em similaridade de programas para ajudar a reduzir o custo do teste de mutação;
- Experimenta 20 configurações variadas possíveis do $SiMut$ e relata os resultados alcançados;
- Aponta as configurações promissoras do $SiMut$ para a previsão dos operadores de mutação com maiores escores de mutação para classes não testadas.

Projetos grandes, e naturalmente complexos, são inviáveis para aplicação de teste de mutação em todas as suas classes. Mesmo assim, para algumas classes mais críticas, o teste de mutação pode ser aplicado com custo reduzido baseando-se em resultados obtidos neste trabalho, em que a configuração que obteve o melhor resultado foi *Código processado / mdist - Jaccard*.

6.3 Limitações

Os resultados coletados e o *framework* implementado possuem algumas limitações que podem interferir ou variar os resultados apresentados. Na sequência listam-se algumas dessas limitações.

- Embora o *framework SiMut* implementado tenha sido preparado para tratar códigos-fontes Java com múltiplas classes ou *inner classes* em um mesmo arquivo, nos experimentos realizados neste trabalho não se utilizou nenhum código-fonte com essas características.
- O grau de confiabilidade dos resultados coletados estão limitados ao funcionamento correto das ferramentas acopladas (ProGuard, mdist, Weka, JPlag e CKJM extended) ao *framework* implementado.
- Embora durante a implementação do *framework SiMut* testes foram realizados, os resultados coletados estão limitados ao acoplamento feito de forma correta das ferramentas ao *framework*. Os testes realizados envolveram comparação dos resultados gerados utilizando-se o *framework* implementado com os resultados gerados diretamente pelas ferramentas acopladas. Além disso, diversos resultados, entre todos os gerados, foram escolhidos de forma aleatória para serem analisados desde a abstração computada até a formação do grupo de referência G.
- Por utilizarem-se apenas programas simples nos experimentos, neste trabalho não se investigou uma relação entre tempo de execução e efetividade entre cada umas das combinações executadas. Nota-se que de acordo com a quantidade de programas, ou complexidade deles, o tempo de execução do *framework SiMut* pode aumentar consideravelmente.
- Até a data da escrita deste trabalho, o módulo responsável pela redução de custo não funciona em outros sistemas operacionais além do Microsoft Windows.
- A efetividade dos resultados de teste de mutação coletados nos experimentos também se limita à qualidade da base de dados de artefatos (PIZZOLETO; GUARNIERI; FERRARI, 2021) utilizada neste trabalho.

6.4 Trabalhos Futuros

- Ampliar o banco de dados de programas com programas mais complexos, permitindo assim reproduzir os experimentos com um conjunto de dados maior.
- Explorar outras técnicas de redução de custo (além da Mutação Seletiva) como, por exemplo, Operadores Essenciais (BARBOSA; MALDONADO; VINCENZI, 2001) e técnicas que classificam mutantes com base em algoritmos de aprendizado de máquina.
- Criar uma interface gráfica no *framework SiMut* que permita a criação do arquivo de execução XML de maneira facilitada para o testador, apresentando todas as técnicas disponíveis para a etapa de abstração e similaridade.
- Permitir que o usuário possa indicar, por meio de um atributo, o comando de execução Maven necessário para compilar cada um dos programas inseridos no arquivo de execução XML do *framework SiMut*.
- Utilizando um banco de dados de programas com programas mais complexos, realizar um estudo sobre o tempo de execução de cada configuração disponível no *SiMut*, criando uma relação entre efetividade e tempo de execução.
- Visando integrar todas as etapas do *framework SiMut* em um único programa, migrar o módulo de cálculo de redução de custo, atualmente escrito em C#, para a linguagem Java e integrá-lo ao *framework* implementado.

6.5 Publicações Resultantes deste Trabalho

Durante a realização deste trabalho, publicaram-se dois estudos:

- "*Definition of a Knowledge Base Towards a Benchmark for Experiments with Mutation Testing*" (PIZZOLETO; GUARNIERI; FERRARI, 2021): neste estudo, objetivou-se criar uma base de dados padronizada para ser utilizada como recurso em etapas de verificação e validação de teste de software, mais especificamente no domínio de teste de mutação. O banco de dados apresentado neste estudo inclui aproximadamente 2.000 classes, 50.000 casos de teste de 195.000 mutantes para 5 projetos escritos na linguagem Java. Recursos desse banco de dados, mais especificamente do projeto "simplePrograms', foram utilizados neste trabalho de mestrado.
- "*An Automated Framework for Cost Reduction of Mutation Testing Based on Program Similarity*" (GUARNIERI; PIZZOLETO; FERRARI, 2022): este estudo contém o mesmo conteúdo presente nos capítulos deste trabalho, entretanto, de forma mais resumida e objetiva.

Referências

ANANTH, P. et al. Optimizing obfuscation: avoiding barrington's theorem. In: *Proceedings of the 2014 ACM SIGSAC Conference on Computer and Communications Security*. [S.l.: s.n.], 2014. p. 646–658. Citado na página 40.

BANSIYA, J. Class cohesion metric for object oriented designs. *Journal of Object-Oriented Programming*, v. 11, n. 8, p. 47–52, 1999. Citado na página 38.

BANSIYA, J.; DAVIS, C. A hierarchical model for object-oriented design quality assessment. *IEEE Transactions on Software Engineering*, v. 28, n. 1, p. 4–17, 2002. Citado 2 vezes nas páginas 38 e 54.

BARBOSA, E. F.; MALDONADO, J. C.; VINCENZI, A. M. R. Toward the Determination of Sufficient Mutant Operators for C. *Soft. Testing, Verification and Reliability*, John Wiley & Sons, v. 11, n. 2, 2001. Citado 2 vezes nas páginas 32 e 93.

CHIDAMBER, S. R.; KEMERER, C. F. A metrics suite for object oriented design. *IEEE Trans. on Soft. Eng.*, IEEE, v. 20, n. 6, 1994. ISSN 0098-5589. Citado 4 vezes nas páginas 36, 37, 54 e 90.

CILIBRASI, R.; VITÁNYI, P. M. Clustering by compression. *IEEE Transactions on Information theory*, IEEE, v. 51, n. 4, p. 1523–1545, 2005. Citado na página 43.

COUSOT, P.; COUSOT, R. Abstract interpretation: A unified lattice model for static analysis of programs by construction or approximation of fixpoints. In: *Proceedings of the 4th ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages*. New York, NY, USA: Association for Computing Machinery, 1977. (POPL '77), p. 238–252. ISBN 9781450373500. Disponível em: <https://doi.org/10.1145/512950.512973>. Citado na página 36.

CRUZ, R. C.; ELER, M. M. Using a Cluster Analysis Method for Grouping Classes According to their inferred Testability: An Investigation of CK Metrics, Code Coverage and Mutation Score. In: *Proc. the 36th Internl. Conf. of the Chilean Computer Science Society (SCCC)*. [S.l.]: Chilean Comp. Science Society, 2017. Citado 5 vezes nas páginas 36, 68, 86, 89 e 90.

DALLILO, L. D.; PIZZOLETO, A. V.; FERRARI, F. C. An evaluation of internal program metrics as predictors of mutation operator score. In: *Proceedings of the IV Brazilian Symposium on Systematic and Automated Software Testing*. New York, NY, USA: Association for Computing Machinery, 2019. (SAST 2019), p. 12–21. ISBN 9781450376488. Disponível em: <https://doi.org/10.1145/3356317.3356323>. Citado 11 vezes nas páginas 24, 25, 26, 33, 36, 65, 66, 68, 73, 86 e 91.

DELAMARO, M. E.; MALDONADO, J. C.; JINO, M. *Introdução ao teste de software*. [S.l.]: Campus-Elsevier, 2007. ISBN 9788535226348. Citado 4 vezes nas páginas 23, 27, 28 e 30.

DEMILLO, R. A.; LIPTON, R. J.; SAYWARD, F. G. Hints on Test Data Selection: Help for the Practicing Programmer. *IEEE Computer*, IEEE, v. 11, n. 4, 1978. Citado 2 vezes nas páginas 23 e 29.

DEMPSTER, A. P.; LAIRD, N. M.; RUBIN, D. B. Maximum likelihood from incomplete data via the EM algorithm. *Journal of the Royal Statistical Society: Series B (Methodological)*, Wiley Online Library, v. 39, n. 1, p. 1–22, 1977. Citado 2 vezes nas páginas 47 e 56.

FELDT, R. et al. Test set diameter: Quantifying the diversity of sets of test cases. In: IEEE. *2016 IEEE International Conference on Software Testing, Verification and Validation (ICST)*. [S.l.], 2016. p. 223–233. Citado 2 vezes nas páginas 43 e 58.

FELDT, R. et al. Searching for cognitively diverse tests: Towards universal test diversity metrics. In: IEEE. *2008 IEEE International Conference on Software Testing Verification and Validation Workshop*. [S.l.], 2008. p. 178–186. Citado na página 58.

FERRARI, F. C.; PIZZOLETO, A. V.; OFFUTT, A. J. A Systematic Review of Cost Reduction Techniques for Mutation Testing: Preliminary Results. In: *Proc. the 13th Internl. Workshop on Mutation Analysis (Mutation)*. [S.l.]: IEEE, 2018. Citado na página 24.

GUARNIERI, G. F.; PIZZOLETO, A. V.; FERRARI, F. C. An Automated Framework for Cost Reduction of Mutation Testing Based on Program Similarity. In: *17th International Workshop on Mutation Analysis (Mutation)*. [S.l.: s.n.], 2022. (to appear). Citado 11 vezes nas páginas 33, 34, 50, 61, 62, 68, 73, 81, 83, 84 e 93.

HAMLET, R. G. Testing Programs with the Aid of a Compiler. *IEEE Trans. on Soft. Eng.*, IEEE, v. 3, n. 4, jul. 1977. ISSN 0098-5589. Citado 2 vezes nas páginas 23 e 29.

HENDERSON-SELLERS, B. *Object-oriented metrics: measures of complexity*. [S.l.]: Prentice-Hall, Inc., 1995. Citado na página 38.

HOWDEN, W. E. Weak Mutation Testing and Completeness of Test Sets. *IEEE Trans. on Soft. Eng.*, IEEE, v. 8, n. 4, 1982. ISSN 0098-5589. Citado na página 32.

JACCARD, P. Étude comparative de la distribution florale dans une portion des alpes et des jura. *Bull Soc Vaudoise Sci Nat*, v. 37, p. 547–579, 1901. Citado 2 vezes nas páginas 42 e 58.

JAMMALAMADAKA, K.; PARVEEN, N. Equivalent Mutant Identification Using Hybrid Wavelet Convolutional Rain Optimization. *Software: Practice and Experience*, v. 52, n. 2, 2022. Citado 2 vezes nas páginas 90 e 91.

JIA, Y.; HARMAN, M. An Analysis and Survey of the Development of Mutation Testing. *IEEE Trans. on Soft. Eng.*, IEEE, v. 37, n. 5, 2011. ISSN 0098-5589. Citado na página 24.

JURECZKO, M.; SPINELLIS, D. Using object-oriented design metrics to predict software defects. In: *Proceedings of the 5th International Conference on Dependability of Computer Systems (DepCoS–RELCOMEX)*. Wroclaw, Poland: Oficyna Wydawnicza Politechniki Wroclawskiej, 2010. p. 69–81. Citado na página 54.

KINTIS, M.; MALEVRIS, N. Identifying More Equivalent Mutants via Code Similarity. In: *Proc. the 20th Asia-Pacific Software Engineering Conference (APSEC)*. [S.l.]: IEEE, 2013. Citado 3 vezes nas páginas 25, 90 e 91.

KURTZ, B. et al. Analyzing the Validity of Selective Mutation with Dominator Mutants. In: *Proc. the 24th Internl. Symposium on Foundations of Software Engineering (FSE)*. [S.l.]: ACM, 2016. ISBN 978-1-4503-4218-6. Citado na página 24.

LACERDA, J. T. S.; FERRARI, F. C. Towards the Establishment of a Sufficient Set of Mutation Operators for AspectJ Programs. In: *Proc. the 8th Brazilian Workshop on Systematic and Automated Software Testing (SAST)*. [S.l.]: Brazilian Computer Society, 2014. Citado 2 vezes nas páginas 24 e 32.

LEVENSHTEIN, V. I. et al. Binary codes capable of correcting deletions, insertions, and reversals. In: SOVIET UNION. *Soviet physics doklady*. [S.l.], 1966. v. 10, n. 8, p. 707–710. Citado 2 vezes nas páginas 42 e 58.

LI, W. Software product metrics. *IEEE Potentials*, v. 18, n. 5, p. 24–27, 2000. Citado na página 36.

LINN, C.; DEBRAY, S. Obfuscation of executable code to improve resistance to static disassembly. In: *Proceedings of the 10th ACM conference on Computer and communications security*. [S.l.: s.n.], 2003. p. 290–299. Citado na página 40.

MARTIN, R. Oo design quality metrics. *An analysis of dependencies*, v. 12, n. 1, p. 151–170, 1994. Citado na página 37.

McCabe, T. J. A Complexity Measure. *IEEE Trans. on Soft. Eng.*, SE-2, n. 4, 1976. ISSN 2326-3881. Citado 3 vezes nas páginas 39, 54 e 67.

MORELL, L. J. A Theory of Fault-Based Testing. *IEEE Trans. on Soft. Eng.*, IEEE, v. 16, n. 8, 1990. ISSN 0098-5589. Citado na página 29.

MYERS, G. J.; BADGETT, T.; SANDLER, C. *The Art of Software Testing*. 3rd. ed. [S.l.]: John Wiley & Sons, 2011. Citado 5 vezes nas páginas 23, 27, 28, 29 e 39.

OFFUTT, A. J. Investigations of the software testing coupling effect. *ACM Transactions on Software Engineering and Methodology (TOSEM)*, ACM New York, NY, USA, v. 1, n. 1, p. 5–20, 1992. Citado na página 29.

OFFUTT, A. J.; ROTHERMEL, G.; ZAPF, C. An Experimental Evaluation of Selective Mutation. In: *Proc. the 15th Internl. Conference on Software Engineering (ICSE)*. [S.l.]: IEEE, 1993. Citado 5 vezes nas páginas 24, 32, 49, 63 e 77.

OFFUTT, A. J.; UNTCH, R. H. Mutation 2000: Uniting the Orthogonal. In: *Proc. the Mutation 2000 Symposium*. [S.l.]: Kluwer Academic Publishers, 2000. ISSN 1386-2944. Citado 2 vezes nas páginas 24 e 32.

OMAR, E.; GHOSH, S. An Exploratory Study of Higher Order Mutation Testing in Aspect-Oriented Programming. In: *Proc. the 23th Internl. Symposium on Software Reliability Engineering (ISSRE)*. [S.l.]: IEEE, 2012. Citado 2 vezes nas páginas 24 e 32.

PAPADAKIS, M. et al. Trivial Compiler Equivalence: A Large Scale Empirical Study of a Simple, Fast and Effective Equivalent Mutant Detection Technique. In: *Proc. the 37th Internl. Conference on Software Engineering (ICSE)*. [S.l.]: ACM, 2015. Citado na página 24.

PAPADAKIS, M. et al. Mutation Testing Advances: An Analysis and Survey. In: MEMON, A. M. (Ed.). *Advances in Computers*. [S.l.]: Elsevier, 2019. v. 112. Citado na página 32.

PELLEG, D.; MOORE, A. W. X-means: Extending K-means with Efficient Estimation of the Number of Clusters. In: *Proc. the 17th Internl. Conference on Machine Learning*. [S.l.]: Morgan Kaufmann Publishers, 2000. ISBN 1-55860-707-2. Citado 4 vezes nas páginas 25, 46, 56 e 91.

PIZZOLETO, A. V. et al. SiMut: Exploring Program Similarity to Support the Cost Reduction of Mutation Testing. In: *15th International Workshop on Mutation Analysis (Mutation)*. [S.l.: s.n.], 2020. ISBN 978-1-5386-6352-3. Citado 9 vezes nas páginas 24, 25, 26, 32, 33, 36, 49, 65 e 91.

PIZZOLETO, A. V. et al. A Systematic Literature Review of Techniques and Metrics to Reduce the Cost of Mutation Testing. *Journal of Systems and Software*, v. 157, 2019. ISSN 0164-1212. Citado 2 vezes nas páginas 24 e 33.

PIZZOLETO, A. V.; GUARNIERI, G. F.; FERRARI, F. C. Definition of a Knowledge Base Towards a Benchmark for Experiments with Mutation Testing. In: *SBES-IIER*. [S.l.: s.n.], 2021. Citado 6 vezes nas páginas 63, 66, 67, 86, 92 e 93.

POLO, M.; PIATTINI, M.; GARCÍA-RODRÍGUEZ, I. Decreasing the Cost of Mutation Testing with Second-Order Mutants. *Soft. Testing, Verification and Reliability*, John Wiley & Sons, v. 19, n. 2, 2009. Citado na página 24.

PRECHELT, L. et al. Finding plagiarisms among a set of programs with jplag. *J. UCS*, v. 8, n. 11, p. 1016, 2002. Citado 2 vezes nas páginas 41 e 59.

ROJAS, J. M. et al. Combining Multiple Coverage Criteria in Search-Based Unit Test Generation. In: *Search-Based Software Engineering*. Cham: [s.n.], 2015. Citado na página 67.

SCHLEIMER, S.; WILKERSON, D. S.; AIKEN, A. Winnowing: local algorithms for document fingerprinting. In: *Proceedings of the 2003 ACM SIGMOD international conference on Management of data*. [S.l.: s.n.], 2003. p. 76–85. Citado na página 41.

SIAMI-NAMIN, A.; ANDREWS, J. H.; MURDOCH, D. J. Sufficient Mutation Operators for Measuring Test Effectiveness. In: *Proc. the 30th Internl. Conference on Software Engineering (ICSE)*. [S.l.]: ACM, 2008. Citado 2 vezes nas páginas 24 e 32.

TANG, M.-H.; KAO, M.-H.; CHEN, M.-H. An empirical study on object-oriented metrics. In: IEEE. *Proceedings sixth international software metrics symposium (Cat. No. PR00403)*. [S.l.], 1999. p. 242–249. Citado na página 39.

TITCHEU CHEKAM, T. et al. Selecting Fault Revealing Mutants. *Empirical Software Engineering*, Springer, v. 25, p. 434–487, 2020. Citado 3 vezes nas páginas 24, 25 e 89.

UNTCH, R. H. Mutation-based Software Testing Using Program Schemata. In: *Proc. the 30th Annual Southeast Regional Conference (ACM-SE)*. [S.l.]: ACM, 1992. Citado na página 32.

UNTCH, R. H. On Reduced Neighborhood Mutation Analysis Using a Single Mutagenic Operator. In: *Proc. the 47th Annual Southeast Regional Conference (ACM-SE)*. [S.l.]: ACM, 2009. Citado 3 vezes nas páginas 32, 77 e 91.

WALENSTEIN, A. et al. Similarity in Programs. In: *Duplication, Redundancy, and Similarity in Software*. [S.l.: s.n.], 2007. (Dagstuhl Seminar Proceedings). ISSN 1862-4405. Citado 2 vezes nas páginas 40 e 41.

WITTEN, I. H.; FRANK, E. Data mining: practical machine learning tools and techniques with java implementations. *Acm Sigmod Record*, ACM New York, NY, USA, v. 31, n. 1, p. 76–77, 2002. Citado na página 56.

WOHLIN, C. et al. *Experimentation in Software Engineering*. [S.l.]: Springer, 2012. ISBN 3642290434. Citado na página 86.

WU, X. et al. Top 10 algorithms in data mining. *Knowledge and information systems*, Springer, v. 14, n. 1, p. 1–37, 2008. Citado na página 45.

APÊNDICE A – XML com todas as possibilidades de configuração

Figura 17 – XML com todas as possibilidades de configuração.

```
<?xml version="1.0" encoding="UTF-8" standalone="no"?>
<project description="SiMut_Possibilidades">
  <programs>
    <program id="1" name="programaJaTestado" />
    <program path="c:/programas/novoPrograma" name="novoPrograma" />
    ...
  </programs>
  <abstractions>
    <abstraction name="OriginalSourceCode" />
    <abstraction name="ProGuard">
      <options>
        <option name="obfuscate|optimize|shrink" />
        ...
      </options>
    </abstraction>
    <abstraction name="CKJM-Extended">
      <options>
        <option name="wmc|dit|noc|cbo|rfc|lcom|ca|ce|npm|lcom3|loc|dam|moa|mfa|cam|ic|cbm|amc|acc" />
        ...
      </options>
    </abstraction>
  </abstractions>
  <similarities groups="n">
    <sources>
      <source name="OriginalSourceCode|ProGuard|CKJM-Extended" />
      ...
    </sources>
    <nprograms>
      <nprogram name="novoPrograma">
        <class name="novaClasse" />
        ...
      </nprogram>
      ...
    </nprograms>
    <similarity algorithm="ncd|jaccard|levenshtein" name="mdist">
      <group level="0" percentage="0...100" />
      ...
      <group level="n-1" percentage="0...100" />
    </similarity>
    <similarity algorithm="xmeans" distance="euclidean|manhattan" name="weka" recluster="0...100">
      <group level="0" />
      ...
      <group level="n-1" />
    </similarity>
    <similarity algorithm="em" name="weka" recluster="0...100">
      <group level="0" />
      ...
      <group level="n-1" />
    </similarity>
    <similarity name="jplag">
      <group level="0" percentage="0...100" />
      ...
      <group level="n-1" percentage="0...100" />
    </similarity>
  </similarities>
</project>
```

Fonte: Produzido pelo autor.