UNIVERSIDADE DE SÃO PAULO
ESCOLA DE ARTES, CIÊNCIAS E HUMANIDADES
PROGRAMA DE PÓS-GRADUAÇÃO EM SISTEMAS DE INFORMAÇÃO

DAVI SILVA RODRIGUES

**TAIGA: uma abordagem para geração de dados de teste por meio de
algoritmo genético para programas de processamento de imagens**

São Paulo

2018

DAVI SILVA RODRIGUES

**TAIGA: uma abordagem para geração de dados de teste por meio de
algoritmo genético para programas de processamento de imagens**

Dissertação apresentada à Escola de Artes,
Ciências e Humanidades da Universidade de
São Paulo para obtenção do título de Mestre
em Ciências pelo Programa de Pós-graduação
em Sistemas de Informação.

Área de concentração: Metodologia e
Técnicas da Computação

Versão corrigida contendo as alterações
solicitadas pela comissão julgadora em 24
de novembro de 2017. A versão original
encontra-se em acervo reservado na Biblio-
teca da EACH-USP e na Biblioteca Digital
de Teses e Dissertações da USP (BDTD), de
acordo com a Resolução CoPGr 6018, de 13
de outubro de 2011.

Orientador: Profa. Dra. Fátima de Lourdes
dos Santos Nunes Marques

São Paulo

2018

Autorizo a reprodução e divulgação total ou parcial deste trabalho, por qualquer meio convencional ou eletrônico, para fins de estudo e pesquisa, desde que citada a fonte.

CATALOGAÇÃO-NA-PUBLICAÇÃO

(Universidade de São Paulo. Escola de Artes, Ciências e Humanidades. Biblioteca)
CRB-8 4625

Rodrigues, Davi Silva

TAIGA : uma abordagem para geração de dados de teste por meio de algoritmo genético para programas de processamento de imagens / Davi Silva Rodrigues ; orientadora, Fátima de Lourdes dos Santos Nunes Marques. – 2017.

127 f. : il

Dissertação (Mestrado em Ciências) - Programa de Pós-Graduação em Sistemas de Informação, Escola de Artes, Ciências e Humanidades, Universidade de São Paulo.

Versão corrigida

1. Teste e avaliação de software. 2. Processamento de imagens.
3. Algoritmos genéticos. I. Marques, Fátima de Lourdes dos Santos Nunes, orient. II. Título.

CDD 22.ed.– 005.14

Dissertação de autoria de Davi Silva Rodrigues, sob o título **“TAIGA: uma abordagem para geração de dados de teste por meio de algoritmo genético para programas de processamento de imagens”**, apresentada à Escola de Artes, Ciências e Humanidades da Universidade de São Paulo, para obtenção do título de Mestre em Ciências pelo Programa de Pós-graduação em Sistemas de Informação, na área de concentração Metodologia e Técnicas da Computação, aprovada em 24 de novembro de 2017 pela comissão julgadora constituída pelos doutores:

Profa. Dra. Fátima de Lourdes dos Santos Nunes Marques

Universidade de São Paulo

Presidente

Prof. Dr. Auri Marcelo Rizzo Vincenzi

Universidade Federal de São Carlos

Prof. Dr. Marcel Parolin Jackowski

Universidade de São Paulo

Prof. Dr. Ivandré Paraboni

Universidade de São Paulo

Agradecimentos

Agradeço a Deus. Agradeço aos meus pais e à minha irmã por me apoiarem constantemente, apesar de todas as incertezas.

Agradeço à minha orientadora, professora Fátima, pela companhia, pelos conselhos, pela compreensão e por sempre me incentivar a ir além.

Agradeço ao professor Delamaro por sua grande contribuição no desenvolvimento deste trabalho.

Resumo

RODRIGUES, Davi Silva. **TAIGA**: uma abordagem para geração de dados de teste por meio de algoritmo genético para programas de processamento de imagens. 2018. 127 f. Dissertação (Mestrado em Ciências) – Escola de Artes, Ciências e Humanidades, Universidade de São Paulo, São Paulo, 2017.

As atividades de teste de *software* são de crescente importância devido à maciça presença de sistemas de informação em nosso cotidiano. Programas de Processamento de Imagens (PI) têm um domínio de entrada bastante complexo e, por essa razão, o teste tradicional realizado com esse tipo de programa, conduzido majoritariamente de forma manual, é uma tarefa de alto custo e sujeita a imperfeições. No teste tradicional, em geral, as imagens de entrada são construídas manualmente pelo testador ou selecionadas aleatoriamente de bases de imagens, muitas vezes dificultando a revelação de defeitos no *software*. A partir de um mapeamento sistemático da literatura realizado, foi identificada uma lacuna no que se refere à geração automatizada de dados de teste no domínio de imagens. Assim, o objetivo desta pesquisa é propor uma abordagem — denominada TAIGA (*Test imAge generatIon by Genetic Algorithm*) — para a geração de dados de teste para programas de PI por meio de algoritmo genético. Na abordagem proposta, operadores genéticos tradicionais (mutação e *crossover*) são adaptados para o domínio de imagens e a função *fitness* é substituída por uma avaliação de resultados provenientes de teste de mutação. A abordagem TAIGA foi validada por meio de experimentos com oito programas de PI distintos, nos quais observaram-se ganhos de até 38,61% em termos de *mutation score* em comparação ao teste tradicional. Ao automatizar a geração de dados de teste, espera-se conferir maior qualidade ao desenvolvimento de sistemas de PI e contribuir com a diminuição de custos com as atividades de teste de *software* neste domínio.

Palavras-chave: Geração de dados de teste. Geração de imagens de teste. Processamento de imagens. Teste de software. Algoritmos genéticos. Teste evolutivo. Teste de mutação. Mutation score.

Abstract

RODRIGUES, Davi Silva. **TAIGA**: an Approach to Test Image Generation for Image Processing Programs Using Genetic Algorithm. 2018. 127 p. Dissertation (Master of Science) – School of Arts, Sciences and Humanities, University of São Paulo, São Paulo, 2017.

The massive presence of information systems in our lives has been increasing the importance of software test activities. Image Processing (IP) programs have very complex input domains and, therefore, the traditional testing for this kind of program is a highly costly and vulnerable to errors task. In traditional testing, usually, testers create images by themselves or they execute random selection from images databases, which can make it harder to reveal faults in the software under test. In this context, a systematic mapping study was conducted and a gap was identified concerning the automated test data generation in the images domain. Thus, an approach for generating test data for IP programs by means of genetic algorithms was proposed: TAIGA — **T**est **imA**ge generat**I**on by **G**enetic **A**lgorithm. This approach adapts traditional genetic operators (mutation and crossover) to the images domain and replaces the fitness function by the evaluation of the results of mutation testing. The proposed approach was validated by the execution of experiments involving eight distinct IP programs. TAIGA was able to provide up to 38.61% increase in mutation score when compared to the traditional testing for IP programs. It's expected that the automation of test data generation elevates the quality of image processing systems development and reduces the costs of software test activities in the images domain.

Keywords: Test data generation. Test image generation. Image processing. Software test. Genetic algorithms. Evolutionary test. Mutation testing. Mutation score.

Lista de figuras

Figura 1 – Grafo de Fluxo de Controle (GFC) do programa <i>Identifier</i>	22
Figura 2 – Fluxograma de um algoritmo genético	32
Figura 3 – Cromossomo em um algoritmo genético padrão com codificação binária	33
Figura 4 – Roleta criada a partir da população descrita na Tabela 1	35
Figura 5 – <i>Crossover</i> de um único ponto entre dois indivíduos	36
Figura 6 – Mutação de um gene de um cromossomo binário	37
Figura 7 – Estudos primários aceitos por base de dados	40
Figura 8 – Síntese da condução do mapeamento sistemático	41
Figura 9 – Distribuição dos estudos aceitos entre as categorias apresentadas	45
Figura 10 – Distribuição dos estudos aceitos por ano e por categoria	46
Figura 11 – Relação entre técnicas de teste e formas de representação de programas. Eixo horizontal: Grafo de Fluxo de Controle (CFG — <i>Control Flow Graph</i>), <i>Combination Index Table</i> , código-fonte, DFG (<i>Data Flow Graph</i>), grafo, máquina de estados, requisitos de <i>software</i> , tabela de variáveis e de controles e <i>Unified Modelling Language</i>	48
Figura 12 – Relação entre técnicas de teste e categorias de estudos. Eixo horizontal: algoritmos híbridos, arquitetura, função <i>fitness</i> , população e representação dos indivíduos	49
Figura 13 – Relação entre categorias de estudos e métricas de avaliação das técnicas de geração de dados de teste por meio de AGs. Eixo horizontal: algoritmos híbridos, arquitetura, função <i>fitness</i> , população e representação dos indivíduos	58
Figura 14 – Procedimentos metodológicos utilizados neste trabalho	64
Figura 15 – Imagens de teste iniciais utilizadas nas etapas de desenvolvimento e de validação da abordagem	69
Figura 16 – Arquitetura da abordagem TAIGA	72
Figura 17 – Fluxograma do AG adaptado para o domínio de imagens utilizado na abordagem TAIGA	73
Figura 18 – Classe <code>Image</code> : codificação das imagens processadas pelo AG	76

Figura 19 – Operador de mutação da abordagem TAIGA, com parâmetro genético $mutation\ parts = 2$: (a) imagem original, (b) imagem modificada	78
Figura 20 – Operador de <i>crossover</i> horizontal da abordagem TAIGA: (a) imagem pai 1, (b) imagem pai 2, (c) imagem resultante	80
Figura 21 – Operador de <i>crossover</i> vertical da abordagem TAIGA: (a) imagem pai 1, (b) imagem pai 2, (c) imagem resultante	80
Figura 22 – Fluxograma da função <i>fitness</i> utilizada na abordagem TAIGA	81
Figura 23 – Exemplos de imagens de teste geradas pela abordagem TAIGA para os programas (a) Bordas, (b) Mediana, (c) <i>Fingerprint</i> e (d) Esqueletização; a definição dos programas foi apresentada na Tabela 6	84
Figura 24 – Exemplo de relatório gerado pela abordagem TAIGA após a geração de imagens de teste para o programa Dilatação	85
Figura 25 – Gráfico dos tempos de execução médios (em minutos) observados nos 672 experimentos conduzidos com a abordagem TAIGA e o teste tradicional de programas de PI	103
Figura 26 – Gráfico dos números médios de gerações do AG observados nos 640 experimentos realizados com a abordagem TAIGA	105
Figura 27 – Módulo de geração e visualização de mutantes da ferramenta <i>muJava</i> .	126
Figura 28 – Módulo de execução e visualização de mutantes da ferramenta <i>muJava</i> .	127

Lista de algoritmos

Algoritmo 1 – Algoritmo para a mutação de uma imagem <code>img</code> na abordagem TAIGA . . .	79
Algoritmo 2 – Trecho do programa Bordas	100
Algoritmo 3 – Trecho de mutante do programa Bordas	100
Algoritmo 4 – Trecho do programa Dilatação	102
Algoritmo 5 – Trecho de mutante do programa Dilatação	102

Lista de tabelas

Tabela 1 – Cálculo de valor esperado para quatro indivíduos	34
Tabela 2 – Estudos primários incluídos na fase de extração do MS — teste funcional	42
Tabela 3 – Estudos primários incluídos na fase de extração do MS — teste de mutação	43
Tabela 4 – Estudos primários incluídos na fase de extração do MS — teste estrutural	43
Tabela 5 – Operadores de mutação da ferramenta <i>muJava</i> utilizados pela aborda- gem TAIGA	66
Tabela 6 – Programas de PI utilizados nas etapas de desenvolvimento e validação da abordagem TAIGA apresentados em ordem de complexidade (linhas de código)	69
Tabela 7 – Parâmetros dos programas de PI utilizados nas etapas de desenvolvi- mento e validação da abordagem TAIGA	70
Tabela 8 – Parâmetros genéticos utilizados pela abordagem TAIGA	74
Tabela 9 – Quantidade de experimentos e de execuções realizados na etapa de validação da abordagem TAIGA	87
Tabela 10 – Resultados dos experimentos realizados com 348 mutantes do programa Média: <i>mutation score</i>	89
Tabela 11 – Resultados dos experimentos realizados com 348 mutantes do programa Média: tempos de execução e diferença percentual (Δ)	89
Tabela 12 – Resultados dos experimentos realizados com 164 mutantes do programa Bordas: <i>mutation score</i>	90
Tabela 13 – Resultados dos experimentos realizados com 164 mutantes do programa Bordas: tempos de execução e diferença percentual (Δ)	90
Tabela 14 – Resultados dos experimentos realizados com 298 mutantes do programa Mediana: <i>mutation score</i>	91
Tabela 15 – Resultados dos experimentos realizados com 298 mutantes do programa Mediana: tempos de execução e diferença percentual (Δ)	91
Tabela 16 – Resultados dos experimentos realizados com 154 mutantes do programa Equalização: <i>mutation score</i>	92

Tabela 17 – Resultados dos experimentos realizados com 154 mutantes do programa	
Equalização: tempos de execução e diferença percentual (Δ)	92
Tabela 18 – Resultados dos experimentos realizados com 303 mutantes do programa	
Dilatação: <i>mutation score</i>	93
Tabela 19 – Resultados dos experimentos realizados com 303 mutantes do programa	
Dilatação: tempos de execução e diferença percentual (Δ)	93
Tabela 20 – Resultados dos experimentos realizados com 369 mutantes do programa	
Erosão: <i>mutation score</i>	94
Tabela 21 – Resultados dos experimentos realizados com 369 mutantes do programa	
Erosão: tempos de execução e diferença percentual (Δ)	94
Tabela 22 – Resultados dos experimentos realizados com 310 mutantes do programa	
Esqueletização: <i>mutation score</i>	95
Tabela 23 – Resultados dos experimentos realizados com 310 mutantes do programa	
Esqueletização: tempos de execução e diferença percentual (Δ)	95
Tabela 24 – Resultados dos experimentos realizados com 360 mutantes do programa	
<i>Fingerprint</i> : <i>mutation score</i>	96
Tabela 25 – Resultados dos experimentos realizados com 360 mutantes do programa	
<i>Fingerprint</i> : tempos de execução e diferença percentual (Δ)	96
Tabela 26 – Maiores ganhos de <i>mutation score</i> das imagens de teste geradas pela abordagem TAIGA	97
Tabela 27 – Ganhos de <i>mutation score</i> (MS) das imagens de teste geradas pela abordagem TAIGA a partir de imagens iniciais “mais realistas” para os programas de PI utilizados	98
Tabela 28 – Exemplos de trechos de programas mutantes equivalentes ao programa	
Bordas devido às validações implícitas presentes em APIs para imagens	101
Tabela 29 – Exemplos de trechos de mutantes equivalentes ao programa Dilatação	102
Tabela 30 – Quantidades de mutantes gerados pela ferramenta <i>muJava</i> , descartados, equivalentes e utilizados nos experimentos	104

Lista de abreviaturas e siglas

AG	Algoritmo Genético
API	<i>Application Programming Interface</i>
BPEL	<i>Business Process Execution Language</i>
CBIR	<i>Content-Based Image Retrieval</i>
CDG	<i>Control Dependence Graph</i>
DFG	<i>Data Flow Graph</i>
ES	Engenharia de <i>Software</i>
EV	Estratégias Evolutivas
ECJ	<i>Evolutionary Computation for Java</i>
GFC	Grafo de Fluxo de Controle
JGAP	<i>Java Genetic Algorithms Package</i>
MC/DC	<i>Modified Condition/Decision Coverage</i>
MS	Mapeamento Sistemático
MS	<i>Mutation Score</i>
PE	Programação Evolutiva
PG	Programação Genética
PI	Processamento de Imagens
RGB	<i>Red-Green-Blue</i>
SC	Sistemas Classificadores
TAIGA	<i>Test imAge generatIon by Genetic Algorithm</i>
UML	<i>Unified Modeling Language</i>
VV&T	Verificação, Validação e Teste

Sumário

1	Introdução	16
1.1	<i>Motivação e justificativa</i>	16
1.2	<i>Objetivos e hipótese</i>	17
1.3	<i>Procedimentos metodológicos</i>	18
1.4	<i>Organização do documento</i>	18
2	Fundamentação teórica	20
2.1	<i>Teste de software</i>	20
2.2	<i>Técnicas de teste de software</i>	20
2.2.1	Teste de mutação	23
2.3	<i>Processamento de imagens</i>	25
2.4	<i>Geração de dados de teste</i>	27
2.5	<i>Algoritmos genéticos</i>	28
2.5.1	Arquitetura dos algoritmos genéticos	30
2.5.2	População de cromossomos	32
2.5.3	Operadores genéticos	33
2.6	<i>Considerações finais</i>	37
3	Mapeamento sistemático	38
3.1	<i>Geração de dados de teste no domínio de imagens</i>	38
3.2	<i>Protocolo e condução do mapeamento sistemático</i>	39
3.3	<i>Análise global dos resultados obtidos</i>	41
3.4	<i>Uso de algoritmos genéticos para geração de dados de teste</i>	46
3.5	<i>Função fitness</i>	49
3.5.1	Teste funcional	49
3.5.2	Teste de mutação	50
3.5.3	Teste estrutural	51
3.6	<i>População</i>	53
3.6.1	Teste funcional	53
3.6.2	Teste estrutural	53
3.7	<i>Representação dos indivíduos</i>	54

3.7.1	Teste funcional	54
3.7.2	Teste estrutural	55
3.8	<i>Arquitetura do algoritmo genético</i>	56
3.9	<i>Algoritmos híbridos</i>	56
3.10	<i>Métricas de avaliação para técnicas de geração de dados de teste</i> . . .	57
3.11	<i>Discussão, desafios e oportunidades</i>	59
3.11.1	Adaptação ao problema	60
3.11.2	Dados complexos e sistemas grandes	61
3.11.3	Tempo de execução e métricas de avaliação	62
3.11.4	Ambientes virtuais tridimensionais	62
3.12	<i>Considerações finais</i>	63
4	Procedimentos metodológicos	64
4.1	<i>Revisão da literatura</i>	64
4.2	<i>Desenvolvimento da abordagem proposta</i>	65
4.2.1	Tecnologias utilizadas	65
4.2.2	Definição das adaptações realizadas no algoritmo genético	67
4.3	<i>Validação experimental</i>	68
4.4	<i>Considerações finais</i>	71
5	TAIGA — <i>Test imAge generatIon by Genetic Algorithm</i> . .	72
5.1	<i>Adaptação de algoritmo genético para o domínio de imagens</i>	72
5.1.1	Parâmetros genéticos	74
5.1.2	Codificação das soluções candidatas	75
5.1.3	Operador de mutação	78
5.1.4	Operadores de <i>crossover</i>	79
5.1.5	Função <i>fitness</i>	80
5.1.6	Condições de parada	83
5.2	<i>Imagens de teste geradas e métricas de avaliação obtidas</i>	84
5.3	<i>Considerações finais</i>	86
6	Resultados e discussão	87
6.1	<i>Experimentos realizados com programas de PI</i>	87
6.1.1	Programa Média	89

6.1.2	Programa Bordas	90
6.1.3	Programa Mediana	91
6.1.4	Programa Equalização	92
6.1.5	Programa Dilatação	93
6.1.6	Programa Erosão	94
6.1.7	Programa Esqueletização	95
6.1.8	Programa <i>Fingerprint</i>	96
6.2	<i>Discussão dos resultados</i>	97
6.2.1	Análise do <i>Mutation Score</i>	97
6.2.2	Tempo de Execução	102
6.2.3	Quantidade de gerações do AG	104
6.3	<i>Considerações finais</i>	105
7	Conclusões	106
7.1	<i>Limitações da abordagem</i>	107
7.2	<i>Contribuições</i>	108
7.3	<i>Trabalhos futuros</i>	109
	Referências¹	110
	 APÊNDICES	 120
	Apêndice A – Protocolo do mapeamento sistemático	121
	Apêndice B – Geração de programas mutantes com a ferramenta <i>muJava</i>	124

¹ De acordo com a Associação Brasileira de Normas Técnicas. NBR 6023.

1 Introdução

Sistemas de Informação estão presentes nos mais diversos setores da sociedade e, por esta razão, a qualidade de um sistema torna-se fundamental. A Engenharia de *Software* (ES), por meio de atividades de Verificação, Validação e Teste (VV&T), possibilita desenvolver sistemas com maior confiabilidade e menos erros (DELAMARO; MALDONADO; JINO, 2007).

De acordo com Delamaro, Maldonado e Jino (2007), o sucesso do teste de *software* consiste em identificar a maior parte dos defeitos — passos ou definições errados que são causados por enganos cometidos pelo programador — com o menor tempo e esforço possíveis. Idealmente, a atividade de teste deve utilizar todo o domínio de entrada do programa a fim de executá-lo e verificar as saídas produzidas. Entretanto, a cardinalidade do domínio inviabiliza esta estratégia para a maioria dos programas, tornando os custos da atividade de testes proibitivos. É necessário, portanto, utilizar os dados de teste capazes de revelar o maior número de defeitos dentro do menor tempo e com o menor esforço possíveis.

Processamento de Imagens (PI) é o nome dado ao conjunto de técnicas que permitem a captura, a representação e a transformação de imagens utilizando computadores (PEDRINI; SCHWARTZ, 2008).

No teste tradicional de programas de PI, usualmente conduzido de forma manual e não sistemática, os dados de teste são selecionados ou construídos também manualmente. Esta abordagem tradicional aumenta os custos e pode não ser eficaz, visto que as imagens utilizadas podem não revelar os defeitos existentes no programa. Um testador, por exemplo, terá menor esforço ao utilizar geração automática de imagens de teste em vez de selecionar imagens de uma base de imagens existente. Abordagens automatizadas podem gerar imagens com maiores chances de maximizar a descoberta de defeitos e, consequentemente, são alternativas para a redução de custos da atividade de teste (BASHIR; BANURI, 2008).

1.1 Motivação e justificativa

Estima-se que a atividade de teste corresponda à metade dos custos de desenvolvimento de sistemas (ALLI et al., 2010). Para programas de Processamento de Imagens (PI),

estes custos são maiores devido à complexidade inerente ao domínio de entrada (JAMEEL et al., 2014).

Segundo Delamaro, Maldonado e Jino (2007), a tarefa de geração de dados de teste é uma forma de automatizar o processo de seleção de elementos do domínio de entrada do programa que atendam aos critérios de teste definidos. Os dados de teste podem ser gerados de diferentes maneiras: aleatoriamente até que o critério de teste seja atendido; a partir de execuções simbólicas ou dinâmicas do programa; utilizando algoritmos genéticos, e a partir de técnicas que gerem casos sensíveis a defeitos (teste de mutação, teste de domínios, teste baseado em restrições, teste baseado em predicados, entre outros) (DELAMARO; MALDONADO; JINO, 2007).

Em um mapeamento sistemático da literatura sobre geração de dados de teste por meio de algoritmos genéticos, os estudos encontrados, em sua maioria, geram dados triviais (como números e *strings*) para o teste de programas de propósito geral de baixa complexidade, tais como determinação do maior entre dois números, classificação de triângulos etc. (HANH; BINH; TUNG, 2016; VARSHNEY; MEHROTRA, 2016; MANN et al., 2016; KHAN; AMJAD; SRIVASTAVA, 2016). Outros trabalhos geram dados triviais em domínios específicos para sistemas orientados a objetos (SILVA; VAN SOMEREN, 2010), sistemas de tempo real (NÚÑEZ et al., 2013), sistemas embarcados (VUDATHA et al., 2011) e de controle de *drones* (BETTS; PETTY, 2016). Nenhum dos trabalhos encontrados aborda a geração de dados de teste em domínios complexos, como os de imagens, sons e vídeos.

Portanto, existe uma lacuna na pesquisa de teste de programas de PI. O problema de gerar dados de teste é complexo e, no domínio de imagens, pouco explorado. Este trabalho visa a definir uma abordagem para a geração de dados de teste que seja mais efetiva do que o teste tradicional para programas de PI.

1.2 Objetivos e hipótese

O objetivo deste trabalho é propor, implementar e validar uma abordagem que gere um conjunto de imagens com a finalidade de constituir dados de testes eficazes para programas de PI.

Como objetivos específicos, tem-se:

1. definir uma representação cromossômica adequada para as imagens que constituirão os dados de teste para programas de PI;
2. definir critérios para a aferição da qualidade dos dados de testes e,
3. definir uma função de avaliação (função *fitness*) que determine a qualidade dos dados de teste, de acordo com os critérios de qualidade definidos.

A hipótese que será avaliada por este trabalho é: “um conjunto de imagens geradas por meio de algoritmo genético é capaz de revelar mais defeitos quando comparado ao teste tradicional realizado em programas de processamento de imagens”.

1.3 *Procedimentos metodológicos*

Os métodos de pesquisa utilizados são apresentados resumidamente nesta seção. O detalhamento dos procedimentos está disponível no Capítulo 4.

Esta pesquisa experimental foi desenvolvida em três etapas principais: revisão da literatura, desenvolvimento e validação.

Inicialmente, uma análise exploratória da literatura foi conduzida no tópico de geração de dados de teste para programas de PI. Os resultados obtidos nessa análise motivaram a condução de um mapeamento sistemático no tópico de geração de dados de teste por meio de algoritmos genéticos, que identificou o estado da arte nesta área de pesquisa (Capítulo 3).

A partir da análise dos resultados da etapa de revisão da literatura, foi definida uma nova abordagem de geração de dados de teste para programas de PI usando algoritmos genéticos. Paralelamente à etapa de desenvolvimento, as métricas de avaliação e os experimentos foram definidos na etapa de validação. Finalmente, a análise dos dados coletados por meio dos experimentos confirmou parcialmente a hipótese enunciada na seção anterior.

1.4 *Organização do documento*

Além desta introdução, esta dissertação está dividida nos seguintes capítulos:

- Capítulo 2: esse capítulo apresenta a fundamentação teórica necessária para compreender o presente trabalho, a saber: teste de *software*, processamento de imagens, geração de dados de teste e algoritmos genéticos.
- Capítulo 3: esse capítulo apresenta o mapeamento sistemático da literatura realizado no tópico de geração de dados de teste por meio de algoritmos genéticos.
- Capítulo 4: nesse capítulo são apresentados os métodos de pesquisa empregados neste trabalho.
- Capítulo 5: a abordagem proposta é apresentada e detalhada nesse capítulo.
- Capítulo 6: os resultados obtidos por meio dos experimentos conduzidos são apresentados e discutidos nesse capítulo.
- Capítulo 7: esse capítulo encerra o texto, sintetiza a motivação e a proposta do trabalho e apresenta as considerações finais.

2 Fundamentação teórica

Esta seção apresenta os principais conceitos envolvidos no presente trabalho: teste de *software*, processamento de imagens, geração de dados de teste e algoritmos genéticos.

2.1 Teste de *software*

O desenvolvimento de sistemas de informação pode se tornar complexo de acordo com os requisitos que devem ser atendidos. Além disso, as características e as habilidades das pessoas envolvidas neste processo exercem grande influência nos resultados. Por consequência, a presença de problemas não é rara (DELAMARO; MALDONADO; JINO, 2007).

O teste de *software* pode ser entendido como a comparação entre o resultado observado em uma execução controlada e o resultado esperado. Aos instrumentos que possibilitam decidir se o resultado observado está correto, dá-se o nome de *oráculos* (SOMMERVILLE, 2007). Os dados de entrada e as saídas esperadas constituem um caso de teste.

O principal objetivo do teste de *software* é revelar os erros de um programa (MYERS, 1979). Os erros revelados por um teste bem sucedido podem ser classificados como defeitos, enganos, erros ou falhas. *Defeitos* são passos, processos ou definições errados, causados por *enganos* cometidos pelas pessoas. Os defeitos podem fazer com que programas assumam um estado inconsistente durante a execução, ou seja, causam *erros*. Resultados incorretos gerados por conta de erros são chamados de *falhas* (DELAMARO; MALDONADO; JINO, 2007).

2.2 Técnicas de teste de *software*

Os testes idealmente deveriam considerar todo o domínio de entrada de um *software* (conjunto de valores de entrada possíveis) (SOMMERVILLE, 2007). Todavia, os domínios de entrada podem ter alta cardinalidade e inviabilizarem testes exaustivos. Para evitar este problema, são considerados subdomínios capazes de representar todos os elementos. Existem

três tipos de técnicas de testes consideradas na definição dos subdomínios: funcionais, estruturais e baseadas em defeitos (DELAMARO; MALDONADO; JINO, 2007).

O teste funcional, também conhecido como caixa preta, verifica se um sistema atende aos requisitos para os quais foi construído. Os subdomínios de entrada são definidos a partir dos requisitos, independentemente da estrutura interna do sistema. São exemplos de critérios de teste funcional: particionamento em classes de equivalência, análise do valor limite, teste funcional sistemático, grafo causa-efeito e *error-guessing*. Estes critérios são fortemente dependentes de uma boa especificação de requisitos e, por serem independentes da implementação, podem ser utilizados em conjunto com qualquer paradigma de programação (FABBRI; VICENZI; MALDONADO, 2007).

O teste estrutural, também conhecido como caixa branca, é baseado na implementação do programa. Os subdomínios de entrada são definidos de modo a garantir que determinadas condições, funções, definições ou usos de variáveis sejam exercitados. Os critérios estruturais podem ser divididos em três grupos: baseado em fluxo de controle, baseado em fluxo de dados e baseado na complexidade. Esses critérios apresentam o problema da correção coincidente: um mesmo caminho de um programa pode apresentar resultados corretos ou incorretos dependendo da entrada utilizada (BARBOSA et al., 2007).

A maior parte dos critérios de teste estrutural utiliza uma representação dos programas chamada grafo de fluxo de controle (GFC). Um GFC é um grafo orientado em que os nós equivalem a blocos de instruções e as arestas equivalem às possíveis sequências de blocos. Todas as instruções de um mesmo bloco devem ser necessariamente executadas em sequência. A partir do caminho percorrido em um GFC, é possível determinar os componentes do programa que são exercitados no teste (BARBOSA et al., 2007). A Figura 1 reproduz o GFC do programa *Identifier*.

De acordo com Barbosa et al. (2007), os critérios de teste baseados em fluxo de controle determinam os componentes a serem testados a partir das características de controle da execução do programa (comandos ou desvios). São exemplos de critérios baseados em fluxo de controle: todos-caminhos, todos-nós e todos-arcos.

Figura 1 – Grafo de Fluxo de Controle (GFC) do programa *Identifier*

Fonte: BARBOSA et al., 2007

Todos-Caminhos Deverão ser executados todos os caminhos possíveis do programa.

Todos-Nós Cada nó do GFC deve ser visitado durante a execução do programa (todos os comandos devem ser executados pelo menos uma vez).

Todos-Arcos Cada aresta do GFC deve ser exercitada durante a execução do programa (todas as possibilidades de desvio de fluxo do programa devem ser executadas).

Os critérios baseados em fluxo de dados determinam os requisitos de teste a partir da atribuição de valores às variáveis e dos usos desses valores (BARBOSA et al., 2007). Os critérios baseados em fluxo de dados utilizam uma extensão do GFC, o grafo Def-Use. Este grafo, além possuir informações referentes ao fluxo de controle, associa pontos do programa em que variáveis são definidas e em que são utilizadas (RAPPS; WEYUKER, 1982). Os principais critérios baseados em fluxo de dados são todas-definições e todos-usos.

Todas-Definições Cada uma das definições de variáveis deve ser exercitada.

Todos-Usos Cada associação entre pontos de definição e de uso de variáveis deve ser exercitada, sem que o valor da variável seja redefinido ao longo do caminho de execução do programa.

Segundo Pressman (2010), os critérios de teste baseados em complexidade determinam os requisitos de teste a partir de informações a respeito da complexidade do programa. O principal critério deste tipo é o critério de McCabe. Este critério utiliza a complexidade ciclomática do programa, isto é, o número de caminhos que introduzem uma nova aresta ao caminho de execução que ainda não havia sido coberta.

Vicenzi et al. (2007) definem o teste baseado em defeitos como uma técnica que utiliza os próprios defeitos de um programa para determinar os subdomínios de entrada. Podem-se gerar programas mutantes ou manualmente semear defeitos no programa original. As saídas do programa original são comparadas às saídas produzidas pelos programas mutantes ou com defeitos semeados a fim de se detectarem diferenças entre seus comportamentos. Neste contexto, os dados de teste são os dados capazes de evidenciar comportamentos diferentes entre as versões modificadas do programa e a versão original.

2.2.1 Teste de mutação

O teste de mutação (DEMILLO; LIPTON; SAYWARD, 1978) é uma técnica capaz de gerar dados de teste efetivos (e de aferir a qualidade de dados existentes) a partir de versões modificadas do programa sob teste. São criadas diferentes versões do programa original, cada uma com um único defeito introduzido. Desse modo, é possível determinar a qualidade do dado de teste a partir da quantidade e dos tipos de defeitos que este é capaz de revelar.

As versões modificadas do programa sob teste são chamadas de mutantes e são criadas por operadores de mutação. Cada operador de mutação corresponde a um tipo de defeito: inclusão, exclusão e troca de operadores (matemáticos, relacionais ou de atribuição por exemplo), comandos, variáveis, linhas de código, etc. O objetivo dos operadores de mutação é criar mutantes que simulem os defeitos mais comuns inseridos no desenvolvimento de *software* (HANH; BINH; TUNG, 2014).

Quando a saída produzida por um programa mutante é diferente da saída produzida pelo programa original, este mutante é considerado “morto”. Caso as saídas produzidas por ambos os programas forem iguais, então o mutante é considerado vivo (HANH; BINH; TUNG, 2016).

O objetivo do teste de mutação é prover um conjunto de teste que “mate” todos os mutantes gerados. Mutantes que permanecem vivos fazem com que o testador projete novos casos de teste com o intuito de “matá-los”. Existem também mutantes equivalentes, que são aqueles funcionalmente iguais ao programa original, ou seja, que sempre produzem, para qualquer dado de entrada, o mesmo resultado do programa original (DEMILLO; LIPTON; SAYWARD, 1978). Como consequência, mutantes equivalentes nunca morrem.

De acordo com Budd e Angluin (1982), a geração e a identificação de dados de teste adequados (capazes de garantir que um programa não possui defeitos) são problemas computacionais indecidíveis. No contexto do teste de mutação, isso significa que não é possível gerar e identificar dados de teste capazes de revelar os defeitos de todas as versões mutantes possíveis de um programa. Por esse motivo, o teste considera um conjunto finito de programas mutantes.

O teste de mutação é fundamentado em dois princípios empíricos definidos por DeMillo, Lipton e Sayward (1978):

- **Programadores são competentes:** os programadores criam programas muito próximos de estarem corretos, pois, devido à constante iteração do processo de desenvolvimento, conhecem os defeitos mais comuns e têm a possibilidade de analisar profundamente o código-fonte.
- **Efeito de acoplamento:** defeitos complexos são combinações de defeitos simples. Por consequência, os dados de teste capazes de identificar mutantes por meio dos defeitos simples também são sensíveis o suficiente para identificar defeitos mais complexos.

Um conjunto de dados de teste é avaliado por seu *mutation score* (MS), de acordo com a Equação 1. O MS é a razão entre o número de mutantes mortos (K) e o número total de mutantes (T), excluindo-se os mutantes equivalentes (E).

$$Mutation\ Score = \frac{K}{T - E} \quad (1)$$

A identificação de programas equivalentes é, em geral, um problema indecidível (BUDD; ANGLUIN, 1982). Assim, não se pode esperar que programas mutantes equivalentes sejam, sempre, identificados de forma automática. Adicionado a isso, o grande número de mutantes gerados, mesmo para pequenos programas, faz com que o custo do teste de

mutação seja alto. Para solucionar tais problemas, a estratégia mais utilizada é a redução do número de mutantes, por meio de algum tipo de seleção (OFFUTT; ROTHERMEL; ZAPF, 1993; USAOLA; MATEO, 2010).

2.3 Processamento de imagens

Segundo Conci, Azevedo e Leta (2008), imagens digitais são representadas por uma função bidimensional f em que as variáveis x e y são coordenadas espaciais e $f(x, y)$ representa a intensidade retornada por um sensor no ponto (x, y) . Cada ponto (x, y) é chamado de *picture element* ou simplesmente *pixel*. Os valores de intensidade são finitos e discretos. No domínio espacial, matrizes também são utilizadas como forma de representação de imagens.

A resolução espacial de uma imagem digital indica o seu menor detalhe discernível (GONZALEZ; WOODS, 2008). Essa medida é relevante apenas quando expressa em unidades de distância — geralmente *dots per inch* (dpi) — e pode ser interpretada como o maior número de pares de linhas discerníveis por unidade de distância (ou, intuitivamente, como o “tamanho” do *pixel*).

Analogamente, a resolução de contraste indica a menor mudança discernível de níveis de intensidade de uma imagem digital (GONZALEZ; WOODS, 2008). Entretanto, essa medida independe de unidades de distância. A quantidade de níveis de contraste é determinada pelo *hardware* do sensor. Geralmente, os níveis de contraste são potências de dois: 8 *bits* (256 níveis), 16 *bits* (65536 níveis) e assim por diante.

Os *pixels* vizinhos de um determinado *pixel* p definem vários tipos de vizinhança. As mais comuns são a vizinhança de 4 e a vizinhança de 8. A vizinhança de 4, $N_4(p)$, é constituída pelos *pixels* nas posições $(x - 1, y)$, $(x + 1, y)$, $(x, y + 1)$ e $(x, y - 1)$. A vizinhança de 8, $N_8(p)$, é constituída por $N_4(p)$ em conjunto com os quatro *pixels* diagonais nas posições $(x - 1, y + 1)$, $(x - 1, y - 1)$, $(x + 1, y + 1)$ e $(x + 1, y - 1)$ (GONZALEZ; WOODS, 2008).

Dois *pixels* p e q são adjacentes se estiverem conectados. Caso eles compartilhem uma borda, são ditos 4-adjacentes ou adjacentes por borda. Caso eles compartilhem um vértice, são ditos 8-adjacentes ou adjacentes por vértice (NUNES, 2006).

De acordo com Gonzalez e Woods (2008), é possível determinar se dois *pixels* são conectados a partir das relações de adjacência e de um critério de similaridade em uma determinada escala de cor. Considerando V como o conjunto de possíveis valores de intensidade $f(x, y)$ tem-se:

- conectividade-4: os *pixels* p e q , cujas intensidades são valores de V , são 4-conectados se q estiver no conjunto $N_4(p)$;
- conectividade-8: os *pixels* p e q , cujas intensidades são valores de V , são 8-conectados se q estiver no conjunto $N_8(p)$ e,
- conectividade-m: os *pixels* p e q , cujas intensidades são valores de V , são m-conectados se q estiver no conjunto $N_4(p)$ ou se estiver em $N_D(p)$ e não existirem *pixels* em $N_4(p) \cap N_4(q)$ com valores de intensidade em V .

A distância D entre os *pixels* p e q pode ser definida de diferentes formas como, por exemplo, a distância euclidiana (Equação 2), a *Manhattan* (Equação 3) e a *Chessboard* (Equação 4). O cálculo das distâncias é realizado a partir das coordenadas (x_p, y_p) e (x_q, y_q) dos *pixels* p e q . A distância euclidiana é baseada no teorema de Pitágoras, enquanto as distâncias *Manhattan* e *Chessboard* consideram os deslocamentos em linhas retas, em quatro ou oito direções, respectivamente (GONZALEZ; WOODS, 2008).

$$D_e(p, q) = \sqrt{(x_p - x_q)^2 + (y_p - y_q)^2} \quad (2)$$

$$D_4(p, q) = |x_p - x_q| + |y_p - y_q| \quad (3)$$

$$D_8(p, q) = \max(|x_p - x_q|, |y_p - y_q|) \quad (4)$$

De acordo com Gonzalez e Woods (2008), é possível classificar as técnicas de processamento de imagens em três níveis: baixo, médio e alto.

As técnicas de nível baixo englobam técnicas de pré-processamento de imagens. Estas técnicas removem dados indesejados e destacam os importantes. Podem ser utilizadas na recuperação de estruturas da imagem (descontinuidade de superfície, orientação, profundidade, velocidade, entre outras). São exemplos de técnicas de baixo nível as técnicas de suavização (como filtro de média, filtro de mediana e filtro gaussiano) e realce de imagens

(como equalização do histograma e quantização do histograma) (GONZALEZ; WOODS, 2008).

As técnicas de nível médio envolvem a segmentação de imagens (divisão da imagem em regiões ou objetos de interesse), classificação e descrição de objetos. A detecção de descontinuidades é um exemplo de técnica de nível médio (CONCI; AZEVEDO; LETA, 2008).

As técnicas de alto nível têm como objetivo conferir significado às imagens, aos objetos que foram reconhecidos. A análise de imagens e a visão computacional (simulação das funções cognitivas associadas à visão) são exemplos de técnicas de alto nível (GONZALEZ; WOODS, 2008).

2.4 Geração de dados de teste

Segundo Delamaro, Maldonado e Jino (2007), as técnicas de teste funcional, estrutural e baseada em defeitos particionam o domínio de entrada de acordo com critérios específicos. Determinados os subdomínios, é necessário selecionar dados de teste que atendam aos critérios de teste definidos ou automatizar este processo ao gerar os dados.

A técnica de geração aleatória é de implementação relativamente fácil e apresenta menores custos. Os dados de teste são gerados aleatoriamente até atenderem aos critérios de teste. Entretanto, caminhos não-executáveis podem não ser detectados e não há garantias de que os dados gerados são os que possuem a maior probabilidade de revelar defeitos (VERGILIO; MALDONADO; JINO, 2007).

Outra técnica utilizada é a geração simbólica. De acordo com Vergilio, Maldonado e Jino (2007), esta técnica produz expressões algébricas como representação dos caminhos de um programa. São obtidas, então, expressões simbólicas das variáveis de saída em função das variáveis de entrada e das condições de execução dos caminhos. As principais limitações são a representação de *loops* e de arranjos. Apesar de ser uma técnica com custos maiores do que a geração aleatória, a geração simbólica é mais eficaz.

A geração com execução dinâmica tem como base a execução real do programa e, portanto, não tem as limitações da geração simbólica. Em vez de expressões simbólicas, são utilizados dados reais. O fluxo de execução do programa é monitorado e determinam-se as variáveis de entrada que causam erros (VERGILIO; MALDONADO; JINO, 2007).

Os dados de teste podem ser gerados, também, a partir de técnicas de busca e otimização. Estas técnicas consideram a geração de dados de teste como uma busca em um grande espaço de soluções. Cada solução possui uma aptidão para resolver o problema, determinada por uma função de avaliação (função *fitness*) (ALLI et al., 2010). Dentre as técnicas de busca, as metaheurísticas têm sido as mais aplicadas ao teste de *software* nos últimos anos (DAVE; AGRAWAL, 2015).

As metaheurísticas são técnicas genéricas de busca e otimização capazes de encontrar soluções ótimas (ou bastante próximas à ótima) para problemas cujos espaços de solução são grandes e complexos (ALLI et al., 2010). A geração de dados de teste pode ser entendida como um problema de busca de soluções em que o espaço de soluções é o domínio de entrada do programa. As soluções desejadas são aquelas que melhor atendem a determinados critérios de teste e que são capazes de revelar defeitos no *software* (DAVE; AGRAWAL, 2015). São exemplos de metaheurísticas: busca Tabu, *simulated annealing* (resfriamento simulado), *particle swarm optimization* (otimização por enxame de partículas), colônia de formigas, algoritmo memético, sistemas imunes artificiais e algoritmo genético (BOUCHACHIA, 2007).

2.5 Algoritmos genéticos

De acordo com Mitchell (1999), algoritmos evolucionários inspiram-se na biologia para solucionar problemas de busca e otimização. Estes algoritmos tratam as soluções candidatas de um problema como indivíduos de uma população, que é evoluída por meio de processos que simulam a variação e a seleção genéticas naturais.

Os algoritmos genéticos (AG) constituem a classe de algoritmos evolucionários mais conhecida. Além desta subdivisão, há as estratégias evolutivas (EV), a programação evolutiva (PE), a programação genética (PG) e os sistemas classificadores (SC) (VON ZUBEN, 2011).

As estratégias evolutivas foram criadas por Rechenberg (1973) e aprimoradas por Schwefel (1975). Nesta classe de algoritmos, a população é composta por vetores de números em ponto flutuante, que incluem os parâmetros da própria estratégia evolutiva. A seleção é determinística e a evolução ocorre por meio de mutação (probabilística) e *crossover*.

A programação evolutiva foi proposta por Fogel, Owens e Walsh (1966). Não há *crossover* nesta abordagem e a população é submetida apenas à mutação (probabilística). A seleção, em vez de determinística como nas estratégias evolutivas, é probabilística. A PE foi desenhada para populações compostas por máquinas de estado.

A programação genética foi desenvolvida por Koza (1992). Nesta abordagem, a população é constituída por programas computacionais, representados por árvores (estruturas de dados). São utilizados *crossover*, mutação e seleção (probabilística).

De acordo com Goldberg (1989), sistemas classificadores são sistemas de aprendizado de máquina baseados em regras simples. Estas regras podem ser geradas por algoritmos genéticos a fim de melhorar o aprendizado do sistema.

Holland (1975) desenvolveu os algoritmos genéticos como uma forma de abstrair a evolução biológica e incorporá-la em sistemas computacionais. Diferentemente das estratégias evolutivas e da programação genética, nos algoritmos genéticos, a população é composta por muitos indivíduos e é submetida aos processos de *crossover*, mutação e inversão (MITCHELL, 1999). No algoritmo criado por Holland (1975), a mutação ocorre com uma probabilidade baixa, de modo que o *crossover* tenha a maior influência no processo evolutivo. A seleção dos indivíduos é probabilística e proporcional à aptidão de cada indivíduo, aferida pela função *fitness* (VON ZUBEN, 2011).

Os principais objetivos de Holland ao propor os algoritmos genéticos foram: abstrair e explicar detalhadamente a seleção e a variação genéticas observadas na natureza; e construir sistemas artificiais que possuam os principais mecanismos evolutivos observados na natureza (GOLDBERG, 1989).

Algoritmos genéticos são aplicados em diversas áreas como: programação automática (evolução de programas Lisp e evolução de autômatos celulares); análise e predição de dados (predição de estruturas de proteínas); redes neurais (evolução dos pesos de redes neurais e evolução da arquitetura das redes neurais) (MITCHELL, 1999); roteamento (robótica, problema do caixeiro viajante, roteamento de veículos); sequenciamento de tarefas; gerenciamento de processos e de memória; processamento de sinais (VON ZUBEN, 2011) e geração de dados de teste (ALLI et al., 2010).

O princípio de funcionamento dos algoritmos genéticos é a descoberta, priorização e recombinação de blocos construtivos. Os blocos construtivos podem ser entendidos como sequências de fragmentos de informações capazes de aumentar a aptidão dos indivíduos para resolver o problema proposto (MITCHELL, 1999).

Comparados aos métodos de busca e otimização tradicionais, os algoritmos genéticos (e metaheurísticas em geral) apresentam diferenças fundamentais (GOLDBERG, 1989):

- AGs manipulam uma codificação dos parâmetros do problema (isto é, os dados de teste), em vez dos parâmetros propriamente ditos;
- AGs buscam soluções em uma população de pontos, em vez de “caminharem” entre pontos isolados;
- AGs usam informações de custo ou recompensa (isto é, função *fitness*), em vez de derivadas de funções ou outras informações auxiliares;
- AGs têm regras de transição probabilísticas, em vez de determinísticas.

2.5.1 Arquitetura dos algoritmos genéticos

Os algoritmos genéticos simulam o processo de evolução biológica de Darwin no contexto de busca e otimização de problemas complexos. Cada solução candidata é considerada um indivíduo. Em cada geração de indivíduos, ocorrem a reprodução de indivíduos e a mutação de genes. Apenas os indivíduos mais aptos sobrevivem. Estes mecanismos garantem a variação do material genético e, por consequência, das soluções candidatas. Apesar de haver um fator de aleatoriedade (mutação), as informações históricas são consideradas ao longo da execução do algoritmo (devido à operação de *crossover*). Em outras palavras, o operador de *crossover* mantém traços das gerações passadas nas gerações futuras. São essas características dos algoritmos genéticos que possibilitam uma busca eficiente em um grande espaço de soluções (GOLDBERG, 1989).

Segundo Mitchell (1999), não existe uma definição precisa de algoritmos genéticos, entretanto é notável que todas as implementações possuem alguns componentes básicos em comum: população, seleção, reprodução e mutação.

População Em geral, os indivíduos são representados por vetores de números (inteiros ou reais, dependendo do domínio do problema), que são os cromossomos. Cada posição do vetor, ou gene, representa uma característica do indivíduo (solução candidata), um parâmetro do algoritmo.

Seleção Uma função de avaliação (*fitness*) é definida a fim de aferir a aptidão de uma solução a resolver o problema. Os indivíduos mais aptos são seleciona-

dos para reprodução. Os vetores que correspondem às melhores soluções candidatas (de acordo com a função *fitness*) são utilizados pelo operador de reprodução e dão origem a novos indivíduos.

Reprodução Os indivíduos mais aptos trocam material genético entre si (*crossover*) e dão origem a novos indivíduos. Em geral, este operador genético é implementado como uma divisão de vetores em n partes. Posteriormente, partes de diferentes vetores são unidas a fim de dar origem a novos vetores (indivíduos).

Mutação Alguns genes dos indivíduos podem sofrer alterações aleatórias, com o objetivo de promover a variação entre os indivíduos. Atribuem-se, usualmente, números aleatórios a uma posição do vetor escolhida aleatoriamente.

Conforme apresentado na Figura 2, a execução de um algoritmo genético pode ser detalhada nos seguintes passos (MITCHELL, 1999):

1. A população inicial é gerada aleatoriamente. São gerados n cromossomos de l bits.
2. A aptidão de cada cromossomo (solução candidata) é aferida pela função *fitness*.
3. Os passos seguintes deverão ser repetidos até que m novos indivíduos sejam gerados.
 - a) Selecionar um par de cromossomos da população para reprodução. Quanto maior a aptidão de um cromossomo, maior será a probabilidade de ele ser selecionado. A seleção pode escolher o mesmo cromossomo duas vezes.
 - b) A reprodução (*crossover*) ocorre com probabilidade p_c e troca os genes dos pais a partir de uma posição aleatória, dando origem a dois filhos. Caso não ocorra *crossover*, os filhos serão idênticos aos pais.
 - c) Cada gene de cada filho poderá sofrer uma mutação com probabilidade p_m , a fim de promover a diversidade genética na população.
4. Os indivíduos que sobreviverão e integrarão a próxima geração são escolhidos.
5. Se as condições de parada não tiverem sido alcançadas, voltar para o passo 2.

A sequência de passos apresentada representa um algoritmo genético básico. Podem existir variações nos diferentes componentes básicos ou em elementos como, por exemplo, a condição de parada. Os algoritmos genéticos podem utilizar diferentes representações cromossômicas, probabilidades e tipos de *crossover* e mutação, tamanhos de população,

Figura 2 – Fluxograma de um algoritmo genético

Fonte: adaptado de Xibo e Na (2011) e Fischer e Tönjes (2012)

entre outras possibilidades (MITCHELL, 1999). As seções seguintes apresentarão as particularidades de cada componente citado.

2.5.2 População de cromossomos

As soluções candidatas de um problema são codificadas, geralmente, em vetores de números. Os valores das posições dos vetores representam *genes* e os vetores representam *cromossomos* (Figura 3). Cada posição nos cromossomos é chamada de *locus*. Os genes podem ser entendidos como codificações de características ou traços do indivíduo; são instâncias de um dos possíveis *alelos*, representados por números (outros alfabetos podem ser utilizados) (MITCHELL, 1999).

Figura 3 – Cromossomo em um algoritmo genético padrão com codificação binária

Fonte: adaptado de VON ZUBEN, 2011

Existem diferentes tipos de codificação de soluções candidatas. As principais são a codificação binária, codificação multicaracteres, codificação com valores reais e a codificação em árvore (MITCHELL, 1999).

A codificação binária, proposta por Holland (1975), utiliza tamanho fixo de cromossomos e ordem fixa de genes. Apesar de ser a codificação mais estudada, ela não é ideal para a maioria dos problemas como, por exemplo, a evolução de estruturas (ou pesos) de uma rede neural artificial. Neste caso, uma codificação que utilize vários caracteres e suporte valores reais é mais indicada.

A codificação em árvore permite que as soluções candidatas sejam representadas de maneira flexível, dado que o *crossover* e a mutação podem formar novas árvores de qualquer tamanho (MITCHELL, 1999). Koza (1992) utilizou este tipo de codificação para representar programas na técnica de programação evolutiva.

Há, ainda, outro fator importante relativo à população: os indivíduos podem ser haploides ou diploides. Caso as soluções candidatas sejam representadas por indivíduos haploides, cada um deles terá um único cromossomo. Caso as soluções candidatas sejam representadas por indivíduos diploides, cada um deles terá um par de cromossomos (GOLDBERG, 1989).

2.5.3 Operadores genéticos

Os algoritmos genéticos utilizam diversos operadores no processo de evolução das populações. Os mais utilizados são seleção, *crossover* (recombinação) e mutação.

Seleção de indivíduos

O operador de seleção determina os indivíduos que serão utilizados pelo operador de reprodução, de acordo com sua aptidão (*fitness*). A aptidão de um indivíduo é dada pela função *fitness*, que diferencia as soluções candidatas em termos da capacidade de resolver o problema (VON ZUBEN, 2011). Os indivíduos mais aptos terão mais chances de serem escolhidos para reprodução e, desta forma, dar origem a novos indivíduos mais aptos (GOLDBERG, 1989).

Caso a seleção seja muito rigorosa, soluções subótimas com *fitness* alto podem impedir que soluções ótimas com *fitness* menor sobrevivam. Em contrapartida, um operador de seleção pouco rigoroso implicará em um processo de evolução mais lento (MITCHELL, 1999).

O método de seleção proposto por Holland (1975), *Roulette Wheel*, considera o valor esperado de cada indivíduo da população. O valor esperado de um indivíduo é a razão entre o seu valor de *fitness* e a soma de todos os valores de *fitness* dos demais indivíduos da população (Tabela 1). Cada indivíduo é associado a uma faixa da roleta, proporcionalmente ao seu valor esperado (Figura 4). A roleta é girada n vezes (para uma população de n indivíduos) e os indivíduos que estiverem sob o ponteiro da roleta serão selecionados para o processo de reprodução que originará a próxima população (MITCHELL, 1999).

Tabela 1 – Cálculo de valor esperado para quatro indivíduos

Indivíduo	Cromossomo	<i>Fitness</i>	Valor esperado	Graus
a	0001100101010	6,0	0,5	180°
b	0101001010101	3,0	0,25	90°
c	1011110100101	1,5	0,125	45°
d	1010010101001	1,5	0,125	45°

Fonte: adaptado de VON ZUBEN, 2011

Este método permite que o mesmo indivíduo seja selecionado mais de uma vez e, apesar de ser pouco provável, pode selecionar apenas os piores indivíduos para reprodução. Para minimizar o impacto destes problemas, é possível adotar a estratégia de elitismo (manter intactos os indivíduos mais aptos de cada geração) (MITCHELL, 1999).

Figura 4 – Roleta criada a partir da população descrita na Tabela 1

Fonte: adaptado de VON ZUBEN, 2011

Outros métodos de seleção podem ser utilizados. Caso seja necessário incluir um controle do rigor do processo de seleção, o método de Boltzmann pode ser aplicado. O método de seleção por classificação distribui os indivíduos na roleta priorizando os mais aptos. A seleção por torneio toma aleatoriamente dois indivíduos, gera um número aleatório r e o compara a um parâmetro k : se $r < k$, o indivíduo mais apto é selecionado; caso contrário, o indivíduo menos apto é selecionado. Utilizando a seleção *steady state*, novas gerações substituirão apenas alguns dos indivíduos menos aptos, em vez de substituir toda a população (MITCHELL, 1999).

Crossover

O operador de *crossover*, ou recombinação, promove a troca de material genético entre dois indivíduos, gerando dois descendentes que farão parte da próxima população (GOLDBERG, 1989).

De acordo com a probabilidade de *crossover* (p_c), dois indivíduos podem ter seus genes trocados a partir de um locus definido aleatoriamente. Este é o *crossover* de um único ponto (Figura 5).

Figura 5 – *Crossover* de um único ponto entre dois indivíduos

Fonte: adaptado de VON ZUBEN, 2011

A partir do ponto de *crossover*, os dois indivíduos são divididos em duas partes. O primeiro descendente terá os genes da parte esquerda do primeiro pai e os genes da parte direita do segundo pai. Analogamente, o segundo descendente terá os genes da parte esquerda do segundo pai e os genes da parte direita do primeiro pai (LINDEN, 2008).

A recombinação também pode ser feita a partir de mais de um ponto. O *crossover* de vários pontos seleciona aleatoriamente várias posições e efetua a recombinação dos genes entre os pais. O *crossover* uniforme utiliza uma probabilidade independente para cada locus, que determina se os genes serão trocados ou não (MITCHELL, 1999).

Assim como os outros operadores já mencionados, o *crossover* é probabilístico (p_c) e é possível que não ocorra recombinação entre indivíduos. Nestes casos, são gerados descendentes idênticos aos pais (MITCHELL, 1999).

Mutação

A mutação, usualmente, é o operador secundário nos algoritmos genéticos e ocorre com probabilidades menores. O *crossover* é o maior responsável por gerar novos indivíduos que herdem as melhores características das gerações anteriores. A mutação tem o papel de garantir a diversidade genética entre os indivíduos e é um dos mecanismos que reduzem as chances de que algoritmos genéticos fiquem “presos” em máximos locais (MITCHELL, 1999).

Este operador realiza a troca aleatória de um dos alelos de um gene. Cada gene do cromossomo tem uma probabilidade p_m de sofrer mutação (VON ZUBEN, 2011). Em cromossomos binários, a mutação troca zeros por uns (Figura 6).

Figura 6 – Mutação de um gene de um cromossomo binário

Fonte: Davi Silva Rodrigues, 2017

As probabilidades p_c e p_m de *crossover* e mutação (respectivamente) podem ser modificadas ao longo da execução do algoritmo genético com o objetivo de balancear o efeito dos operadores na evolução da população (ALETI; GRUNSKE, 2015; ALETI; MOSER, 2016). Deste modo, pode-se aumentar a probabilidade de *crossover* no início da execução a fim de explorar o espaço de soluções. Analogamente, ao final da execução, a probabilidade de mutação pode ser aumentada para que máximos ou mínimos locais sejam evitados (MITCHELL, 1999).

2.6 Considerações finais

Neste capítulo foram apresentadas as bases teóricas desta pesquisa: os principais conceitos referentes ao teste de *software*, à geração de dados de teste e aos algoritmos genéticos. Tais conceitos serão empregados no capítulo seguinte, que apresenta um mapeamento sistemático da literatura no tópico de geração de dados de teste por meio de AGs.

3 Mapeamento sistemático

Mapeamento sistemático (MS) é uma revisão da literatura realizada de acordo com passos bem definidos e devidamente documentados, a fim de tornar possível a auditoria e a reprodução dos métodos. Diferentemente das revisões sistemáticas, o MS tem como objetivo fornecer uma visão abrangente de uma área de pesquisa. Um MS deve classificar, categorizar ou agrupar as evidências apresentadas por estudos primários, de modo a responder questões de pesquisa (KITCHENHAM, 2007).

Esta seção apresenta um MS da literatura no tópico de geração de dados de teste por meio de algoritmos genéticos.

3.1 Geração de dados de teste no domínio de imagens

Uma análise exploratória da literatura foi realizada com a ferramenta Google Scholar (2017), que compila artigos científicos de diversas bases bibliográficas, assim como em algumas das fontes mais relevantes na área da Ciência da Computação: IEEE Xplore Digital Library (2017), ACM Digital Library (2017), Scopus (2017) e Science Direct (2017).

Apenas o trabalho de Jameel et al. (2014) foi encontrado no âmbito de geração de dados de teste para sistemas de PI. Neste estudo, os pesquisadores propuseram uma técnica de geração de dados de teste que utiliza segmentação de imagens e execução simbólica, na qual são atribuídos símbolos às variáveis e às saídas esperadas, com base nos caminhos de execução do programa. Utilizando Matlab, a técnica proposta identifica os caminhos possíveis de execução e é capaz de gerar dados de teste para todos eles. A complexidade da entrada é tratada por meio da utilização de janelas, que são partes de uma imagem. É feito o processamento por janelas até que todos os *pixels* sejam processados.

O trabalho de Narciso (2013) é específico para o domínio de imagens, porém apresenta uma técnica de seleção de casos de teste. Outros trabalhos abordam a geração de dados de teste, porém em outros domínios e utilizando diversas técnicas. Wang et al. (2009) utilizaram geração aleatória de dados a fim de executar testes de programas escritos em linguagem C. Li et al. (2010) propuseram a geração de dados de teste para *web services* que utilizam BPEL (*Business Process Execution Language*) com base nos caminhos de execução da aplicação.

Os resultados da análise exploratória indicaram um *gap* na pesquisa em geração de dados de teste para programas de PI, além de uma tendência dos estudos em utilizar AGs para gerar dados de teste ou apresentar revisões sistemáticas sobre o tema (ALLI et al., 2010; AFZAL; TORKAR; FELDT, 2009; DAVE; AGRAWAL, 2015; GUPTA; ROHIL, 2013). Por esses motivos, decidiu-se por um aprofundamento na literatura, por meio da condução de um mapeamento sistemático, conforme apresentado a seguir.

3.2 Protocolo e condução do mapeamento sistemático

Este MS foi realizado a fim de obter um panorama do estado da arte da pesquisa em geração de dados de teste por meio de algoritmos genéticos. O protocolo foi elaborado a partir dos resultados encontrados a partir da análise exploratória apresentada na Seção 3.1. O Apêndice A apresenta o protocolo completo do mapeamento.

A busca foi realizada, em dezembro de 2016, nas bases de dados IEEE Xplore Digital Library (2017), ACM Digital Library (2017), Scopus (2017) e Science Direct (2017). A mesma *string* de busca foi utilizada em todas as bases: (“genetic algorithm” OR “evolutionary test”) AND (“test case generation” OR “test data generation”).

A seguinte questão de pesquisa guiou o processo do mapeamento sistemático:

- Quais são as principais técnicas de geração de dados de teste de *software* que utilizam algoritmos genéticos?

Foram definidos critérios de inclusão **I** e de exclusão **E** para a seleção dos estudos primários. A seleção incluiu os estudos que atenderam a pelo menos um dos critérios de inclusão e não contemplaram nenhum dos critérios de exclusão. Os critérios de inclusão e de exclusão estão listados abaixo.

I1 Estudos que apresentem técnicas de geração de dados de testes por meio de algoritmos genéticos.

I2 Estudos que apresentem as vantagens e as desvantagens dos algoritmos genéticos para a geração de dados de testes.

E1 Estudos que não foram escritos em inglês.

E2 Estudos que não passaram por revisão por pares (capítulos de livros, por exemplo).

E3 Estudos que apresentem revisões de literatura.

E4 Estudos indisponíveis na *web*.

A Figura 7 indica a quantidade de estudos primários aceitos em cada uma das bases de dados a partir da *string* de busca. Ao todo, foram retornados 767 resultados, dos quais 119 estavam duplicados entre as bases e 580 foram excluídos. Os estudos aceitos estão distribuídos entre as bases de dados da seguinte forma:

- 40 estudos (59%) na base *Scopus*;
- 17 estudos (25%) na base *IEEE Xplore Digital Library*;
- 6 estudos (9%) na base *Science Direct*, e
- 5 estudos (7%) na base *ACM Digital Library*.

Figura 7 – Estudos primários aceitos por base de dados

Fonte: Davi Silva Rodrigues, 2017

A Figura 8 sintetiza a fase de condução do MS, que selecionou 68 estudos primários. A busca realizada nas bases de dados mencionadas anteriormente retornou 767 resultados. Após a exclusão de 119 resultados duplicados, os demais estudos foram incluídos ou excluídos de acordo com os critérios definidos:

- estudos secundários, capítulos de livros, estudos que não foram escritos em inglês, estudos indisponíveis e estudos não relacionados à questão de pesquisa foram excluídos e,

- estudos que não se enquadraram em nenhum dos critérios de exclusão e apresentaram técnicas de geração de dados de teste por meio de AGs ou mostraram suas vantagens e desvantagens foram incluídos.

Figura 8 – Síntese da condução do mapeamento sistemático

Fonte: Davi Silva Rodrigues, 2017

Uma ficha de extração de dados foi elaborada e aplicada a cada um dos estudos primários incluídos. Foram extraídos o tipo de técnica de teste de *software* utilizada, as métricas de avaliação utilizadas, a forma de representação dos programas sob teste, o país em que a pesquisa foi realizada e uma categoria relativa às adaptações propostas nos algoritmos genéticos utilizados.

3.3 Análise global dos resultados obtidos

As Tabelas 2, 3 e 4 apresentam os 68 estudos primários incluídos, agrupados por técnica de teste de *software*, além de parte dos dados da ficha de extração: a categoria à qual o estudo pertence, a técnica de teste para qual os dados são gerados e as métricas de avaliação utilizadas. Os estudos geraram dados para teste funcional (Tabela 2), de mutação (Tabela 3) e estrutural (Tabela 4).

Toda implementação de AG encontrada nos estudos incluídos contém os componentes básicos apresentados na Seção 2.5 (isto é, população, seleção e operadores de reprodução e de mutação). Entretanto, as adaptações dos componentes do AG e as adaptações específicas do problema em questão estão intimamente relacionadas e, portanto, é difícil desacoplá-las. Dessa forma, os estudos foram categorizados de acordo com a adaptação que foi destacada pelos seus autores, dado que os componentes mencionados anteriormente estão relacionados tanto ao AG quanto ao problema sob análise.

Os estudos incluídos foram divididos em cinco categorias, de acordo com a adaptação mais significativa apresentada por seus autores: função *fitness* (Seção 3.5), população (Seção 3.6), representação dos indivíduos (Seção 3.7) e arquitetura (Seção 3.8). Finalmente, os algoritmos híbridos são apresentados na Seção 3.9.

Tabela 2 – Estudos primários incluídos na fase de extração do MS — teste funcional

Categoria	Nº	Estudo	Avaliação
Função <i>fitness</i>	1	(BETTS; PETTY, 2016)	Desvio lateral máximo
	2	(DERDERIAN et al., 2011)	Cobertura de estados, Número de gerações
	3	(HUANG; COHEN; MEMON, 2010)	Cobertura de dados de teste viáveis
	4	(NÚÑEZ et al., 2013)	Cobertura de estados
	5	(SHARMA; PASALA; KOMMINENI, 2012)	Número de gerações, Tempo de execução
	6	(VUDATHA et al., 2011)	Número de gerações
Algoritmo híbrido	7	(ALESIO et al., 2015)	Tempo de execução
	8	(LIN et al., 2012)	Cobertura de combinações
População	9	(FISCHER; TÖNJES, 2012)	Proporção entre dados de teste válidos e inválidos
	10	(ZHOU; ZHAO; YOU, 2014)	Número de gerações
Representação dos indivíduos	11	(ALSMADI, 2010)	Número de erros revelados, Tempo de execução
	12	(ARORA; SINHA, 2014)	Número de erros revelados
	13	(LEFTICARU; IPATE, 2007)	Cobertura de caminhos
	14	(PENG; LU, 2011)	Cobertura de estados

Fonte: Davi Silva Rodrigues, 2017

Tabela 3 – Estudos primários incluídos na fase de extração do MS — teste de mutação

Categoria	Nº	Estudo	Avaliação
Função <i>fitness</i>	15	(HANH; BINH; TUNG, 2016)	<i>Mutation score</i>
	16	(KHAN; AMJAD, 2015)	<i>Mutation score</i>
	17	(RANI; SURI, 2015)	Número de mutantes
	18	(YAO; GONG; ZHANG, 2015)	Cobertura de instruções, Tempo de execução, <i>Mutation score</i>
Algoritmo híbrido	19	(HANH; BINH; TUNG, 2014)	<i>Mutation score</i> , Tempo de execução

Fonte: Davi Silva Rodrigues, 2017

Tabela 4 – Estudos primários incluídos na fase de extração do MS — teste estrutural

Categoria	Nº	Estudo	Avaliação
Arquitetura	20	(ALETI; GRUNSKE, 2015)	Cobertura de ramificações
	21	(EL-SERAFY et al., 2015)	Número de gerações, Tempo de execução
	22	(MANIKUMAR; KUMAR; MARUTHAMUTHU, 2016)	Cobertura de ramificações, Número de gerações
	23	(PACHAURI; SRIVASATAVA, 2015)	Número de gerações, Cobertura de caminhos
Função <i>fitness</i>	24	(AHMED; ALI, 2015)	Cobertura de caminhos, Número de gerações
	25	(AHMED; HERMADI, 2008)	Número de gerações
	26	(ALEB; KECHID, 2013)	Número de gerações, Número de indivíduos
	27	(CAO; HU; LI, 2009)	Número de gerações, Tempo de execução
	28	(CHEN; ZHONG, 2009)	Número de indivíduos, Tempo de execução
	29	(GHIDUK; HARROLD; GIRGIS, 2007)	Cobertura de requisitos, Número de indivíduos, Tempo de execução, Número de gerações
	30	(GHIDUK; GIRGIS, 2010)	Cobertura de instruções, Número de gerações
	31	(HERMADI; AHMED, 2003)	Número de gerações
	32	(JIN; JIANG; ZHANG, 2011)	Número de gerações
	33	(KHAN; AMJAD; SRIVASATAVA, 2016)	Cobertura de caminhos

Continua...

Tabela 4 – Estudos primários incluídos na fase de extração do MS — teste estrutural

Continuação			
Categoria	Nº	Estudo	Avaliação
	34	(LAKHOTIA; HARMAN; MC-MINN, 2007)	Cobertura de ramificações
	35	(MANN et al., 2016)	Cobertura de ramificações
	36	(PARTHIBAN; SUMALATHA, 2013)	Cobertura de ramificações
	37	(PINTO; VERGILIO, 2010)	Número de gerações, Tempo de execução
	38	(SHIMIN; ZHANGANG, 2011)	Tempo de execução
	39	(SHUAI et al., 2015)	Tempo de execução
	40	(SOLTANI; PANICHELLA; DEURSEN, 2016)	Número de erros reproduzidos
	41	(SUN; JIANG, 2010)	Cobertura de caminhos
	42	(VARSHNEY; MEHROTRA, 2016)	Número de gerações, Tempo de execução
	43	(XIBO; NA, 2011)	Cobertura de caminhos
	44	(YAO; GONG; WANG, 2015)	Tempo de execução
	45	(ZHANG; GONG, 2010)	Tempo de execução, Número de gerações
Algoritmo híbrido	46	(BOUCHACHIA, 2007)	Número de gerações, Cobertura de condições
	47	(FAN et al., 2015)	Número de indivíduos, Cobertura de decisões
	48	(GARG; GARG, 2015)	Número de indivíduos
	49	(JI; SUN, 2012)	Número de gerações, Cobertura de caminhos
	50	(KIFETEW et al., 2013)	Cobertura de ramificações
	51	(MAYAN; RAVI, 2014)	Tempo de execução, Cobertura de código
	52	(MILLER; REFORMAT; ZHANG, 2006)	Cobertura de ramificações, Cobertura de instruções
	53	(TAN; LONGXIN; XIUMEI, 2009)	Cobertura de caminhos
População	54	(ALSHRAIDEH; MAHAFZAH; AL-SHARAEH, 2011)	Tempo de execução, Número de gerações
	55	(CHEN; ZHONG, 2008)	Cobertura de caminhos, Tempo de execução
	56	(DEEPAK; SAMUEL, 2009)	Número de gerações, Tempo de execução, Cobertura de caminhos
Continua...			

Tabela 4 – Estudos primários incluídos na fase de extração do MS — teste estrutural

			Continuação
Categoria	Nº	Estudo	Avaliação
	57	(GONG; ZHANG; YAO, 2011)	Número de execuções da função <i>fitness</i>
	58	(TIAN; GONG, 2016)	Número de execuções da função <i>fitness</i> , Tempo de execução
	59	(YANG et al., 2016)	Cobertura de ramificações, Número de gerações, Tempo de execução
	60	(YAO; GONG, 2014)	Cobertura de caminhos, Tempo de execução
	61	(ZHANG et al., 2010)	Tempo de execução
	62	(ZHANG; WU; BAO, 2015)	Cobertura de caminhos, Tempo de execução, Número de indivíduos
Representação dos indivíduos	63	(KHANDELWAL; TOMAR, 2015)	Cobertura de caminhos
	64	(KOLEEJAN; XUE; ZHANG, 2015)	Tempo de execução, Cobertura de código
	65	(LIJUAN; YUE; HONGFENG, 2012)	Cobertura de ramificações viáveis
	66	(LIU; WANG; WANG, 2013)	Cobertura de caminhos
	67	(SILVA; VAN SOMEREN, 2010)	Número de erros revelados, Tempo de execução
	68	(TONELLA, 2004)	Cobertura de ramificações

Fonte: Davi Silva Rodrigues, 2017

A Figura 9 apresenta a distribuição dos estudos apresentados nas Tabelas 2, 3 e 4 entre as categorias descritas anteriormente.

Figura 9 – Distribuição dos estudos aceitos entre as categorias apresentadas

Fonte: Davi Silva Rodrigues, 2017

Cada uma das categorias foi dividida em até três subgrupos, de acordo com o tipo de técnica de teste para a qual os dados de teste foram gerados: teste funcional, teste de mutação ou teste estrutural.

A partir dos dados extraídos, foi possível observar que 37% dos estudos incluídos foram desenvolvidos na China e 22% foram desenvolvidos na Índia. Os 68 estudos incluídos foram publicados entre os anos de 2003 e 2016, conforme mostra a Figura 10. O ano de 2015 destaca-se como o ano com a maior quantidade de artigos publicados.

Figura 10 – Distribuição dos estudos aceitos por ano e por categoria

Fonte: Davi Silva Rodrigues, 2017

3.4 Uso de algoritmos genéticos para geração de dados de teste

A maior parte dos estudos aceitos implementa técnicas de geração de dados para teste estrutural. Em termos percentuais, 72% dos estudos automatizaram a geração de dados para teste estrutural, 21% para teste funcional e 7% para teste de mutação.

A maioria das técnicas de geração de dados para teste estrutural extraem da implementação um grafo de fluxo de controle (GFC) (ALETI; GRUNSKE, 2015; CHEN; ZHONG, 2008; LAKHOTIA; HARMAN; MCMINN, 2007; KHANDELWAL; TOMAR, 2015; SUN; JIANG, 2010), um grafo de fluxo de dados (DFG — *Data Flow Graph*, uma extensão do GFC que inclui o uso e a definição de variáveis) (KHAN; AMJAD; SRIVAS-

TAVA, 2016; PINTO; VERGILIO, 2010; VARSHNEY; MEHROTRA, 2016; ZHANG; WU; BAO, 2015) ou outros grafos (JIN; JIANG; ZHANG, 2011; MAYAN; RAVI, 2014; MILLER; REFORMAT; ZHANG, 2006) e os utilizam como forma de representação do programa. Muitos estudos utilizaram o código-fonte diretamente como representação do programa sob teste (BOUCHACHIA, 2007; EL-SERAFY et al., 2015; GHIDUK; HARROLD; GIRGIS, 2007; HERMADI; AHMED, 2003; LIJUAN; YUE; HONGFENG, 2012; MANN et al., 2016; PARTHIBAN; SUMALATHA, 2013; YANG et al., 2016). Apenas um estudo propôs o uso de tabelas de variáveis e de controles em vez de grafos ou código-fonte (ALEB; KECHID, 2013). Invariavelmente, a população inicial é gerada aleatoriamente (AHMED; ALI, 2015; DEEPAK; SAMUEL, 2009; KIFETEW et al., 2013). Após o processo de evolução, os indivíduos mais aptos são utilizados como dados de teste nos programas.

As técnicas de geração de dados para teste funcional utilizam como representação do programa máquinas de estado (ARORA; SINHA, 2014; DERDERIAN et al., 2011; NÚÑEZ et al., 2013; LEFTICARU; IPATE, 2007), grafos baseados em interface de usuário (ALSMADI, 2010) e em requisições *web* (PENG; LU, 2011). A população inicial também é gerada aleatoriamente (ALSMADI, 2010; HUANG; COHEN; MEMON, 2010; LEFTICARU; IPATE, 2007) ou a partir de dados previamente existentes (PENG; LU, 2011).

As técnicas que geram dados para teste de mutação utilizam o código-fonte dos programas diretamente em vez de usar grafos, máquinas de estado ou outras formas de representação. O objetivo dessas técnicas é gerar dados de teste capazes de matar versões mutantes do programa sob teste (HANH; BINH; TUNG, 2014; HANH; BINH; TUNG, 2016; KHAN; AMJAD, 2015; RANI; SURI, 2015).

Na geração de dados de teste (funcional, de mutação ou estrutural), os algoritmos genéticos (a partir da representação dos programas) geram indivíduos (dados de teste) capazes de revelar os defeitos dos programas. As informações conhecidas e específicas de cada programa são embutidas na representação cromossômica dos dados de teste e na função *fitness*, que determina a aptidão de cada indivíduo da população em resolver o problema em questão. Todas as técnicas são avaliadas por uma ou mais métricas (Seção 3.10), que, na maior parte das vezes, estão relacionadas com a forma de representação do programa.

A Figura 11 apresenta a relação entre a forma de representação dos programas sob teste e a técnica de teste de *software* utilizada. O tamanho das bolhas representa

a quantidade de estudos em cada categoria. Conforme mencionado anteriormente, os estudos que geram dados para teste estrutural utilizam principalmente GFC e DFG. Em contrapartida, os estudos que geram dados para teste funcional utilizam representações que incorporam especificações e requisitos de *software*. Os estudos que geram dados para teste de mutação utilizam o código-fonte diretamente.

Figura 11 – Relação entre técnicas de teste e formas de representação de programas. Eixo horizontal: Grafo de Fluxo de Controle (CFG — *Control Flow Graph*), *Combination Index Table*, código-fonte, DFG (*Data Flow Graph*), grafo, máquina de estados, requisitos de *software*, tabela de variáveis e de controles e *Unified Modelling Language*

Fonte: Davi Silva Rodrigues, 2017

A distribuição dos estudos por técnica de teste de *software* e por categorias é apresentada na Figura 12. O tamanho das bolhas novamente indica o número de estudos de cada categoria. Alguns estudos propuseram técnicas de geração de dados para teste funcional, no entanto, nenhum deles destacou a arquitetura do AG. Na geração de dados para teste estrutural, todos os tipos de adaptações foram destacados por, ao menos, um dos estudos. Todas as técnicas de geração de dados para teste de mutação destacaram as adaptações realizadas na função *fitness*. A arquitetura foi o aspecto menos destacado entre os estudos, enquanto a função *fitness* foi o mais destacado pelos estudos.

A necessidade de promover adaptações relacionadas ao problema é inerente aos AGs. Os autores dos estudos apresentados nas próximas seções destacaram, em seus artigos, adaptações de aspectos específicos que foram o foco de seus trabalhos e que tornaram

Figura 12 – Relação entre técnicas de teste e categorias de estudos. Eixo horizontal: algoritmos híbridos, arquitetura, função *fitness*, população e representação dos indivíduos

Fonte: Davi Silva Rodrigues, 2017

possível a obtenção de resultados satisfatórios na geração de dados para teste funcional, de mutação e estrutural.

3.5 Função *fitness*

As adaptações da função *fitness* foram as mais citadas nos estudos (32 estudos, 47%). Um total de 22 estudos geraram dados para teste estrutural, seis geraram dados para teste funcional e quatro geraram dados para teste de mutação.

3.5.1 Teste funcional

Betts e Petty (2016) propuseram o uso de um AG para gerar dados de teste para um sistema de controle de voo de *drones*. A função *fitness* é baseada no desvio lateral máximo de uma determinada trajetória. Os dados de teste gerados pelo AG, então, representam as trajetórias válidas que são mais difíceis de serem mantidas.

Núñez et al. (2013) geraram dados de teste para sistemas de tempo real, representados por máquinas de estados. A função *fitness* penaliza os dados de teste que não atendem às restrições temporais impostas por este tipo de sistema. Derderian et al. (2011) introduziram uma função *fitness* específica para máquinas de estado finitas que analisa

a probabilidade da ocorrência de uma transição de estado, além da complexidade das restrições temporais aplicáveis.

Vudatha et al. (2011) apresentaram uma técnica que gera dados de teste para sistemas embarcados. Huang, Cohen e Memon (2010) embutiram a viabilidade dos dados de teste na função *fitness*: os dados de teste, que representam ações realizadas em uma *Graphical User Interface* (GUI), são executados e penalizados caso sejam inviáveis (se geraram ações impossíveis de serem realizadas na GUI). Sharma, Pasala e Kommineni (2012) incluíram informações relacionadas às restrições impostas por requisitos funcionais na função *fitness*. Logo, a avaliação dos indivíduos penaliza os dados de teste que não respeitam as restrições.

3.5.2 Teste de mutação

Nestes quatro estudos, o *mutation score* é utilizado como ou relacionado à função *fitness*. No estudo desenvolvido por Khan e Amjad (2015), inicialmente, os dados de teste são gerados aleatoriamente e, então, o *mutation score* é calculado. Quando os dados de teste iniciais não atingem um *score* mínimo, a execução do AG é iniciada e esses dados constituem a população inicial.

Hanh, Binh e Tung (2016) introduziram uma função *fitness* para avaliar especificamente a propagação de erros ao longo dos caminhos de execução de um programa e para avaliar a capacidade dos dados de teste de matar mutantes.

Rani e Suri (2015) substituíram a função *fitness* pela execução de programas mutantes: caso a população atual não mate todos os mutantes (não revele todos os defeitos), os operadores de *crossover* e de mutação são utilizados, o que implica na evolução da população.

A função *fitness* da técnica proposta por Yao, Gong e Zhang (2015) prioriza a cobertura de instruções e os dados de teste que estão mais distribuídos pelo domínio de entrada do programa sob teste.

3.5.3 Teste estrutural

De todos os estudos incluídos, 21 destacaram a adaptação feita na função *fitness* para gerar dados para teste estrutural. Os autores apresentaram cinco tipos de funções *fitness*: multi-objetivo (LAKHOTIA; HARMAN; MCMINN, 2007; PINTO; VERGILIO, 2010; VARSHNEY; MEHROTRA, 2016; ZHANG; GONG, 2010), multi-caminhos (AHMED; HERMADI, 2008; AHMED; ALI, 2015; ALEB; KECHID, 2013; CAO; HU; LI, 2009; CHEN; ZHONG, 2009; GHIDUK; HARROLD; GIRGIS, 2007; GHIDUK; GIRGIS, 2010; HERMADI; AHMED, 2003; JIN; JIANG; ZHANG, 2011; SUN; JIANG, 2010; XIBO; NA, 2011; YAO; GONG; WANG, 2015), de caminho único (KHAN; AMJAD; SRIVASTAVA, 2016; MANN et al., 2016; PARTHIBAN; SUMALATHA, 2013; SHIMIN; ZHANGANG, 2011), baseada em cobertura de código (SHUAI et al., 2015) e uma especificamente para determinar se um dado de teste é capaz de lançar exceções em tempo de execução (SOLTANI; PANICHELLA; DEURSEN, 2016).

A maioria desses estudos propõe funções *fitness* baseadas no GFC dos programas (AHMED; HERMADI, 2008; AHMED; ALI, 2015; CAO; HU; LI, 2009; CHEN; ZHONG, 2009; GHIDUK; GIRGIS, 2010; LAKHOTIA; HARMAN; MCMINN, 2007; PINTO; VERGILIO, 2010; SHIMIN; ZHANGANG, 2011; SUN; JIANG, 2010; YAO; GONG; WANG, 2015). Outras técnicas usam diferentes representações de programas como DFG (KHAN; AMJAD; SRIVASTAVA, 2016; PINTO; VERGILIO, 2010; VARSHNEY; MEHROTRA, 2016), tabela de controles/variáveis (ALEB; KECHID, 2013), CDG (*Control-Dependence Graph*) (JIN; JIANG; ZHANG, 2011) ou o próprio código-fonte (AHMED; ALI, 2015; GHIDUK; HARROLD; GIRGIS, 2007; HERMADI; AHMED, 2003; MANN et al., 2016; PARTHIBAN; SUMALATHA, 2013; SHUAI et al., 2015; SOLTANI; PANICHELLA; DEURSEN, 2016; XIBO; NA, 2011; ZHANG; GONG, 2010). Jin, Jiang e Zhang (2011) definiram CDG como uma extensão do GFC que acrescenta informações referentes à dependência de controles.

Técnicas que apresentam funções *fitness* multi-objetivo otimizaram dados de teste em termos de alocação de memória, tempo de execução e número de erros revelados a fim de maximizar a qualidade dos resultados de cada execução do AG (LAKHOTIA; HARMAN; MCMINN, 2007; PINTO; VERGILIO, 2010). Apesar de ter como objetivo a cobertura de múltiplos caminhos, o estudo realizado por Varshney e Mehrotra (2016)

tenta otimizar as soluções candidatas para os critérios de fluxo de dados todos-usos e todas-definições (o estudo utiliza DFG como forma de representação do programa sob teste).

Similarmente, as técnicas que propõem a cobertura de múltiplos caminhos buscam gerar dados de teste eficientemente. Ghiduk e Girgis (2010) definiram uma função *fitness* que maximiza a cobertura de caminhos de um GFC, mas analisa apenas a dominância de caminhos. Yao, Gong e Wang (2015) propuseram uma técnica de geração de dados de teste baseada em duas funções *fitness*: uma multi-caminhos, que divide os indivíduos em grupos, e outra, que determina a aptidão de cada indivíduo. Hermadi e Ahmed (2003) consideraram a distância dos dados de teste e suas violações relativas a diferentes caminhos e embutiram ambas as informações na função *fitness*. Zhang e Gong (2010) propuseram uma função *fitness* que determina se os dados de teste são capazes de cobrir múltiplos caminhos e de revelar defeitos.

Mann et al. (2016) introduziram uma função *fitness* específica para um único caminho de execução. Funções matemáticas representam condições presentes no código-fonte e indicam se os dados de teste são capazes de atender a essas condições. Os resultados das funções matemáticas são atribuídos aos indivíduos como valor de aptidão. Khan, Amjad e Srivastava (2016) apresentaram uma técnica que gera dados de teste capazes de cobrir todos os caminhos de um DFG utilizado como forma de representação do programa sob teste

Soltani, Panichella e Deursen (2016) usaram um AG para reproduzir erros conhecidos de um programa. A função *fitness* proposta verifica se uma linha de código que lança exceções é coberta pelo dado de teste e se a pilha de execução gerada é similar à pilha gerada pelo programa original. Shuai et al. (2015) propuseram uma técnica de geração de dados de teste que revela vulnerabilidades do *software*. A função *fitness* atribui valores de aptidão mais altos para os dados de teste que maximizam a cobertura de código sem aumentar os custos de execução. Os autores utilizaram o código-fonte diretamente em vez de GFC ou DFG como forma de representação do programa.

3.6 População

A adaptação da população foi destacada por 11 estudos (16% do total). Dois estudos propuseram técnicas de geração de dados para teste funcional e nove estudos geraram dados para teste estrutural.

3.6.1 Teste funcional

Ambas as técnicas que geraram dados para teste funcional utilizaram especificações em vez de grafos ou código-fonte como forma de representação do programa sob teste.

Fischer e Tönjes (2012) propuseram os micro algoritmos genéticos (μ AGs), que “matam” os indivíduos menos aptos a cada intervalo específico de gerações (a cada dez gerações, por exemplo). Essa técnica utiliza os requisitos do *software* como forma de representação do programa.

Zhou, Zhao e You (2014) utilizaram três populações que evoluem de modo independente, mas permitiram que houvesse migração de indivíduos entre as populações. Dois dos melhores indivíduos (de acordo com a função *fitness*) de duas populações substituem os dois piores indivíduos de uma terceira população; e os programas são representados por máquinas de estados.

3.6.2 Teste estrutural

Os autores dos estudos aqui apresentados propuseram o uso de múltiplas populações (ALSHRAIDEH; MAHAFZAH; AL-SHARAEH, 2011; CHEN; ZHONG, 2008; DEEPAK; SAMUEL, 2009; GONG; ZHANG; YAO, 2011; TIAN; GONG, 2016; ZHANG et al., 2010; ZHANG; WU; BAO, 2015), regeneração de população (YANG et al., 2016) e compartilhamento de indivíduos (YAO; GONG, 2014). A maior parte das nove técnicas utiliza GFC como forma de representação (ALSHRAIDEH; MAHAFZAH; AL-SHARAEH, 2011; CHEN; ZHONG, 2008; DEEPAK; SAMUEL, 2009; GONG; ZHANG; YAO, 2011; TIAN; GONG, 2016; YAO; GONG, 2014). Os demais estudos utilizam DFG (ZHANG; WU; BAO, 2015) ou o próprio código-fonte (YANG et al., 2016).

As técnicas propostas por Chen e Zhong (2008) e Zhang, Wu e Bao (2015) utilizam três populações independentes que permitem a migração de indivíduos entre si. Os dois melhores indivíduos de duas populações (em termos de aptidão) substituem os dois piores indivíduos da terceira população. A técnica introduzida por Yao e Gong (2014) também utiliza múltiplas populações, porém elas evoluem conjuntamente. A função *fitness* de cada população avalia os indivíduos de todas as populações e atribui aos indivíduos um valor de aptidão global, o que aumenta as chances de se encontrar uma solução ótima para o problema. Alshraideh, Mahafzah e Al-Sharaeh (2011) e Deepak e Samuel (2009) apresentaram AGs com migração aleatória de indivíduos, independentemente do valor de aptidão atribuído pela função *fitness*.

Usando o critério de cobertura de ramificações e um fator de envelhecimento da população, Yang et al. (2016) definiram um AG capaz de identificar se a população atual é um ótimo local. Se o AG de fato estiver em um ótimo local, então a população atual é descartada e uma nova é gerada aleatoriamente.

Gong, Zhang e Yao (2011) e Zhang et al. (2010) resolveram o problema da geração de dados para teste de múltiplos caminhos dividindo os caminhos-alvo em grupos e processando cada um deles como uma subpopulação independente. Tian e Gong (2016) propuseram um tipo de AG coevolucionário que utiliza múltiplas subpopulações independentes. Os melhores candidatos gerados pelas subpopulações constituem uma população inicial cooperativa.

3.7 Representação dos indivíduos

Dez estudos (15% do total) destacaram as codificações de soluções candidatas propostas. Quatro estudos propuseram técnicas de geração de dados para teste funcional e seis estudos geraram dados para teste estrutural.

3.7.1 Teste funcional

Lefticaru e Ipate (2007) utilizaram diagramas UML de máquinas de estados como forma de abstrair os programas e adaptaram a representação dos indivíduos no AG. A população de soluções candidatas é composta por listas de parâmetros de entrada que exercitam caminhos específicos em um grafo de transições de estados. Arora e Sinha (2014)

introduziram uma técnica de geração de dados de teste também baseada em máquinas de estados. Os autores definiram cromossomos de tamanho variável, que está diretamente relacionado ao número de transições de estado.

Alsmadi (2010) codificou elementos de GUIs em cromossomos binários que representam as posições vertical e horizontal desses elementos. Adicionalmente, o autor propôs uma estrutura de dados que atribui dois cromossomos para cada elemento da GUI (um para a posição vertical e outro para a posição horizontal). Peng e Lu (2011) codificaram requisições *web* em cromossomos cujos genes são compostos por dois tipos de estruturas de dados (nós e páginas). Esses genes não são limitados a um alfabeto binário e armazenam diversas informações de interesse.

3.7.2 Teste estrutural

As técnicas específicas para teste estrutural utilizam como forma de representação dos programas sob teste GFC (KHANDELWAL; TOMAR, 2015; LIU; WANG; WANG, 2013) ou o próprio código-fonte (KOLEEJAN; XUE; ZHANG, 2015; LIJUAN; YUE; HONGFENG, 2012; SILVA; VAN SOMEREN, 2010; TONELLA, 2004).

Khandelwal e Tomar (2015) apresentaram uma abordagem para a geração de dados para teste de programas orientados a aspectos em que as soluções candidatas são definidas pelos caminhos exercitados de um GFC. Liu, Wang e Wang (2013) utilizaram cromossomos com números reais a fim de manter a precisão das soluções candidatas requerida pelo problema.

Lijuan, Yue e Hongfeng (2012) definiram um novo processo de geração da população inicial: as particularidades do domínio de entrada são consideradas na criação das soluções iniciais. Para um parâmetro inteiro que costuma ser negativo, por exemplo, as soluções iniciais criadas serão predominantemente negativas em vez de positivas. Koleejan, Xue e Zhang (2015) propôs uma representação multivetor dos indivíduos; desse modo, um cromossomo codifica mais de uma solução candidata.

Silva e van Someren (2010) e Tonella (2004) propuseram codificações específicas para representação de objetos e métodos: os indivíduos armazenam informações como tipo de retorno de método e tipos válidos de parâmetros.

3.8 Arquitetura do algoritmo genético

A adaptação da arquitetura foi a menos citada nos estudos incluídos (6% do total). Todos os quatro estudos propuseram técnicas de geração de dados para teste estrutural: duas técnicas utilizam GFC como representação do programa e as outras duas utilizam o próprio código-fonte.

Ao criar um novo passo na execução do algoritmo, Aleti e Grunske (2015) propuseram AGs capazes de ajustar seus parâmetros genéticos durante o processo de otimização. Manikumar, Kumar e Maruthamuthu (2016) introduziram uma técnica baseada em dois AGs: o primeiro gera dados de teste especificamente para testar os nós de um GFC e o segundo usa esses dados como população inicial. Então, uma nova população (otimizada para o GFC inteiro) é gerada.

El-Serafy et al. (2015) propuseram uma técnica de geração de dados de teste que também utiliza dois AGs. O primeiro gera dados de teste de acordo com o critério de teste estrutural *Modified Condition/Decision Coverage* (MC/DC). O segundo considera dados de teste existentes como problemas a serem resolvidos e, portanto, gera novos dados de teste adequados a cada novo problema. Pachauri e Srivasatava (2015) desenvolveram uma técnica baseada em AGs paralelos. Todos os AGs evoluem a mesma população, porém cada um possui uma função *fitness* diferente.

3.9 Algoritmos híbridos

Onze estudos (16% do total) combinaram AGs com outras técnicas. Esses estudos não destacaram nenhuma adaptação. A maior parte desses estudos gera dados para teste estrutural e utilizam *strings* como dados de entrada. Os estudos realizados por Alesio et al. (2015) e Lin et al. (2012) geraram dados para teste funcional. Hanh, Binh e Tung (2014) geraram dados para teste de mutação.

As técnicas híbridas propostas adicionaram um novo passo aos AGs (BOUCHACHIA, 2007; JI; SUN, 2012; KIFETEW et al., 2013; LIN et al., 2012; TAN; LONGXIN; XIUMEI, 2009), utilizaram AGs como um passo de um processo maior (ALESIO et al., 2015; MILLER; REFORMAT; ZHANG, 2006) ou aumentaram a eficiência de operadores genéticos (FAN et al., 2015; GARG; GARG, 2015; MAYAN; RAVI, 2014). Todos os estudos

foram desenvolvidos para sistemas de propósitos gerais, com exceção da técnica proposta por Alesio et al. (2015), que foi desenvolvida para sistemas embarcados de tempo real.

Algoritmos genéticos foram combinados com técnicas de resfriamento simulado (JI; SUN, 2012; HANH; BINH; TUNG, 2014; TAN; LONGXIN; XIUMEI, 2009), tabela de cobertura (MILLER; REFORMAT; ZHANG, 2006), *combination-index table* (CIT) (LIN et al., 2012), programação com restrições (ALESIO et al., 2015), sistemas imunes artificiais (BOUCHACHIA, 2007; HANH; BINH; TUNG, 2014; TAN; LONGXIN; XIUMEI, 2009) e decomposição em valores singulares (KIFETEW et al., 2013).

Fan et al. (2015) e Mayan e Ravi (2014) substituíram o operador de mutação por uma busca Tabu. Garg e Garg (2015) utilizaram subida da montanha (*hill climbing*) no operador de seleção a fim de obter melhores resultados. Hanh, Binh e Tung (2014) combinaram AG, resfriamento simulado e sistemas imunes artificiais para gerar soluções mais adequadas em termos de *mutation score*.

3.10 Métricas de avaliação para técnicas de geração de dados de teste

Todos os estudos considerados neste mapeamento sistemático utilizam métricas específicas para avaliar os dados de teste gerados. Essas métricas possibilitam aferir a qualidade dos dados de teste, a viabilidade do tempo necessário para gerá-los, além da eficiência e da eficácia das técnicas propostas quando comparadas a outras (geração aleatória, por exemplo). As métricas de avaliação utilizadas por cada estudo podem ser vistas nas Tabelas 2, 3 e 4.

A Figura 13 apresenta a relação entre as categorias de estudos e as métricas de avaliação utilizadas por cada técnica. As métricas mais utilizadas são o número de gerações (iterações do AG) e o tempo de execução, especialmente em AGs que destacaram as adaptações da função *fitness*. As únicas métricas utilizadas por estudos de todas as categorias são cobertura de caminhos, cobertura de ramificações e tempo de execução.

As métricas de avaliação de técnicas de geração de dados para teste estrutural podem ser classificadas em três grupos: métricas baseadas na representação do programa, baseadas nos AGs e baseadas no programa sob teste.

- **Métricas de avaliação baseadas em GFC/DFG:** cobertura de ramificações (ALETI; GRUNSKE, 2015; KIFETEW et al., 2013; LAKHOTIA; HARMAN; MC-

Figura 13 – Relação entre categorias de estudos e métricas de avaliação das técnicas de geração de dados de teste por meio de AGs. Eixo horizontal: algoritmos híbridos, arquitetura, função *fitness*, população e representação dos indivíduos

Fonte: Davi Silva Rodrigues, 2017

MINN, 2007; MANIKUMAR; KUMAR; MARUTHAMUTHU, 2016), cobertura de ramificações viáveis (LIJUAN; YUE; HONGFENG, 2012), cobertura de caminhos (AHMED; ALI, 2015; CHEN; ZHONG, 2008; DEEPAK; SAMUEL, 2009; JI; SUN, 2012; KHAN; AMJAD; SRIVASTAVA, 2016), cobertura de instruções (GHIDUK; GIRGIS, 2010; KOLEEJAN; XUE; ZHANG, 2015; MAYAN; RAVI, 2014; MILLER; REFORMAT; ZHANG, 2006), cobertura de decisões (FAN et al., 2015) e cobertura de condições (BOUCHACHIA, 2007).

- **Métricas de avaliação baseadas nos AGs:** tempo de execução (ALSHRAIDEH; MAHAFZAH; AL-SHARAEH, 2011; CAO; HU; LI, 2009; EL-SERAFY et al., 2015;

GHIDUK; HARROLD; GIRGIS, 2007; SHIMIN; ZHANGANG, 2011; SHUAI et al., 2015; ZHANG et al., 2010), número de indivíduos gerados (ALEB; KECHID, 2013; GARG; GARG, 2015; GHIDUK; HARROLD; GIRGIS, 2007), número de gerações (ALEB; KECHID, 2013; BOUCHACHIA, 2007; CAO; HU; LI, 2009; HERMADI; AHMED, 2003; JI; SUN, 2012; ZHANG; GONG, 2010), número de execuções da função *fitness* (GONG; ZHANG; YAO, 2011; TIAN; GONG, 2016) e número de erros revelados (SILVA; VAN SOMEREN, 2010).

- **Métricas de avaliação baseadas no programa sob teste:** número de erros reproduzidos (SOLTANI; PANICHELLA; DEURSEN, 2016).

A avaliação das técnicas de geração de dados para teste funcional é baseada na cobertura de caminhos do GFC (LEFTICARU; IPATE, 2007), dados de teste viáveis (HUANG; COHEN; MEMON, 2010), combinações (LIN et al., 2012), estados (DERDERIAN et al., 2011; NÚÑEZ et al., 2013), transição de estados (PENG; LU, 2011), número de gerações (SHARMA; PASALA; KOMMINENI, 2012; VUDATHA et al., 2011; ZHOU; ZHAO; YOU, 2014), número de erros revelados (ALSMADI, 2010; ARORA; SINHA, 2014), desvio lateral máximo (BETTS; PETTY, 2016), proporção entre dados de teste válidos/inválidos (FISCHER; TÖNJES, 2012) e tempo de execução (ALESIO et al., 2015).

As técnicas de geração de dados para teste de mutação utilizam *mutation score* (HANH; BINH; TUNG, 2014; HANH; BINH; TUNG, 2016; KHAN; AMJAD, 2015; YAO; GONG; ZHANG, 2015), número de mutantes criados (RANI; SURI, 2015) e tempo de execução (HANH; BINH; TUNG, 2014).

3.11 Discussão, desafios e oportunidades

Nos últimos 14 anos, aumentou a quantidade de estudos publicados no tópico de geração de dados de teste por meio de algoritmos genéticos. Os estudos incluídos na etapa de extração do mapeamento sistemático foram publicados entre os anos de 2003 e 2016. Os estudos da década de 2000, em sua maioria, buscam afirmar a eficiência dos algoritmos genéticos na solução do problema da geração de dados de teste. Os estudos mais recentes, entretanto, já reconhecem os AGs como ferramentas poderosas na automatização da geração de dados de teste e propõem refinamentos, melhorias e novas aplicações desses algoritmos.

De modo geral, as técnicas de geração de dados de teste são aplicadas com maior frequência ao teste estrutural de *software*. Todavia, independentemente do tipo de teste, o AG utiliza o código-fonte ou uma representação do programa (GFC, DFG, máquina de estados finitos, diagramas UML, entre outros) que, na maior parte das vezes, está relacionada à métrica de avaliação da técnica. Deve ser definida uma codificação das soluções candidatas, isto é, uma representação cromossômica dos dados de teste. Também deve ser definida uma função *fitness* responsável por aferir a aptidão de cada indivíduo da população. Os operadores genéticos de *crossover* e mutação são os responsáveis por promover a variedade genética (a “movimentação” do AG entre as soluções no espaço de busca).

3.11.1 Adaptação ao problema

Os AGs precisam ser adaptados ao problema que se deseja resolver para que sejam o mais eficazes possível. Apesar de cada um dos componentes básicos dos AGs (população, seleção e operadores de *crossover* e mutação) exigirem adaptações, a maior parte das técnicas detalhadas nas seções anteriores destacaram a função *fitness*, cuja adaptação está diretamente relacionada ao problema.

A necessidade de adaptar os componentes de um AG ao problema de interesse oferece vantagens e desvantagens. É necessário conhecer o máximo possível sobre o problema para que os componentes do AG possam embutir este conhecimento e terem maiores chances de gerar dados de teste de maior qualidade. Adicionalmente, problemas diferentes precisam de adaptações diferentes, o que pode exigir esforço constante dos desenvolvedores até que se alcancem bons resultados para problemas específicos. Entretanto, é possível que problemas semelhantes possam compartilhar um ou mais componentes de um AG com poucas alterações. Portanto, a arquitetura bem definida de um AG facilita a sua implementação e viabiliza o reúso de componentes quando tecnologias adequadas — como linguagens de programação orientadas a objetos — são utilizadas. Consequentemente, a principal vantagem obtida ao se adaptar um AG é embutir o conhecimento relacionado ao problema na própria implementação do algoritmo (desde que os componentes sejam devidamente implementados e testados). Dessa forma, cobre-se uma classe inteira de

problemas similares dentro de um mesmo domínio em vez de atender apenas um problema, o que minimiza os esforços de desenvolvimento.

3.11.2 Dados complexos e sistemas grandes

Sistemas de informação cada vez mais incorporam dados multimídia (imagens, vídeos e sons) e informações tridimensionais. No entanto, a maioria das aplicações de AGs na geração de dados de teste encontradas neste mapeamento sistemático envolve domínios de entrada e de saída triviais (caracteres, números, símbolos, etc.). Nenhum dos estudos incluídos abordaram o problema da geração de dados de teste em domínios complexos como o de imagens e o de dados multimídia. As representações cromossômicas propostas são adequadas para dados triviais, porém não são capazes de manipular dados complexos de modo eficiente.

O desenvolvimento de abordagens que utilizam AGs para gerar dados de teste em domínios complexos requer, na maioria das vezes, mudanças profundas em diversos componentes do algoritmo, dado que todos eles devem ser adequados a um domínio bastante específico. Não se pode, por exemplo, executar uma operação de *crossover* em imagens sem a garantia de que o indivíduo resultante seja outra imagem válida. Similarmente, a geração de uma população inicial aleatória pode resultar em imagens inválidas e, talvez, essa classe de programas exija que uma população inicial adequada seja fornecida. Portanto, os operadores genéticos, assim como as funções *fitness* e os outros componentes, devem ser projetados para trabalhar somente com entradas e saídas válidas. Conclui-se, então, que sistemas que possuem dados de entrada e saída complexos constituem uma oportunidade de pesquisa pouco explorada.

Além disso, os estudos discutidos neste mapeamento propõem técnicas de geração de dados de teste para sistemas pequenos e, até mesmo, implementações de métodos nativos da linguagem de programação Java. Desse modo, essas técnicas podem não ter um desempenho aceitável quando aplicadas a sistemas grandes. Consequentemente, o uso de sistemas grandes na avaliação das técnicas e o desenvolvimento de novas técnicas mais eficientes para essa classe de sistemas também são boas oportunidades de pesquisa.

Apesar de terem sido considerados apenas sistemas pequenos e dados de teste triviais, as métricas de avaliação mais utilizadas pelos estudos foram o tempo de execução

e a quantidade de gerações do AG. Isso indica que um dos desafios na geração de dados de teste por meio de AGs é automatizar (de modo eficiente e com um tempo de execução factível) o teste de sistemas grandes que requerem dados de teste complexos. São esses os sistemas presentes na indústria, que precisa da ajuda de pesquisadores nas atividades de teste e validação.

3.11.3 Tempo de execução e métricas de avaliação

Algoritmos genéticos podem exigir tempos de execução muito longos. Alguns estudos propuseram o uso de AGs paralelos e outros apresentaram AGs com múltiplas populações. Esse problema piora quando domínios complexos são considerados, conforme dito anteriormente. Trabalhos futuros podem propor representações cromossômicas específicas e funções *fitness* que otimizem o tempo de execução, visto que funções *fitness* são utilizadas muitas vezes ao longo da execução do AG e manipulam parâmetros codificados.

A maioria dos estudos utilizou o tempo de execução e o número de gerações como métricas de avaliação das técnicas de geração de dados de teste propostas. Dado que essas métricas foram obtidas por meio de testes realizados com sistemas pequenos, elas não são capazes de indicar se sistemas grandes com dados complexos podem ser propriamente testados com as técnicas propostas. A definição de um *benchmark* padrão para programas que utilizam dados complexos tornaria as métricas de avaliação mais significativas.

3.11.4 Ambientes virtuais tridimensionais

Ambientes virtuais, compostos por modelos tridimensionais (3D), viabilizam a interação humano-computador em três dimensões e em tempo real. Usualmente, ambientes virtuais simulam situações reais como, por exemplo, treinamentos, visualização, entretenimento, etc. A presença desse tipo de sistema tem aumentado na sociedade devido a fatores como amplo acesso à tecnologia, popularização de jogos e desenvolvimento de dispositivos que melhoram a interação e oferecem visualização 3D.

Desse modo, são inúmeros as possíveis ações realizadas pelos usuários para atingir um objetivo e os comportamentos dos sistemas. Isso dificulta a geração de entradas e saídas para a composição de casos de teste. Modelos tridimensionais podem ter comportamentos

autônomos (movimentos predefinidos) e são dependentes das ações do usuário (deformação de um modelo após a colisão com um outro modelo manipulado pelo usuário, por exemplo). Além disso, a automação das atividades de teste deve considerar a execução em tempo real e paralela. Algoritmos genéticos são capazes de fazer buscas em espaços de soluções grandes e complexos e, assim, podem encontrar formas de interações humano-computador relevantes em ambientes virtuais, envolvendo a busca em um universo de incontáveis ações do usuário e comportamentos de modelos 3D. Ao indicar as ações e comportamentos críticos, AGs poderiam facilitar a geração de dados de teste para sistemas de realidade virtual.

3.12 Considerações finais

Algoritmos genéticos têm sido utilizados com sucesso na geração de dados de teste. Esses algoritmos viabilizam a busca em um espaço de soluções grande, visto que são não-determinísticos. No entanto, o problema sob análise deve ser profundamente conhecido e esse conhecimento deve ser embutido nos componentes do AG.

Os estudos analisados por este mapeamento sistemático, em sua maioria, possuem domínios de baixa complexidade e, mesmo assim, consideram o tempo de execução como uma das principais métricas de avaliação utilizadas na aferição de qualidade das técnicas propostas.

Gerar dados de teste complexos dentro de um tempo de execução razoável e, ao mesmo tempo, usando representações cromossômicas que garantam a integridade dos dados é o grande desafio da pesquisa em geração de dados de teste por meio de algoritmos genéticos.

Neste capítulo foi apresentado um MS da literatura no tópico de geração de dados de teste por meio de AGs. Este mapeamento, em conjunto com a teoria apresentada no Capítulo 2, constituíram a base sobre a qual foi definida a abordagem proposta por esta pesquisa e que é apresentada no Capítulo 5.

4 Procedimentos metodológicos

Neste capítulo são apresentados os métodos de pesquisa empregados a fim de se atingir os objetivos citados na Seção 1.2. Este trabalho foi realizado em três etapas: revisão da literatura, desenvolvimento e validação experimental (Figura 14).

Figura 14 – Procedimentos metodológicos utilizados neste trabalho

Fonte: Davi Silva Rodrigues, 2017

Durante a etapa de revisão de literatura, foram conduzidas uma análise exploratória e um mapeamento sistemático (Seção 4.1) que nortearam o desenvolvimento de uma abordagem de geração de dados de teste para programas de PI (Seção 4.2). Paralelamente à etapa de desenvolvimento, foram definidas métricas de avaliação e experimentos para a validação da abordagem proposta (Seção 4.3), o que resultou na confirmação parcial da hipótese levantada na Seção 1.2.

4.1 Revisão da literatura

A etapa inicial deste trabalho incluiu uma análise exploratória da literatura e um mapeamento sistemático. A análise exploratória (Seção 3.1) teve como objetivo fornecer uma visão inicial da área de geração de dados de teste no domínio de imagens. Os resultados dessa análise evidenciaram a ausência de técnicas específicas para o domínio de imagens e o uso frequente de algoritmos genéticos para geração de dados de teste em outros domínios (programas escritos em linguagem C, *web services* e sistemas orientados a objetos, por exemplo).

Esses resultados motivaram a condução de um mapeamento sistemático da literatura no tópico de geração de dados de teste por meio de algoritmos genéticos. Os métodos utilizados na condução do mapeamento (e os resultados obtidos) são detalhados no Capítulo 3. A condução do mapeamento resultou em uma visão geral do estado da arte na área de pesquisa em questão. A análise dos artigos encontrados revelou quais componentes do AG sofrem maiores adaptações, como esses componentes são modificados, como as abordagens são avaliadas, quais métricas de avaliação são mais utilizadas, quais são as limitações das abordagens e quais são as lacunas de pesquisa existentes. O mapeamento sistemático também originou um artigo para periódico (RODRIGUES et al., 2017).

Os resultados citados reafirmaram as conclusões da análise exploratória e forneceram os insumos necessários para a definição de uma nova abordagem de geração de dados de teste para programas de PI.

4.2 *Desenvolvimento da abordagem proposta*

A etapa de desenvolvimento da abordagem proposta foi dividida em estudo de APIs (*Application Programming Interfaces*), definição de operadores genéticos e definição da função *fitness* do AG.

4.2.1 Tecnologias utilizadas

Desde o início, decidiu-se utilizar APIs existentes que fornecessem implementações padrão de algoritmos genéticos que pudessem ser estendidas. Dessa forma, o foco da etapa de desenvolvimento foi a adaptação dos componentes do AG ao domínio de imagens. Por essa razão, o principal critério utilizado na escolha da API foi a flexibilidade oferecida na customização dos componentes e parâmetros do AG.

A abordagem foi implementada em Java. Por isso, foram investigadas diversas bibliotecas especializadas em algoritmos genéticos como, por exemplo, Jenetics (WILHELMSTÖTTER, 2017), JGAP (*Java Genetic Algorithms Package*) (MEFFERT, 2015), ECJ (*Evolutionary Computation for Java*) (LUKE, 2017) e *Watchmaker Framework*.

A API *Watchmaker*, criada por Dyer (2010), foi utilizada neste trabalho por oferecer maior flexibilidade no tratamento da população inicial.

Os testes de mutação realizados pela abordagem proposta utilizaram programas mutantes gerados pela ferramenta *muJava*, criada por Ma, Offutt e Kwon (2005). Essa ferramenta permite gerar programas mutantes, realizar o teste de mutação e obter o *mutation score*. A abordagem proposta utiliza apenas o módulo de geração de mutantes, que dispõe dos operadores de mutação apresentados na Tabela 5.

Tabela 5 – Operadores de mutação da ferramenta *muJava* utilizados pela abordagem TAIGA

Categoria	Operador	Descrição
Operadores aritméticos	AOR	<i>Arithmetic Operator Replacement</i>
	AOI	<i>Arithmetic Operator Insertion</i>
	AOD	<i>Arithmetic Operator Deletion</i>
Operadores relacionais	ROR	<i>Relational Operator Replacement</i>
Operadores condicionais	COR	<i>Conditional Operator Replacement</i>
	COI	<i>Conditional Operator Insertion</i>
	COD	<i>Conditional Operator Deletion</i>
Operadores de <i>shift</i>	SOR	<i>Shift Operator Replacement</i>
Operadores lógicos	LOR	<i>Logical Operator Replacement</i>
	LOI	<i>Logical Operator Insertion</i>
	LOD	<i>Logical Operator Deletion</i>
Operadores de atribuição	ASR	<i>Assignment Operator Replacement</i>
Operadores de deleção	SDL	<i>Statement Deletion</i>
	VDL	<i>Variable Deletion</i>
	CDL	<i>Constant Deletion</i>
	ODL	<i>Operator Deletion</i>

Fonte: adaptado de MA; OFFUTT, 2005

Os operadores de mutação podem ser divididos em sete categorias, a saber: operadores aritméticos, operadores relacionais, operadores condicionais, operadores de *shift*, operadores lógicos, operadores de atribuição e operadores de deleção. Cada uma das categorias é associada a um tipo de operador suportado em Java. Os operadores de mutação relacionais, por exemplo, podem incluir, excluir ou trocar os operadores $<$, $<=$, $=$, $>=$, $>$ e $!=$ em programas que os utilizem.

4.2.2 Definição das adaptações realizadas no algoritmo genético

Paralelamente à escolha da API, iniciou-se a definição das adaptações que foram feitas nos componentes do AG. Foram adaptados os seguintes componentes: codificação das soluções candidatas, população inicial, função *fitness*, operador de mutação e operador de *crossover*.

A adaptação desses componentes foi guiada pelos resultados do mapeamento sistemático. No entanto, todos os artigos incluídos no mapeamento propõem abordagens compatíveis com dados triviais, isto é, dados compostos por números e/ou cadeias de caracteres (Seção 3.4). Essas abordagens possuem as seguintes características em comum no que se refere aos componentes do AG:

- as soluções candidatas são codificadas em caracteres alfanuméricos e são representadas por estruturas de dados simples na implementação;
- a população inicial, constituída por dados triviais, é gerada aleatoriamente;
- a função *fitness*, atrelada ao problema sob análise, é implementada como cálculo de fórmulas ou como outras operações relativamente simples;
- o operador genético de mutação é implementado como troca aleatória de valores em posições específicas da estrutura de dados e,
- o operador de *crossover* cria uma nova estrutura de dados a partir de outras duas.

Imagens, por serem dados complexos, possuem restrições que dificultam ou até mesmo inviabilizam o uso de componentes com essas características. Por essa razão, as adaptações no AG propostas por este trabalho são diferentes daquelas utilizadas nos artigos incluídos no mapeamento sistemático. As adaptações aqui propostas respeitam as particularidades do domínio de imagens e, ao mesmo tempo, mantêm as características básicas do AG (Seção 2.5). As particularidades do domínio de imagens e as adaptações realizadas nos componentes do AG, que culminaram na abordagem TAIGA, são apresentadas no Capítulo 5.

4.3 Validação experimental

Paralelamente à etapa de desenvolvimento, foram definidos os experimentos e as métricas de avaliação que forneceram os insumos para a confirmação parcial da hipótese deste trabalho.

O objetivo desta etapa foi definir imagens de teste iniciais, programas de PI, experimentos e métricas capazes de indicar se a abordagem proposta apresenta maior eficácia quando comparada ao teste tradicional realizado em programas de PI.

As métricas de avaliação foram definidas de modo a viabilizar a comparação da abordagem proposta com outras abordagens que utilizam AGs para gerar dados de teste (ainda que em outros domínios). Foram utilizadas as seguintes métricas de avaliação: tempo de execução, número de gerações do AG e *mutation score*. As duas primeiras foram selecionadas por serem as mais utilizadas na avaliação de abordagens baseadas em AGs (Seção 3.10). A terceira métrica é apresentada na Seção 2.2.1.

A abordagem TAIGA foi desenvolvida com o objetivo de gerar, a partir de uma imagem inicial, um conjunto de imagens mais adequadas para o teste, ou seja, capazes de encontrar o maior número de defeitos. A imagem de entrada pode ter características que a tornem adequada ao programa de PI sob teste ou pode ser, por exemplo, uma imagem aleatória recuperada de um banco de imagens. Diferentemente da maioria das implementações de AGs, a abordagem TAIGA requer que o testador forneça imagens iniciais em vez de gerar uma população inicial aleatória. *Pixels* cujos valores de intensidade sejam determinados aleatoriamente podem ou não constituir uma imagem válida (que tenha algum significado para extração de informação) ou inserir ruído excessivo a ponto de inviabilizar a execução dos programas de PI.

Por esse motivo, ao longo do desenvolvimento e da validação da abordagem foram utilizadas quatro imagens: uma imagem preto e branco (Figura 15a), uma imagem em níveis de cinza (Figura 15b), uma imagem apenas com fundo preto (Figura 15c) e uma impressão digital (Figura 15d). Este conjunto de imagens possui, pelo menos, uma imagem adequada para cada um dos programas de PI que foram utilizados. Foram selecionadas imagens de pequenas dimensões — 100 x 100 *pixels* (Figuras 15a, 15b, 15c) e 240 x 336 *pixels* (Figura 15d) — para que as execuções do AG tornassem factível a coleta de resultados.

Figura 15 – Imagens de teste iniciais utilizadas nas etapas de desenvolvimento e de validação da abordagem

Fonte: (a)(b)(c) Davi Silva Rodrigues, 2017 (d) PERENHA, 2003

Para que a abordagem proposta fosse capaz de gerar imagens de teste para diferentes programas de PI, foram implementados em Java oito programas para serem utilizados durante as etapas de desenvolvimento e de validação (Tabela 6). Os parâmetros utilizados em cada programa são apresentados na Tabela 7.

Tabela 6 – Programas de PI utilizados nas etapas de desenvolvimento e validação da abordagem TAIGA apresentados em ordem de complexidade (linhas de código)

Programa	Descrição	Linhas
Média	Executa suavização da imagem por meio do filtro de média da vizinhança, o que causa o borramento da imagem e a consequente perda de definição das bordas.	30
Bordas	Executa a identificação de bordas, usando o gradiente da função $f(x, y)$ da imagem em cada ponto (x, y) , o que permite detectar mudanças abruptas entre <i>pixels</i> vizinhos.	31
Mediana	Executa suavização da imagem por meio do filtro de mediana da vizinhança, o que causa o borramento da imagem e a consequente perda de definição das bordas.	32
Equalização	Obtém uma imagem cujo histograma apresenta uma distribuição de <i>pixels</i> o mais uniforme possível entre os níveis de intensidade definidos pela resolução de contraste.	33
Dilatação	Uma operação morfológica que pode aumentar os objetos de uma imagem, conectar objetos separados e preencher “buracos”.	52
Erosão	Uma operação morfológica que pode reduzir os objetos de uma imagem, separar objetos conectados e aumentar “buracos”.	53
Esqueletização	Produz uma imagem afinada (esqueleto) ao remover todos os <i>pixels</i> redundantes, isto é, os <i>pixels</i> que não possuem ao menos dois vizinhos próximos das bordas.	186
<i>Fingerprint</i>	<i>Pipeline</i> das seguintes técnicas para pré-processamento de imagens de impressões digitais: equalização do histograma, <i>thresholding</i> (binarização da imagem), dilatação, esqueletização e erosão (PERENHA, 2003).	353

Fonte: Davi Silva Rodrigues, 2017

A complexidade dos programas foi definida de acordo com a quantidade de linhas de código (ALSHRAIDEH; MAHAFZAH; AL-SHARAEH, 2011; KOLEEJAN; XUE; ZHANG, 2015; YAO; GONG; ZHANG, 2015). Foram consideradas apenas as linhas de código que contêm a lógica principal do programa sob teste, desconsiderando comentários. Tal medida de complexidade também fornece uma indicação do potencial número de mutantes que podem ser gerados.

Tabela 7 – Parâmetros dos programas de PI utilizados nas etapas de desenvolvimento e validação da abordagem TAIGA

Programa	Parâmetro	Valor
Média	Tamanho da vizinhança	3 x 3
Bordas		
Mediana		
Dilatação		
Erosão		
<i>Fingerprint</i>		11 x 11
Esqueletização		
Equalização	Total de níveis de cinza	256
<i>Fingerprint</i>		
Esqueletização	Valor de limiar para a	Média dos <i>pixels</i> da vizinhança de tamanho 11 x 11
<i>Fingerprint</i>	binarização de imagens	

Fonte: Davi Silva Rodrigues, 2017

Todos os programas, com exceção do programa Equalização, utilizam *templates* específicos para realizar operações sobre a vizinhança de um *pixel*. Os programas Média e Mediana, por exemplo, substituem o *pixel* localizado no centro deste *template*, respectivamente, pela média e mediana da vizinhança. Os programas que equalizam o histograma das imagens (Equalização e *Fingerprint*) utilizam o total de níveis de cinza determinado pela resolução de contraste. Os programas Esqueletização e *Fingerprint* utilizam uma máscara 11 x 11 para obter o valor limiar utilizado na binarização de imagens.

Na etapa de validação, cada um dos experimentos envolveu um programa de PI diferente, porém com as mesmas imagens de entrada. Todos os experimentos foram executados 20 vezes para a obtenção de valores mínimos, máximos e médios de cada métrica utilizada. Portanto, a etapa de validação experimental envolveu 32 execuções de teste tradicional (como uma imagem escolhida manualmente pelo testador) de programas de PI e 640 execuções da abordagem proposta.

Os experimentos consistiram na execução de cada programa original e da abordagem TAIGA com cada uma das imagens de teste iniciais. Ao final de cada experimento, foram obtidos um relatório com as métricas de avaliação e as novas imagens geradas pela abordagem. Dessa forma, foi possível verificar a eficácia da abordagem TAIGA ao comparar as métricas obtidas em cada experimento com o teste tradicional e com a abordagem proposta.

Foram considerados diferentes cenários nos experimentos realizados:

- programas de PI menos complexos e mais complexos;
- programas processando imagens adequadas e inadequadas a seu propósito;
- diversidade de imagens de teste simulando a utilização de bancos de imagens e,
- aplicações reais.

Para que esses cenários fossem contemplados na validação, cada experimento (com cada um dos programas) utilizou uma população inicial diferente. As populações iniciais foram constituídas de uma única imagem (repetida cinco vezes para atingir o tamanho de população definido na Seção 5.1.1).

O paralelismo na execução das etapas de desenvolvimento e de validação resultou em dois benefícios: (1) facilidade de testar rapidamente ideias de adaptações de componentes do AG e imediatamente observar nas métricas de avaliação o impacto (positivo ou negativo) das modificações realizadas; e (2) facilidade de confirmar a relevância das métricas de avaliação utilizadas, a partir das inúmeras execuções do AG ao longo do desenvolvimento da abordagem.

4.4 Considerações finais

Os procedimentos metodológicos adotados para o processo de elaboração de uma abordagem de geração de dados de teste para programas de PI foram apresentados neste capítulo. A abordagem resultante será apresentada no Capítulo 5. Os resultados dos experimentos realizados e as métricas obtidas são apresentados detalhadamente no Capítulo 6.

5 TAIGA — *Test imAge generatIon by Genetic Algorithm*

A abordagem TAIGA — *Test imAge generatIon by Genetic Algorithm* — tem como objetivo gerar imagens de teste para programas de PI por meio de algoritmo genético. Fornecida uma imagem de entrada, a abordagem é capaz de produzir uma ou mais novas imagens de teste mais adequadas para revelar defeitos em programas de PI. A abordagem consiste em um arcabouço conceitual, cuja arquitetura é apresentada na Figura 16. Partes da abordagem são constituídas por ferramentas desenvolvidas ou disponibilizadas na literatura, conforme descrito a seguir.

Figura 16 – Arquitetura da abordagem TAIGA

Fonte: Davi Silva Rodrigues, 2017

Nota-se que os programas de PI originais, além das versões mutantes geradas pela ferramenta *muJava*, são fornecidos ao algoritmo genético, que gera novas imagens de teste e métricas de avaliação. Portanto, assume-se que os programas de PI originais estão corretos e que, conseqüentemente, os resultados das execuções desses programas são oráculos.

5.1 Adaptação de algoritmo genético para o domínio de imagens

Algoritmos genéticos codificam, modificam e avaliam soluções candidatas. No contexto de imagens, que são dados complexos, o processo de evolução deve gerar novas imagens válidas. Nos trabalhos encontrados na literatura (Capítulo 3), é comum representar as soluções por meio de estruturas de dados simples, gerar a população inicial de modo aleatório e mesclar estruturas de dados na implementação de operadores genéticos.

Imagens formadas por *pixels* com valores aleatórios representam apenas ruídos. Da mesma forma, uma imagem válida pode ser transformada em ruído, caso seja manipulada por um operador de mutação que substitui *pixels* por valores aleatórios — ainda que os valores aleatórios sejam válidos considerando a resolução de contraste da imagem.

O fluxograma do AG aqui proposto é apresentado na Figura 17.

Figura 17 – Fluxograma do AG adaptado para o domínio de imagens utilizado na abordagem TAIGA

Fonte: Davi Silva Rodrigues, 2017

As imagens fornecidas pelo testador constituem a população inicial (Seção 5.1.2), que é avaliada por uma função *fitness* adaptada (Seção 5.1.5). Em seguida, as condições de parada (Seção 5.1.6) são testadas. Caso as condições de parada tenham sido atingidas, então as imagens geradas pela abordagem são disponibilizadas ao testador, juntamente com um relatório de execução (Seção 5.2). Caso as condições de parada ainda não tenham sido atingidas, então é executado mais um passo de evolução. Os indivíduos que serão

utilizados pelos operadores de *crossover* (Seção 5.1.4) e de mutação (Seção 5.1.3) são selecionados pelo método de seleção *Roulette Wheel* (Seção 2.5.3) e então os operadores são executados. Ao final de um passo de evolução, a nova população é avaliada pela função *fitness* e as condições de parada são testadas novamente.

As subseções seguintes apresentam as adaptações realizadas nos componentes do AG de modo a adequá-lo às particularidades do domínio de imagens, porém ainda realizando operações capazes de produzir, em vez de imagens ruidosas, imagens válidas, isto é, adequadas às funcionalidades dos programas de PI testados.

5.1.1 Parâmetros genéticos

O AG da abordagem TAIGA aceita os seguintes parâmetros genéticos: probabilidade de mutação (p_m), probabilidade de *crossover* (p_c), tamanho da população (n) e elitismo. O operador de seleção utilizado foi o *Roulette Wheel*. A Tabela 8 apresenta os parâmetros genéticos utilizados no desenvolvimento e na validação da abordagem.

Tabela 8 – Parâmetros genéticos utilizados pela abordagem TAIGA

Parâmetro genético	Descrição	Valor
Elitismo	Os n melhores indivíduos da população atual são incluídos na nova população inalterados.	1
<i>Mutation parts</i>	Define o número de divisões a serem realizadas nas imagens pelo operador de mutação.	3
p_c	Probabilidade de ocorrência do operador de <i>crossover</i> .	30%
p_m	Probabilidade de ocorrência do operador de mutação.	40%
n	Tamanho da população	5

Fonte: Davi Silva Rodrigues, 2017

Com o objetivo de evitar que novas gerações possam ter valores da função *fitness* menores do que os das gerações anteriores, o melhor indivíduo de uma geração é automaticamente alocado para a próxima geração sem passar pelos operadores genéticos (elitismo). Por essa razão, o tamanho de população foi definido como cinco com o objetivo de garantir que existam ao menos dois pares de indivíduos disponíveis para as operações de *crossover* e mutação. O parâmetro *mutation parts* é detalhado na Subseção 5.1.3.

Os parâmetros genéticos foram definidos de modo empírico ao longo do desenvolvimento e utilizados em todos os experimentos realizados. Foram realizados testes com

diferentes valores para os parâmetros. Os valores apresentados na Tabela 8 são aqueles que produziram os melhores resultados.

As probabilidades de *crossover* e de mutação, ao contrário dos trabalhos encontrados na literatura, foram definidas de modo a priorizar a mutação (Subseção 2.5.3). Probabilidades maiores do que 30% para o *crossover* causam piora no *mutation score* obtido. No caso do operador de mutação, probabilidades de até 40% aumentam o *mutation score*. Para probabilidades maiores, pouca ou nenhuma diferença foi observada durante os testes realizados. É possível que as adaptações realizadas nesses operadores genéticos sejam as responsáveis por essa “inversão de probabilidades”.

5.1.2 Codificação das soluções candidatas

Para que o AG seja capaz de processar e de gerar imagens, é necessário que a codificação utilizada garanta que os *pixels* componham uma imagem válida e que os operadores de mutação e de *crossover* não transformem as imagens em ruído. Dessa forma, as imagens processadas pelos operadores continuam apresentando regiões que tenham significado.

A abordagem TAIGA representa imagens como instâncias da classe **Image** (Figura 18). Esta classe foi implementada com o objetivo de ler e armazenar as imagens e suas características (espaço de cores, valores dos *pixels*, altura, largura, etc.), efetuar comparações *pixel a pixel*, criar cópias, salvá-las na memória secundária, convertê-las para outros espaços de cores, etc.

A classe **Image** foi desenvolvida com o objetivo de centralizar todos os atributos e operações relacionados às imagens processadas e geradas pela abordagem TAIGA. Um atributo do tipo **BufferedImage** (classe nativa da linguagem Java) é o responsável por armazenar as imagens (e suas características) e por efetuar a conversão de espaço de cores (quando necessária).

Outras operações foram desenvolvidas para atender a necessidades específicas. A função *fitness* utiliza o método `clone` para criar cópias idênticas da imagem candidata e testá-la com as versões mutantes do programa sob teste. O programa Esqueletização utiliza os métodos `toBinaryMatrix` e `toImage` para, respectivamente, converter uma imagem para uma matriz binária (que representa os *pixels* apenas com os valores 0 e 1) e para

Figura 18 – Classe **Image**: codificação das imagens processadas pelo AG

Fonte: Davi Silva Rodrigues, 2017

converter uma matriz binária para uma imagem correspondente. Esses métodos específicos destacam a relação próxima que há entre as adaptações dos componentes do AG e as adaptações relacionadas ao problema sob análise.

No contexto de AGs, os objetos da classe **Image** são correspondentes aos cromossomos e cada *pixel* (ou cada elemento da matriz/vetor) corresponde a um gene (Seção 2.5.2).

As funcionalidades implementadas nos métodos da classe **Image** são apresentadas a seguir.

- **Leitura e armazenamento de imagens e de suas características.**

O atributo do tipo **BufferedImage** é responsável por ler e armazenar em memória as imagens processadas. Este atributo possibilita à abordagem TAIGA processar imagens ou como conjuntos de *pixels*, ou como matrizes ou como vetores de números inteiros. As matrizes e os vetores são utilizados pelos operadores genéticos apresentados nas subseções seguintes. Independentemente da forma de representação da imagem, os valores de *pixel* informados são automaticamente validados pelos

métodos da classe `BufferedImage`.

- **Criação de novas imagens.**

O mesmo atributo permite a criação de uma imagem “vazia”, que pode ser “preenchida” posteriormente com a garantia de que seus *pixels* são válidos.

- **Criação de cópias.**

Foi implementado um método específico para a criação de “clones” dos objetos da classe `Image` com o objetivo de viabilizar a sua manipulação sem que se perca o objeto original. Este método é utilizado na função *fitness*.

- **Gravação de imagens na memória secundária.**

Ao final dos experimentos, este método é utilizado para gravar as novas imagens geradas e compor os resultados da abordagem TAIGA.

- **Binarização de imagens (*thresholding*).**

Um método foi implementado a fim de converter todos os *pixels* de uma imagem para brancos ou pretos, tornando-a binarizada. Desse modo, a abordagem facilita o teste de programas de PI que exigem imagens de entrada binarizadas.

- **Comparação de imagens.**

Duas imagens são consideradas iguais se, e somente se, todos os valores de seus vetores correspondentes forem iguais. Esta comparação é utilizada pela função *fitness*.

- **Conversão de espaço de cores.**

Caso sejam testados programas de PI que exijam imagens de entrada em um espaço de cores específico (RGB ou níveis de cinza, por exemplo), há um método específico para realizar as conversões necessárias.

A classe `Image` permite à abordagem TAIGA armazenar imagens e suas características em sua totalidade e, ao mesmo tempo, oferece a flexibilidade necessária aos operadores genéticos para que mutações e *crossover* ocorram sem reduzir as imagens a ruído.

5.1.3 Operador de mutação

A mutação de imagens, caso ocorra por meio da atribuição de valores aleatórios aos *pixels*, pode gerar ruído em vez de uma imagem modificada. Para evitar esse problema, foi definido um operador de mutação específico para imagens que limita o escopo e o fator aleatório do processo.

Em vez de alterar *pixels* aleatoriamente, o operador de mutação seleciona uma parte aleatória da imagem e troca os valores atuais dos *pixels* por valores inversos. A Figura 19 apresenta um exemplo de mutação de uma imagem em preto e branco que teve os *pixels* de um quadrante escolhido aleatoriamente invertidos para branco ou preto.

Figura 19 – Operador de mutação da abordagem TAIGA, com parâmetro genético *mutation parts* = 2: (a) imagem original, (b) imagem modificada

Fonte: Davi Silva Rodrigues, 2017

Inicialmente, a probabilidade de mutação p_m é utilizada a fim de se definir a ocorrência ou não da mutação de uma imagem. Caso a mutação ocorra, a quantidade de partes em que a imagem é dividida é definida pelo parâmetro genético *mutation parts*, que foi apresentado na Subseção 5.1.1. Então, uma das partes resultantes é selecionada aleatoriamente e tem seus *pixels* invertidos (Figura 19b).

O parâmetro *mutation parts* representa o número de partes em que cada dimensão da imagem será dividida, conforme indica o Algoritmo 1.

A manipulação de partes das imagens é realizada a partir dos métodos `getRGB` e `setRGB` da classe `BufferedImage`. O primeiro método retorna um vetor correspondente ao trecho da imagem delimitado pelos parâmetros informados. O segundo método executa o processo inverso ao alterar os *pixels* delimitados pelos parâmetros com os valores de um vetor correspondente.

Desse modo, o operador de mutação é capaz de introduzir modificações em uma parte aleatória da imagem, porém sem gerar ruídos como resultado.

Algoritmo 1 Algoritmo para a mutação de uma imagem *img* na abordagem TAIGA

```

1: partsWidth  $\leftarrow$  img.getWidth() / mutationParts;
2: partsHeight  $\leftarrow$  img.getHeight() / mutationParts;
3:
4: randomNum  $\leftarrow$  getRandomNumber(0, 1);
5: startWidth  $\leftarrow$  randomNum * partsWidth;
6: startHeight  $\leftarrow$  randomNum * partsHeight;
7:
8: if startWidth  $\geq$  img.getWidth() then
9:   startWidth  $\leftarrow$  img.getWidth() - partsWidth;
10:
11: if startHeight  $\geq$  img.getHeight() then
12:   startHeight  $\leftarrow$  img.getHeight() - partsHeight;
13:
14: array  $\leftarrow$  img.getArray(startWidth, startHeight, partsWidth, partsHeight);
15:
16: for i = 0 to array.length - 1 do
17:   if array[i] = 255 then
18:     array[i]  $\leftarrow$  0;
19:   else
20:     array[i]  $\leftarrow$  255;
21:
22: img.setArray(startWidth, startHeight, partsWidth, partsHeight, array);

```

Fonte: Davi Silva Rodrigues, 2017

5.1.4 Operadores de *crossover*

O operador genético de *crossover* promove a recombinação de genes entre indivíduos. Geralmente, são utilizados um ou mais pontos de *crossover* aleatórios. A partir desses pontos, os genes à esquerda e à direita são trocados entre os indivíduos. No domínio de imagens, o mesmo princípio pode ser aplicado.

Em vez de definir pontos aleatórios de *crossover*, a abordagem TAIGA possui dois operadores distintos que podem ser utilizados conjuntamente: *crossover* horizontal e *crossover* vertical. Ambos os operadores respeitam a probabilidade de *crossover* p_c .

Os operadores de *crossover* da abordagem TAIGA dividem as imagens em duas partes horizontalmente (Figura 20) ou verticalmente (Figura 21), a partir do centro. Em seguida, a parte superior/esquerda da primeira imagem (Figura 20a/21a) passa a ser a parte superior/esquerda da imagem resultante (Figura 20c/21c); analogamente, a parte inferior/direita da segunda imagem (Figura 20b/21b) passa a ser a parte inferior/direita da imagem resultante (Figura 20c/21c). Desse modo, evita-se que duas imagens troquem

entre si trechos que representem apenas o fundo de cada imagem — o que anularia os efeitos do operador genético.

Figura 20 – Operador de *crossover* horizontal da abordagem TAIGA: (a) imagem pai 1, (b) imagem pai 2, (c) imagem resultante

Fonte: Davi Silva Rodrigues, 2017

Figura 21 – Operador de *crossover* vertical da abordagem TAIGA: (a) imagem pai 1, (b) imagem pai 2, (c) imagem resultante

Fonte: Davi Silva Rodrigues, 2017

Os resultados dos operadores de mutação e de *crossover* combinam-se ao longo da execução do AG e dão origem a novas imagens.

5.1.5 Função *fitness*

A função *fitness* de um AG deve avaliar a aptidão das soluções candidatas para resolver o problema de interesse. No contexto da abordagem TAIGA, o problema a ser resolvido é revelar os defeitos de qualquer programa de PI. Por essa razão, a função *fitness* deve ser genérica, independente do programa sob teste. No entanto, não há como determinar a aptidão de uma imagem para revelar os defeitos de um programa sem que as características deste não sejam consideradas. O teste de mutação, entretanto, atende a esses dois requisitos.

Utilizando o teste de mutação como função *fitness*, o AG da abordagem TAIGA é capaz de determinar a aptidão de uma imagem para revelar os defeitos de qualquer

programa de PI. Os resultados do teste de mutação (*mutation score*) são utilizados como valor de *fitness*.

A Figura 22 apresenta um fluxograma da avaliação das imagens de teste processadas pelo algoritmo genético.

Figura 22 – Fluxograma da função *fitness* utilizada na abordagem TAIGA

Fonte: Davi Silva Rodrigues, 2017

Para que a função *fitness* possa ser executada, é necessário que os programas de PI sob teste sejam adaptados e que as versões mutantes já tenham sido geradas.

A ferramenta *muJava*, utilizada para gerar programas mutantes (Apêndice B), processa classes separadamente. Adicionalmente, as versões mutantes são criadas a partir dos métodos existentes na classe sob teste. Por esse motivo, os programas de PI foram implementados cada um em uma classe Java. Foram definidos métodos auxiliares que encapsularam toda a lógica não essencial (como, por exemplo, gravação de *logs* e inicialização de objetos auxiliares) e métodos que encapsularam a lógica principal de cada programa. Desse modo, foi possível obter programas com mutações mais significativas e com maiores chances de apresentar falhas na execução.

Após a geração de mutantes, é necessário garantir que cada um dos programas esteja preparado para encerrar sua execução em caso de *loops* infinitos. Tais *loops* podem ocorrer devido às mutações sofridas pelos programas (os limites de um *loop* podem ser excluídos ou substituídos por outros que sejam inatingíveis, por exemplo). Também é possível que mutantes entrem em *loop* com determinadas imagens de teste e executem normalmente com as demais imagens de teste.

Este problema foi resolvido por meio do uso de *threads* exclusivas para cada mutante. A função *fitness* executa cada um dos mutantes e determina um *timeout* de 1500 milissegundos, definido empiricamente, para que uma imagem resultante seja retornada. Consequentemente, mutantes que demoram mais do que 1,5 segundo para retornar uma imagem resultante são considerados mortos.

O *timeout* foi definido a partir de uma série de experimentos com os oito programas de PI utilizados nas etapas de desenvolvimento e de validação. Inicialmente, o *timeout* foi definido como 1 segundo. Em seguida, a função *fitness* foi executada com as imagens que geravam o maior tempo de execução em cada programa. Em cada execução, o *timeout* recebeu pequenos incrementos e os valores de *mutation score* foram armazenados. A partir de 1,5 segundo, o *mutation score* retornado pela função *fitness* não aumentou.

O principal critério utilizado na definição de mutantes vivos e mortos é a comparação entre uma imagem oráculo e a imagem resultante retornada pelo programa mutante.

A imagem oráculo é obtida por meio da execução do programa de PI original. Como entrada, o programa sob teste recebe a imagem que será avaliada pela função *fitness*. A partir da comparação *pixel a pixel* implementada na classe **Image** (Seção 5.1.2), é possível determinar se existem diferenças entre as imagens produzidas pelo programa mutante e pelo programa original. Caso existam diferenças, o mutante em questão é considerado morto. Caso contrário, o mutante é considerado vivo.

Em resumo, a função *fitness* considera morto o mutante que:

- produz uma imagem resultante diferente da imagem oráculo, ou
- gera uma exceção em tempo de execução (falha), ou
- demora mais do que 1,5 segundo para retornar uma imagem resultante.

Por consequência, os mutantes que não atendem aos requisitos listados acima são considerados vivos.

Após o teste de todos os programas mutantes, é possível calcular o *mutation score* da imagem avaliada pela função *fitness* (Equação 5), em que K é o número de mutantes mortos pela imagem e T é o número total de mutantes.

$$\text{Valor de fitness} = \frac{K}{T} \quad (5)$$

Os mutantes equivalentes não são considerados no cálculo do valor de *fitness*. Antes da execução da abordagem, estes mutantes são identificados de modo manual e excluídos do conjunto de mutantes do programa correspondente.

A identificação de mutantes equivalentes é um processo manual. Nos testes iniciais de um programa de PI, todos os mutantes vivos são verificados manualmente. Os programas cujas mutações não interferem de modo algum na imagem resultante — independentemente da imagem de teste utilizada — são considerados funcionalmente equivalentes ao programa original.

5.1.6 Condições de parada

O AG da abordagem TAIGA utiliza duas condições de parada: valor de *fitness* e estagnação.

- **Valor de *fitness*.**

Caso o valor de *fitness* do melhor indivíduo da população atual seja 100%, a execução do AG é encerrada.

- **Estagnação.**

Caso o melhor valor de *fitness* obtido ao longo da execução não aumente por três gerações seguidas, a execução do AG é encerrada.

As duas condições de parada são testadas a cada iteração do AG. O uso de uma meta de valor de *fitness* evita a execução desnecessária do AG após a obtenção do valor máximo de *fitness*. A verificação de estagnação limita o número de gerações e, conseqüentemente, o tempo de execução do AG. Ambas as condições de parada estão intimamente ligadas à função *fitness* apresentada na Seção 5.1.5.

5.2 Imagens de teste geradas e métricas de avaliação obtidas

A abordagem TAIGA permite gerar uma quantidade customizável de dados de teste para os programas de PI de interesse. Podem ser fornecidas como população inicial uma ou mais imagens. A execução da abordagem envolve os passos a seguir.

1. Escolha do programa de PI a ser testado.
2. Escolha de uma ou mais imagens iniciais.
3. Adequação do programa de PI original à ferramenta *muJava*.
4. Geração de programas mutantes a partir da ferramenta *muJava*.
5. Adequação dos programas mutantes para tratamento de *loops* infinitos e execução em *threads* exclusivas.
6. Execução do algoritmo genético.
7. Obtenção de relatório de resultados e gravação, na memória secundária, das novas imagens de teste geradas.

Ao final da execução da abordagem TAIGA, são geradas novas imagens de teste (Figura 23) e relatórios (Figura 24).

Figura 23 – Exemplos de imagens de teste geradas pela abordagem TAIGA para os programas (a) Bordas, (b) Mediana, (c) *Fingerprint* e (d) Esqueletização; a definição dos programas foi apresentada na Tabela 6

Fonte: Davi Silva Rodrigues, 2017

Figura 24 – Exemplo de relatório gerado pela abordagem TAIGA após a geração de imagens de teste para o programa Dilatação

```

SUT: Dilatacao.java
Algoritmo Genético - Imagem em preto e branco - População: 5

Generation 0 (best fitness): 61.386138613861384%
Foram necessárias 5 imagens para matar 61.3861386138613% dos mutantes.
Num. de Mutantes: 303
=====
Generation 1 (best fitness): 62.37623762376238%
Foram necessárias 5 imagens para matar 62.37623762376238% dos mutantes.
Num. de Mutantes: 303
=====
Generation 2 (best fitness): 100.0%
Foram necessárias 4 imagens para matar 100.0% dos mutantes.
Num. de Mutantes: 303
=====

População final

1 | Mutation score (fitness): 100.0%
2 | Mutation score (fitness): 62.37623762376238%
3 | Mutation score (fitness): 61.386138613861384%
4 | Mutation score (fitness): 61.386138613861384%
5 | Mutation score (fitness): 61.386138613861384%

=====

2 generations
Minimum candidates: 4.0
Maximum candidates: 5.0
Average candidates: 4.5
Minimum mutation score: 62.37623762376238%
Maximum mutation score: 100.0%
Average mutation score: 81.1881188118812%

=====

Elapsed time: 5 min 35 s

```

Fonte: Davi Silva Rodrigues, 2017

Os relatórios apresentam os resultados parciais do AG, a população final com *mutation score*, número de imagens necessárias para chegar a 100% de MS e o tempo total decorrido. Caso os experimentos sejam executados mais de uma vez, os tempos mínimo, máximo e médio de execução também são apresentados.

Conforme exposto na Subseção 5.1.1, a abordagem TAIGA pode gerar uma quantidade customizável de novas imagens de teste. Os experimentos conduzidos utilizaram uma

população de tamanho 5 devido às características inerentes ao operador de *crossover*, que processa pares de imagens. No entanto, o testador pode configurar a abordagem de modo a gerar menos imagens ou mais imagens de teste.

A condição de parada do AG determina que, caso seja gerada uma imagem com *mutation score* de 100%, a execução termine. Por esse motivo, é possível que nem todas as imagens da população tenham o mesmo *score*. Porém, sempre que possível, será gerada ao menos uma nova imagem que seja melhor do que as imagens iniciais fornecidas.

5.3 Considerações finais

A abordagem TAIGA proposta por esta pesquisa foi apresentada neste capítulo. Foram detalhadas as adaptações realizadas no AG, assim como as ferramentas disponibilizadas na literatura que foram utilizadas. A execução dos experimentos e as métricas de avaliação obtidas são apresentadas no Capítulo 6.

6 Resultados e discussão

A abordagem proposta foi validada por meio de experimentos com os programas de PI e com as imagens de teste iniciais definidos no Capítulo 4. A Seção 6.1 apresenta os resultados dos experimentos realizados e uma discussão desses resultados é feita na Seção 6.2.

6.1 Experimentos realizados com programas de PI

Para cada um dos oito programas de PI apresentados na Tabela 6, foram executados o teste tradicional — no qual o programa de PI é executado com cada uma das imagens iniciais — e a abordagem TAIGA com cada uma das quatro imagens de teste iniciais (Seção 4.3), totalizando oito experimentos para cada programa de PI. Cada experimento com a abordagem TAIGA foi repetido 20 vezes e os experimentos com teste tradicional foram realizados apenas uma vez para cada imagem inicial. Assim, foram realizados 80 experimentos com a abordagem TAIGA e 4 testes tradicionais para cada programa de PI (Tabela 9). Ao final da execução de cada programa, foram obtidos os valores mínimos, máximos e médios das métricas de avaliação observadas (*mutation score*, tempo de execução e número de gerações).

Tabela 9 – Quantidade de experimentos e de execuções realizados na etapa de validação da abordagem TAIGA

Teste	Imagens	Programas	Experimentos	Repetições	Execuções
Tradicional	4	8	32	1	32
TAIGA			32	20	640

Fonte: Davi Silva Rodrigues, 2017

As 672 execuções foram realizadas em um computador com processador Intel Core i5 4200U e 4GB de RAM. Assim como a própria abordagem TAIGA, os experimentos, os programas de PI e as suas versões mutantes foram implementados em Java. Foi utilizado o ambiente de desenvolvimento Eclipse 4.6.3 e a versão 8 da linguagem Java.

Os mutantes equivalentes foram manualmente identificados e removidos, como descrito na Seção 5.1.5.

Em alguns experimentos, foi executada uma seleção de mutantes a fim de tornar o tempo de execução viável. Essa redução na quantidade de mutantes foi necessária devido

ao custo de processamento inerente ao AG, à complexidade de alguns programas de PI e à quantidade de execuções realizadas. Sem a redução do número de mutantes, algumas execuções precisaram de mais de 60 minutos apenas para executar a função *fitness* com a população inicial. Os experimentos com quantidade reduzida de mutantes são indicados nas subseções 6.1.1 a 6.1.8.

A seleção dos programas mutantes utilizados pela função *fitness* considerou, além da quantidade de mutantes, quais trechos do programa original foram modificados por cada mutante. A ferramenta *muJava* nomeia os mutantes gerados com um sigla referente ao tipo de mutação (de acordo com a Tabela 5) e um número sequencial: números de sequência menores indicam defeitos presentes no início do programa original; e números maiores indicam a presença de defeitos mais próximos ao final do programa original. De posse desses dados, é possível reduzir a quantidade de mutantes e, ao mesmo tempo, aumentar a probabilidade de selecionar mutantes que modificaram trechos diferentes do código-fonte do programa original.

Sendo assim, quando necessário, foram selecionados os 10 primeiros, os 10 últimos e os 10 mutantes mais centrais de cada tipo de mutação (de acordo com seus números sequenciais). Se a ferramenta *muJava* gerar 50 mutantes do tipo AOR (e o total de mutantes for impeditivo para a realização dos experimentos), por exemplo, são selecionados apenas os mutantes de AOR1 a AOR10, de AOR21 a AOR30 e de AOR41 a AOR50. Quando há menos de 30 mutantes de um mesmo tipo, todos são selecionados. Esse processo é repetido para cada tipo de mutação (AOR, ROR, COD, SOR, SDL, VDL, etc.). Dessa forma, é possível descartar programas mutantes e tornar o tempo de execução do AG factível, sem comprometer a qualidade das imagens de teste geradas.

As subseções 6.1.1 a 6.1.8 apresentam tabelas com a consolidação das métricas obtidas a partir das execuções realizadas com cada programa de PI. As tabelas apresentam os valores mínimos, máximos e médios das métricas obtidas a partir de 20 repetições dos experimentos correspondentes com a abordagem TAIGA (um para cada imagem inicial). Os valores de MS indicam as imagens geradas mais capazes e menos capazes de revelar defeitos ao longo da execução do AG. O tempo médio refere-se à média de tempo despendido para finalizar a execução do experimento (gerar uma imagem de teste nova). A diferença percentual (Δ) refere-se ao tempo de execução adicional demandado pela abordagem TAIGA em relação ao teste tradicional (cujas métricas foram obtidas a partir de uma única execução do experimento para cada imagem inicial). A Equação 6 apresenta

o cálculo dessa diferença, em que t_{TAIGA} é o tempo médio despendido pela abordagem TAIGA e $t_{tradicional}$ é o tempo despendido pelo teste tradicional.

$$\Delta = \frac{(t_{TAIGA} - t_{tradicional})}{t_{tradicional}} \cdot 100 \quad (6)$$

6.1.1 Programa Média

Dos 370 mutantes do programa Média, 22 são equivalentes. As Tabelas 10 e 11 apresentam os resultados dos experimentos realizados com os 348 mutantes restantes.

Tabela 10 – Resultados dos experimentos realizados com 348 mutantes do programa Média: *mutation score*

Dado de teste	Abordagem	<i>Mutation score</i> (%)			
		mín.	máx.	médio	ganho
Imagem preto e branco	TAIGA	98,85	100	99,55	1,15
	Tradicional	98,85			
Imagem nível de cinza	TAIGA	100	100	100	0
	Tradicional	100			
Imagem fundo preto	TAIGA	38,51	100	96,60	61,49
	Tradicional	38,51			
Impressão digital	TAIGA	99,43	100	99,79	0
	Tradicional	100			

Fonte: Davi Silva Rodrigues, 2017

Tabela 11 – Resultados dos experimentos realizados com 348 mutantes do programa Média: tempos de execução e diferença percentual (Δ)

Dado de teste	Tempo de execução (s)				Δ (%)
	Tradicional	mín.	TAIGA máx.	médio	
Imagem preto e branco	13	37	130	82	531
Imagem nível de cinza	10	18	18	18	80
Imagem fundo preto	9	37	149	93	933
Impressão digital	15	42	84	60	300

Fonte: Davi Silva Rodrigues, 2017

6.1.2 Programa Bordas

O programa Bordas deu origem a 183 mutantes. Desse total, 19 são mutantes equivalentes. Os experimentos foram, então, realizados com 164 mutantes. As Tabelas 12 e 13 apresentam os resultados obtidos.

Tabela 12 – Resultados dos experimentos realizados com 164 mutantes do programa Bordas: *mutation score*

Dado de teste	Abordagem	<i>Mutation score</i> (%)			
		mín.	máx.	médio	ganho
Imagem preto e branco	TAIGA	100	100	100	0
	Tradicional	100			
Imagem nível de cinza	TAIGA	100	100	100	0
	Tradicional	100			
Imagem fundo preto	TAIGA	60,98	100	96,88	39,02
	Tradicional	60,98			
Impressão digital	TAIGA	100	100	100	0
	Tradicional	100			

Fonte: Davi Silva Rodrigues, 2017

Tabela 13 – Resultados dos experimentos realizados com 164 mutantes do programa Bordas: tempos de execução e diferença percentual (Δ)

Dado de teste	Tempo de execução (s)				Δ (%)
	Tradicional	mín.	TAIGA máx.	médio	
Imagem preto e branco	5	9	12	9	80
Imagem nível de cinza	4	9	9	9	125
Imagem fundo preto	4	18	37	23	475
Impressão digital	5	10	11	10	100

Fonte: Davi Silva Rodrigues, 2017

6.1.3 Programa Mediana

O programa Mediana teve 320 mutantes gerados, dos quais 22 são equivalentes. As Tabelas 14 e 15 apresentam os resultados dos experimentos realizados com esse programa.

Tabela 14 – Resultados dos experimentos realizados com 298 mutantes do programa Mediana: *mutation score*

Dado de teste	Abordagem	<i>Mutation score</i> (%)			
		mín.	máx.	médio	ganho
Imagem preto e branco	TAIGA	80,87	100	98,59	19,13
	Tradicional	80,87			
Imagem nível de cinza	TAIGA	100	100	100	0
	Tradicional	100			
Imagem fundo preto	TAIGA	40,60	100	95,47	59,40
	Tradicional	40,60			
Impressão digital	TAIGA	99,33	99,33	99,33	0
	Tradicional	99,33			

Fonte: Davi Silva Rodrigues, 2017

Tabela 15 – Resultados dos experimentos realizados com 298 mutantes do programa Mediana: tempos de execução e diferença percentual (Δ)

Dado de teste	Tradicional	Tempo de execução (s)			Δ (%)
		mín.	máx.	médio	
Imagem preto e branco	10	37	131	78	680
Imagem nível de cinza	10	18	19	19	90
Imagem fundo preto	10	56	169	106	960
Impressão digital	14	90	91	90	543

Fonte: Davi Silva Rodrigues, 2017

6.1.4 Programa Equalização

A ferramenta *muJava* gerou 168 versões mutantes do programa Equalização. Desses mutantes, 14 são equivalentes. Os resultados dos experimentos realizados com os 154 programas mutantes restantes são apresentados pelas Tabelas 16 e 17.

Tabela 16 – Resultados dos experimentos realizados com 154 mutantes do programa Equalização: *mutation score*

Dado de teste	Abordagem	<i>Mutation score</i> (%)			
		mín.	máx.	médio	ganho
Imagem preto e branco	TAIGA	100	100	100	0
	Tradicional		100		
Imagem nível de cinza	TAIGA	100	100	100	0
	Tradicional		100		
Imagem fundo preto	TAIGA	93,51	100	99,07	6,49
	Tradicional		93,51		
Impressão digital	TAIGA	100	100	100	0
	Tradicional		100		

Fonte: Davi Silva Rodrigues, 2017

Tabela 17 – Resultados dos experimentos realizados com 154 mutantes do programa Equalização: tempos de execução e diferença percentual (Δ)

Dado de teste	Tradicional	Tempo de execução (s)			Δ (%)
		mín.	máx.	médio	
Imagem preto e branco	12	21	30	21	75
Imagem nível de cinza	10	21	21	21	110
Imagem fundo preto	10	63	127	88	780
Impressão digital	11	22	24	22	100

Fonte: Davi Silva Rodrigues, 2017

6.1.5 Programa Dilatação

Foram gerados 328 mutantes do programa Dilatação. Desses mutantes, 25 foram descartados por serem equivalentes. Os resultados dos experimentos realizados com os 303 mutantes restantes são apresentados nas Tabelas 18 e 19.

Tabela 18 – Resultados dos experimentos realizados com 303 mutantes do programa Dilatação: *mutation score*

Dado de teste	Abordagem	<i>Mutation score</i> (%)			
		mín.	máx.	médio	ganho
Imagem preto e branco	TAIGA	61,39	100	86,98	38,61
	Tradicional	61,39			
Imagem nível de cinza	TAIGA	99,67	100	99,80	0,33
	Tradicional	99,67			
Imagem fundo preto	TAIGA	40,92	62,38	60,77	21,46
	Tradicional	40,92			
Impressão digital	TAIGA	98,02	100	99,64	1,98
	Tradicional	98,02			

Fonte: Davi Silva Rodrigues, 2017

Tabela 19 – Resultados dos experimentos realizados com 303 mutantes do programa Dilatação: tempos de execução e diferença percentual (Δ)

Dado de teste	Tradicional	Tempo de execução (s)			Δ (%)
		mín.	TAIGA máx.	médio	
Imagem preto e branco	59	211	960	369	525
Imagem nível de cinza	62	246	500	408	558
Imagem fundo preto	44	474	1077	787	1689
Impressão digital	78	297	731	404	418

Fonte: Davi Silva Rodrigues, 2017

6.1.6 Programa Erosão

O programa Erosão teve 393 mutantes gerados, dos quais 24 são equivalentes. As Tabelas 20 e 21 apresentam os resultados dos experimentos realizados com os 369 mutantes restantes.

Tabela 20 – Resultados dos experimentos realizados com 369 mutantes do programa Erosão: *mutation score*

Dado de teste	Abordagem	<i>Mutation score</i> (%)			
		mín.	máx.	médio	ganho
Imagem preto e branco	TAIGA	98,92	100	99,92	1,08
	Tradicional	98,92			
Imagem nível de cinza	TAIGA	100	100	100	0
	Tradicional	100			
Imagem fundo preto	TAIGA	82,38	100	95,40	17,62
	Tradicional	82,38			
Impressão digital	TAIGA	96,48	100	98,56	3,52
	Tradicional	96,48			

Fonte: Davi Silva Rodrigues, 2017

Tabela 21 – Resultados dos experimentos realizados com 369 mutantes do programa Erosão: tempos de execução e diferença percentual (Δ)

Dado de teste	Tempo de execução (s)				Δ (%)
	Tradicional	mín.	TAIGA máx.	médio	
Imagem preto e branco	23	73	110	79	243
Imagem nível de cinza	18	36	36	36	100
Imagem fundo preto	18	183	588	297	1550
Impressão digital	30	109	305	205	583

Fonte: Davi Silva Rodrigues, 2017

6.1.7 Programa Esqueletização

As Tabelas 22 e 23 apresentam os resultados obtidos a partir dos experimentos realizados com o programa Esqueletização. Ao todo foram gerados 1613 mutantes, dos quais 1294 foram descartados para reduzir o tempo de execução do AG. Dos 319 mutantes restantes, 9 são equivalentes e 310 foram efetivamente utilizados nos experimentos.

Tabela 22 – Resultados dos experimentos realizados com 310 mutantes do programa Esqueletização: *mutation score*

Dado de teste	Abordagem	<i>Mutation score</i> (%)			
		mín.	máx.	médio	ganho
Imagem preto e branco	TAIGA	90,97	100	97,48	9,03
	Tradicional	90,97			
Imagem nível de cinza	TAIGA	99,35	100	99,57	0,65
	Tradicional	99,35			
Imagem fundo preto	TAIGA	13,87	100	84,75	86,13
	Tradicional	13,87			
Impressão digital	TAIGA	100	100	100	0
	Tradicional	100			

Fonte: Davi Silva Rodrigues, 2017

Tabela 23 – Resultados dos experimentos realizados com 310 mutantes do programa Esqueletização: tempos de execução e diferença percentual (Δ)

Dado de teste	Tempo de execução (s)				Δ (%)
	Tradicional	mín.	TAIGA máx.	médio	
Imagem preto e branco	29	129	306	196	576
Imagem nível de cinza	37	133	269	208	462
Imagem fundo preto	6	147	409	266	4333
Impressão digital	59	122	124	123	108

Fonte: Davi Silva Rodrigues, 2017

6.1.8 Programa *Fingerprint*

O programa *Fingerprint* é uma aplicação real utilizada na identificação de impressões digitais. Foram gerados 3030 mutantes, dos quais 2652 foram descartados para reduzir o tempo de execução do AG. Dos 378 mutantes restantes, 18 são equivalentes. As Tabelas 24 e 25 apresentam os resultados dos experimentos realizados com 360 mutantes do *pipeline* de técnicas.

Tabela 24 – Resultados dos experimentos realizados com 360 mutantes do programa *Fingerprint: mutation score*

Dado de teste	Abordagem	<i>Mutation score</i> (%)			
		mín.	máx.	médio	ganho
Imagem preto e branco	TAIGA	63,06	99,17	96,69	36,11
	Tradicional	63,06			
Imagem nível de cinza	TAIGA	100	100	100	0
	Tradicional	100			
Imagem fundo preto	TAIGA	39,17	95,83	88,07	56,66
	Tradicional	39,17			
Impressão digital	TAIGA	98,89	100	99,75	1,11
	Tradicional	98,89			

Fonte: Davi Silva Rodrigues, 2017

Tabela 25 – Resultados dos experimentos realizados com 360 mutantes do programa *Fingerprint: tempos de execução e diferença percentual (Δ)*

Dado de teste	Tempo de execução (s)				Δ (%)
	Tradicional	mín.	TAIGA máx.	médio	
Imagem preto e branco	76	191	725	351	361
Imagem nível de cinza	52	35	36	35	-33
Imagem fundo preto	17	193	583	360	2018
Impressão digital	73	191	524	291	299

Fonte: Davi Silva Rodrigues, 2017

6.2 Discussão dos resultados

Os resultados apresentados na seção anterior serão discutidos nas subseções seguintes: os valores de *mutation score* são analisados na Seção 6.2.1, o tempo de execução da abordagem TAIGA é abordado na Seção 6.2.2 e a quantidade de gerações do AG utilizado é analisada na Seção 6.2.3.

6.2.1 Análise do *Mutation Score*

Os resultados dos experimentos realizados indicam que as imagens de teste geradas pela abordagem TAIGA, na maior parte das vezes, foram capazes de revelar mais defeitos nos programas de PI testados do que as imagens iniciais fornecidas.

Para cada um dos programas utilizados nos experimentos, a abordagem gerou imagens de teste mais sensíveis aos defeitos que obtiveram os ganhos máximos (em termos de *mutation score*) apresentados pela Tabela 26. São apresentados os valores de *mutation score* obtidos ao se executar o programa de PI com a imagem inicial (coluna “Tradicional”), ao se executar a abordagem TAIGA (coluna “TAIGA”) e o ganho obtido pela abordagem proposta. Quanto maior for o ganho de *mutation score* (MS), maior o número de defeitos revelados pela abordagem proposta em comparação ao número de defeitos revelados pelo teste tradicional.

Tabela 26 – Maiores ganhos de *mutation score* das imagens de teste geradas pela abordagem TAIGA

Programa de PI	Imagem inicial	<i>Mutation score</i> (%)		
		Tradicional	TAIGA	Ganho
Média	Fundo preto	38,51	100	61,49
Bordas	Fundo preto	60,98	100	39,02
Mediana	Fundo preto	40,60	100	59,40
Equalização	Fundo preto	93,51	100	6,49
Dilatação	Preto e branco	61,39	100	38,61
Erosão	Fundo preto	82,38	100	17,62
Esqueletização	Fundo preto	13,87	100	86,13
<i>Fingerprint</i>	Fundo preto	39,17	95,83	56,66

Fonte: Davi Silva Rodrigues, 2017

Conforme apresentado na Tabela 26, as imagens geradas pela abordagem TAIGA (a partir da imagem de fundo preto ou da imagem preto e branco) obtiveram ganhos com

todos os programas utilizados. Dos oito programas testados, sete apresentaram os maiores ganhos ao serem testados com imagens geradas a partir da imagem que possui fundo preto. Provavelmente esta imagem não seria selecionada manualmente por um testador em uma abordagem tradicional. No entanto, ela se mostrou a mais adequada para matar os mutantes, visto que foi a partir dela que o AG foi capaz de gerar as imagens de teste que resultaram nos maiores ganhos de MS obtidos pela abordagem proposta.

O programa *Fingerprint* foi o único a não atingir o MS máximo ao utilizar a imagem inicial que gera os maiores ganhos (imagem de fundo preto). Com esse programa, ao se utilizar imagens iniciais em nível de cinza, é possível revelar todos os defeitos dos mutantes. No entanto, o ganho de MS é menor, visto que o teste tradicional obtém MS alto com imagens em nível de cinza. As outras duas imagens (fundo preto e imagem preto e branco) não são capazes de revelar todos os defeitos, mas oferecem maiores ganhos. Como o programa *Fingerprint* inclui uma etapa de *thresholding*, é possível que as imagens em preto e branco não permitam que todos os possíveis defeitos desse trecho de código sejam revelados.

A Tabela 27 apresenta os ganhos obtidos pela abordagem TAIGA considerando apenas dados de teste gerados a partir das imagens “mais realistas” para cada programa testado, isto é, a partir das imagens que um testador provavelmente selecionaria no teste tradicional. Os programas de PI utilizam características específicas das imagens. O programa Bordas, por exemplo, realça as regiões que apresentam determinadas diferenças de contraste. Uma imagem totalmente uniforme em relação à intensidade, por exemplo, não é adequada para a detecção de bordas.

Tabela 27 – Ganhos de *mutation score* (MS) das imagens de teste geradas pela abordagem TAIGA a partir de imagens iniciais “mais realistas” para os programas de PI utilizados

Programa de PI	Imagem inicial	<i>Mutation score</i> (%)		
		Tradicional	TAIGA	Ganho
Média	Preto e branco	98,85	100	1,15
Bordas	Preto e branco	100	-	0
Mediana	Preto e branco	80,87	100	19,13
Equalização	Nível de cinza	100	-	0
Dilatação	Preto e branco	61,39	100	38,61
Erosão	Preto e branco	98,92	100	1,08
Esqueletização	Preto e branco	90,97	100	9,03
<i>Fingerprint</i>	Impressão digital	98,89	100	1,11

Fonte: Davi Silva Rodrigues, 2017

A utilização de imagens iniciais “mais realistas” possibilita à abordagem TAIGA gerar imagens de teste capazes de revelar mais defeitos em programas de PI complexos. Para os programas mais simples, a abordagem é capaz de gerar imagens de teste eficazes mesmo a partir de imagens iniciais que dificilmente seriam selecionadas por testadores em uma abordagem tradicional, conforme apresentado na Tabela 26.

Ao se utilizar imagens iniciais “realistas”, os programas Bordas e Equalização têm todos os seus mutantes mortos apenas com as imagens iniciais fornecidas (Tabela 27). Nesses casos, a abordagem não gera novas imagens de teste, oferecendo indícios de que o teste de programas simples não seria beneficiado pela abordagem proposta. Verifica-se, entretanto, que programas reais de PI são mais complexos e dificilmente uma única técnica é empregada para resolver um problema real.

As maiores diferenças de ganho de MS podem ser notadas entre os programas de Bordas/Equalização (0%) e Dilatação (38,61%). A análise dos dados das tabelas anteriores e dos códigos-fonte indica que os programas com maiores ganhos de MS apresentam características em comum, assim como aqueles com menores ganhos. Os programas que apresentaram ganho de MS (Média, Mediana, Dilatação, Erosão, Esqueletização e *Fingerprint*) possuem as seguintes características em comum:

- muitos laços (`for`, `while` ou `do...while`);
- muitas condições verificadas, em cláusulas `if`, com uso de operadores lógicos e,
- muitos acessos diretos a posições de matrizes ou chamadas de métodos que recebem coordenadas de *pixels*.

A ferramenta *muJava*, ao analisar programas que apresentam essas características, produz um número maior de mutantes. Esses programas possuem muitos operadores lógicos, de incremento, relacionais e de atribuição, que são justamente os alvos dos operadores de mutação da ferramenta (conforme apresentado na Tabela 5). Dessa forma, é possível criar mais mutantes ao inverter, trocar, incluir ou excluir esses operadores.

Os programas que não apresentaram ganho de MS (Bordas e Equalização) apresentam as seguintes características:

- poucos laços (`for`, `while` ou `do...while`);
- nenhuma ou poucas condições verificadas, em cláusulas `if`, com uso de operadores lógicos;

- poucos acessos diretos a posições de matrizes ou chamadas de métodos que recebem coordenadas de *pixels* e,
- possuem mutantes que apenas são equivalentes devido à validação implícita e/ou explícita dos valores dos *pixels*;

A última característica citada se refere aos mutantes que, caso processassem dados de outros domínios, seriam facilmente mortos pelas imagens de teste geradas. Para o programa Bordas (Algoritmo 2), por exemplo, foram gerados seis mutantes que inserem defeitos em uma cláusula `if` responsável por corrigir eventuais valores de *pixels* que sejam maiores do que 255 (Algoritmo 3).

Algoritmo 2 Trecho do programa Bordas

```

1: if nivelCinza > 255 then
2:   nivelCinza ← 255;
3:
4: img.setPixel(coluna, linha, nivelCinza);
```

Fonte: Davi Silva Rodrigues, 2017

Algoritmo 3 Trecho de mutante do programa Bordas

```

1: if -nivelCinza > 255 then
2:   nivelCinza ← 255;
3:
4: img.setPixel(coluna, linha, nivelCinza);
```

Fonte: Davi Silva Rodrigues, 2017

O código-fonte representado pelo Algoritmo 3 valida o inverso da variável que contém o nível de cinza calculado. Essa mutação permite que o método de definição de valores de *pixels* seja chamado com um valor inválido. No entanto, esse defeito não causa um estado inconsistente durante a execução do programa porque programas de PI possuem validações implícitas e/ou explícitas dos limites superiores e inferiores dos valores dos *pixels*. Essas validações já são tratadas pelos métodos fornecidos pelas bibliotecas para evitar que sejam armazenados valores fora do intervalo de resolução de contraste da imagem. O programa Bordas, por exemplo, possui uma validação explícita (linha 1 do Algoritmo 2) e outra implícita (linha 4 do Algoritmo 2). Como mencionado, as APIs específicas para processamento de imagens incluem essa validação — assim como a classe `BufferedImage` (e as demais classes referenciadas por esta) utilizada no desenvolvimento da abordagem. Isso faz com que o erro introduzido no mutante não gere uma saída diferente, o que poderia acontecer em programas com dados de entrada e saída triviais, como *strings* e números.

No cenário de PI, é necessário realizar uma verificação manual para que tais programas sejam considerados equivalentes.

As validações implícitas e explícitas reduzem a probabilidade de que falhas ocorram nos programas de PI, o que dificulta o teste de mutação. Os trechos de programas mutantes apresentados no Algoritmo 3 e na Tabela 28 são equivalentes devido a essas validações.

Tabela 28 – Exemplos de trechos de programas mutantes equivalentes ao programa Bordas devido às validações implícitas presentes em APIs para imagens

Programa original	Mutantes equivalentes
	<code>if(~nivelCinza > 255)</code>
<code>if(nivelCinza > 255) {</code>	<code>if(nivelCinza == 255)</code>
<code>nivelCinza = 255;</code>	<code>if(false)</code>
<code>}</code>	<code>//bloco if excluído</code>
	<code>if(nivelCinza > 255) {}</code>
	<code>//comando do bloco if excluído</code>

Fonte: Davi Silva Rodrigues, 2017

Não há uma forma simples de se evitar as validações implícitas das APIs de imagem. Ainda que se utilize apenas matrizes numéricas para processar as imagens, a gravação dos resultados e a exibição em tela utilizam as APIs. Para se evitar as validações, é necessário utilizar APIs que não façam validações ou desenvolver/customizar uma API. Essas alternativas demandam grande esforço, não condizem com a realidade do desenvolvimento de programas de PI — que faz uso de APIs para aumentar a produtividade — e podem se tornar inviáveis.

Ao contrário dos programas que não apresentaram ganho de MS, os programas que obtiveram ganhos de MS ao utilizar as imagens geradas pela abordagem TAIGA não foram prejudicados pela validação implícita das APIs de imagem. Os mutantes desses programas são, em sua maioria, relacionados aos operadores lógicos, aos acessos às posições de matrizes e aos laços presentes no código-fonte. Um exemplo de mutante do programa Dilatação (Algoritmo 4) é apresentado pelo Algoritmo 5.

A Tabela 29 apresenta mais exemplos de trechos de programas mutantes que não são afetados pela validação implícita e obtiveram ganhos de MS.

Algoritmo 4 Trecho do programa Dilatação

```

1: //processamento
2: do
3:   cont = 0;
4:   while é possível dilatar a imagem do
5:     //processamento
6:     cont++;
7: while cont != 0

```

Fonte: Davi Silva Rodrigues, 2017

Algoritmo 5 Trecho de mutante do programa Dilatação

```

1: //processamento
2: do
3:   cont = 0;
4:   while é possível dilatar a imagem do
5:     //processamento
6:     cont++;
7: while cont > 0

```

Fonte: Davi Silva Rodrigues, 2017

Tabela 29 – Exemplos de trechos de mutantes equivalentes ao programa Dilatação

Programa original	Mutantes equivalentes
do {	do {...} while(cont++ != 0)
...	do {...} while(cont-- != 0)
} while(cont != 0);	do {...} while(-cont != 0)
for(...;cols < largura;...)	for(...;cols != largura;...)
for(...;i < array.length - 1;...)	for(...;i != array.length - 1;...)
for(...;j < array[i].length-1;...)	for(...;j != array[i].length-1;...)

Fonte: Davi Silva Rodrigues, 2017

6.2.2 Tempo de Execução

As execuções dos experimentos com a abordagem TAIGA apresentaram tempos de execução e número de gerações similares entre programas de complexidades parecidas.

Assim como em 38% dos artigos incluídos no mapeamento sistemático (Capítulo 3), a natureza inerente do AG faz com que o tempo de execução da abordagem proposta seja maior do que os tempos obtidos no teste tradicional. Em comparação ao tempo de execução do teste tradicional, o tempo de execução da abordagem TAIGA é, em média, 457% maior. A Figura 25 apresenta um gráfico com os tempos médios de execução dos testes tradicionais e do AG utilizado pela abordagem TAIGA.

Figura 25 – Gráfico dos tempos de execução médios (em minutos) observados nos 672 experimentos conduzidos com a abordagem TAIGA e o teste tradicional de programas de PI

Fonte: Davi Silva Rodrigues, 2017

Em todos os experimentos, o tempo de execução da abordagem TAIGA foi bem superior ao tempo de execução do teste tradicional. Os programas que possuem maior complexidade (maior número de linhas, mais cláusulas *if*, laços, acessos a posições de matrizes, etc.) foram os que apresentaram as maiores diferenças entre a abordagem proposta e o teste tradicional. Os programas mais simples, por sua vez, apresentaram tempos de execução bastante próximos ao do teste tradicional.

Apesar de não ser o programa mais complexo (Tabela 6), o programa Dilatação apresentou o maior tempo de execução entre todos os programas ao processar a imagem com fundo preto (17 min 57 s). Nesse caso, a execução foi encerrada devido ao critério de parada de estagnação e o MS não chegou aos 100%. Os programas mais complexos, Esqueletização e *Fingerprint*, apesar de não terem apresentado o maior tempo de execução, foram os únicos que tiveram mutantes descartados. A Tabela 30 apresenta as quantidades de programas mutantes que foram gerados pela ferramenta *muJava*, descartados para reduzir o tempo de execução, descartados por serem equivalentes aos programas originais e que, de fato, foram utilizados nos experimentos com cada um dos programas de PI.

Visto que, ao utilizar os 1613 programas mutantes, somente a avaliação da população inicial demorou mais de 60 minutos com o programa Esqueletização, o descarte de mutantes

Tabela 30 – Quantidades de mutantes gerados pela ferramenta *muJava*, descartados, equivalentes e utilizados nos experimentos

Programa de PI	Quantidade de programas mutantes			
	gerados	descartados	equivalentes	utilizados
Média	370	0	22	348
Bordas	183	0	19	164
Mediana	320	0	22	298
Equalização	168	0	14	154
Dilatação	328	0	25	303
Erosão	393	0	24	369
Esqueletização	1613	1294	9	310
<i>Fingerprint</i>	3030	2652	18	360

Fonte: Davi Silva Rodrigues, 2017

tornou-se indispensável para viabilizar as 640 execuções de experimentos com a abordagem TAIGA. Para minimizar os possíveis prejuízos à abordagem, esse descarte de mutantes foi realizado de modo a garantir que haja mutantes representando defeitos em diferentes trechos dos programas originais. Adicionalmente, a quantidade de mutantes dos programas mais complexos (após o descarte) é compatível com a média de 272 mutantes gerados para os demais programas. Nessa abordagem, há o risco de, no pior caso, descartar-se mutantes que revelariam defeitos significativos. No entanto, acredita-se que esse risco tenha sido minimizado, considerando-se que a abordagem procura incluir mutantes com alterações em todos os trechos do programa sob teste.

O tempo de execução da abordagem TAIGA é maior do que o tempo de execução do teste tradicional. Contudo, em termos de média de tempo de execução observado nos experimentos, o maior valor registrado foi de 8 min 12 s (com o programa Dilatação). Considerando que este programa está próximo de um programa aplicado a problemas reais, acredita-se que a utilização da abordagem TAIGA é viável dentro do tempo de execução demandado.

6.2.3 Quantidade de gerações do AG

O número de gerações do AG observado nos experimentos manteve-se baixo. Quanto maior a complexidade do programa, maior é o número médio de gerações (Figura 26).

Apesar de 34% dos estudos analisados no mapeamento sistemático (Capítulo 3) utilizarem a quantidade de gerações do AG como métrica de avaliação, o AG da abordagem

Figura 26 – Gráfico dos números médios de gerações do AG observados nos 640 experimentos realizados com a abordagem TAIGA

Fonte: Davi Silva Rodrigues, 2017

TAIGA apresentou um número baixo de gerações. Foram registradas, no máximo, 15 gerações nos experimentos executados com os programas Erosão e *Fingerprint*. O programa Erosão, além de ser menos complexo do que o programa *Fingerprint*, apresenta menores tempos de execução. O programa Dilatação, que apresentou o maior tempo de execução, não possui a maior quantidade de gerações.

Enquanto o tempo de execução fornece uma indicação da complexidade da geração de imagens de teste para um programa, a quantidade de gerações indica a velocidade com a qual o processo de evolução das soluções ocorre, o que pode auxiliar na definição dos parâmetros genéticos do AG. Uma probabilidade baixa de mutação e uma probabilidade muito alta de *crossover* podem, por exemplo, descaracterizar os indivíduos das novas gerações. Consequentemente, o MS dos novos indivíduos seria menor do que o das gerações anteriores e mais iterações do AG seriam necessárias para se atingir as condições de parada. O número de gerações evidencia esse tipo de situação e permite ao testador observar os efeitos dos parâmetros genéticos ao longo da execução.

6.3 Considerações finais

Os resultados dos experimentos realizados com a abordagem TAIGA mostram que, apesar de exigir tempos de execução maiores do que o do teste tradicional, a abordagem revela mais defeitos em programas de PI. No Capítulo 7 serão apresentadas as conclusões, limitações e trabalhos futuros referentes a esta pesquisa.

7 Conclusões

Programas de processamento de imagens (PI) têm um domínio de entrada bastante complexo e, por essa razão, gerar dados de teste no domínio de imagens é uma tarefa de alto custo, manual e sujeita a imperfeições. Não há, de acordo com o mapeamento sistemático da literatura que foi realizado, uma técnica de geração de dados de teste específica para o domínio de imagens. O mapeamento sistemático indicou, ainda, que algoritmos genéticos têm sido utilizados na geração de dados de teste para programas cujas entradas e saídas são triviais (números e caracteres), com resultados satisfatórios.

O objetivo deste trabalho, definido na Seção 1.2, é propor uma abordagem de geração de dados de teste para programas de PI que seja eficaz. Foram, então, definidos uma representação cromossômica para imagens, métricas de avaliação (*mutation score*, tempo de execução e quantidade de gerações) e uma função *fitness* adaptada para o domínio de imagens.

A hipótese aqui avaliada é a de que as imagens geradas pela abordagem proposta (a partir de um AG) são capazes de revelar mais defeitos do que o teste tradicional realizado em programas de PI.

A avaliação dessa hipótese consistiu em 672 execuções de 64 experimentos. Foram utilizados oito programas de PI e quatro imagens de teste iniciais distintos. Para cada conjunto de programas e imagens, foram realizados testes tradicionais e execuções da abordagem TAIGA. Cada um desses experimentos com AG foram repetidos 20 vezes.

Os resultados expostos no capítulo anterior indicam que, em 75% dos experimentos que utilizaram imagens iniciais “realistas” para os programas, a abordagem TAIGA revelou mais defeitos do que o teste tradicional. Na média, a abordagem obteve ganho de 8,76% em relação ao MS obtido pelo teste tradicional. O maior ganho obtido pela abordagem proposta (ao se utilizar apenas imagens “realistas”) foi de 38,61% para o programa de Dilatação. O programa mais complexo, *Fingerprint*, apresentou ganho de 56,66%, apesar de ter sido utilizada uma imagem de fundo preto, que é menos “realista” para esse programa. Portanto, há indícios de que a abordagem proposta pode produzir resultados significativos em programas mais complexos.

Os programas Bordas e Equalização foram os únicos que não apresentaram ganhos, pois as imagens iniciais já conseguiram revelar todos os defeitos no teste tradicional. Além

de serem programas mais simples (Tabela 6), eles não apresentam as características que fazem com que a ferramenta *muJava* gere mais mutantes (possuem poucos laços, poucas cláusulas *if* e poucos acessos à posições de vetores). Adicionalmente, esses programas foram os que mais apresentaram mutantes equivalentes devido às validações implícitas presentes nas APIs para processamento de imagens.

Portanto, a hipótese enunciada na Seção 1.2 foi parcialmente aceita, pois as imagens geradas pela abordagem proposta revelaram mais defeitos em comparação ao teste tradicional em 75% dos experimentos.

7.1 Limitações da abordagem

A abordagem TAIGA, apesar de poder ser utilizada para testar qualquer programa de PI, exige que os programas sob teste sejam profundamente conhecidos. A definição de imagens iniciais adequadas é fundamental para a obtenção de ganhos relevantes para programas de maior complexidade.

Por utilizar AG, é possível que programas de PI precisem de outros operadores genéticos (mutação, *crossover* e seleção de indivíduos) mais adequados para que a abordagem proposta obtenha ganhos. A definição de operadores genéticos é um passo que pode demandar esforço considerável, visto que é necessário que o programador conheça conceitos de PI, AG e teste de *software*. Por isso, a construção de um *framework* que englobe, além da implementação já disponibilizada no presente trabalho, outros operadores genéticos apropriados ao domínio de imagens, pode contribuir para diminuir esta limitação.

O teste de mutação, utilizado pela função *fitness* do AG, exige que os mutantes equivalentes sejam identificados de forma manual, visto que este é, em geral, um problema computacional indecidível. Adicionalmente, programas mais complexos tendem a dar origem a um número maior de versões mutantes. Quanto mais programas mutantes, maior é o esforço de identificação de mutantes equivalentes e maior é o tempo de execução do AG. No entanto, a escolha dos mutantes que serão descartados a fim de reduzir o número de programas executados pode ser automatizada. No passo de geração de programas mutantes (Seção 5.2), é possível incluir na implementação da abordagem TAIGA a seleção de mutantes de acordo com os parâmetros informados como, por exemplo, a quantidade de mutantes a ser selecionada por tipo de mutação da ferramenta *muJava*.

As imagens utilizadas nos experimentos realizados são de pequenas dimensões e, em sua maioria, preto e branco. O uso de imagens maiores aumentaria consideravelmente o tempo de execução do AG. Em validações futuras, imagens com maiores dimensões podem ser usadas. No entanto, as imagens utilizadas, apesar de serem pequenas, provaram que a abordagem TAIGA é factível para programas de PI simples e complexos.

Apesar das limitações citadas, a abordagem TAIGA contribui com a obtenção de *software* na área de PI com maior qualidade e cuja verificação de qualidade possa ser feita de forma mais efetiva do que o teste tradicional.

7.2 Contribuições

Este trabalho oferece as seguintes contribuições para a área de Sistemas de Informação:

- o mapeamento sistemático apresentado no Capítulo 3 deu origem a um artigo (RODRIGUES et al., 2017) que foi submetido para o periódico *ACM Computing Surveys* e, atualmente, encontra-se em estágio de revisão para a etapa seguinte do processo de submissão;
- a TAIGA é uma nova abordagem para o problema da automatização da geração de dados de teste que propõe a definição de adaptações dos componentes do AG para o domínio de imagens, o que é inédito na literatura;
- foram implementados ao longo desta pesquisa programas de PI, classes de teste *JUnit*, representação cromossômica (classe `Image`), uma função *fitness* que executa teste de mutação, operadores de *crossover* e mutação específicos para imagens, além de relatórios detalhados da execução do AG (a implementação está disponível no GitHub¹);
- foram destacadas as particularidades do domínio de imagens que dificultam o teste de mutação, como as validações implícitas presentes em APIs para processamento de imagens.

A partir do exposto, é possível afirmar que este trabalho apresenta contribuições tanto para a área de Processamento de Imagens quanto para a área de Engenharia de *Software*.

¹ Código-fonte da abordagem TAIGA disponível em <https://github.com/DaviSRodrigues/TAIGA>

7.3 Trabalhos futuros

Há algumas oportunidades para que trabalhos futuros contribuam com a melhoria da abordagem aqui proposta:

- investigar formas de melhorar o tempo de execução do AG como, por exemplo, utilizando GPUs e/ou paralelismo para acelerar o processamento da função *fitness*;
- investigar formas de melhorar o funcionamento dos operadores genéticos;
- conduzir validações com imagens de maiores dimensões;
- otimizar a abordagem TAIGA para programas de PI mais complexos;
- investigar formas de utilizar menos programas mutantes para a execução da função *fitness*, porém sem comprometer o desempenho da abordagem em termos de *mutation score*;
- automatizar a seleção de mutantes que serão descartados a fim de tornar o tempo de execução factível;
- melhorar os operadores genéticos de mutação e *crossover*, que poderiam, por exemplo, ajustar seu comportamento às características do programa de PI sob teste;
- criar um *framework* que envolva mais operadores genéticos, opções de representações cromossômicas, flexibilidade na definição de parâmetros genéticos ao longo da execução e que facilite a geração de programas mutantes.

Referências²

- ACM DIGITAL LIBRARY. Acm Digital Library. 2017. Disponível em: <http://dl.acm.org/>. Acesso em: dezembro de 2016. Citado 3 vezes nas páginas 38, 39 e 122.
- AFZAL, W.; TORKAR, R.; FELDT, R. A systematic review of search-based testing for non-functional system properties. *Information and Software Technology*, v. 51, n. 6, p. 957 – 976, 2009. Citado na página 39.
- AHMED, M. A.; ALI, F. Multiple-path testing for cross site scripting using genetic algorithms. *Journal of Systems Architecture*, p. –, 2015. ISSN 1383-7621. Citado 4 vezes nas páginas 43, 47, 51 e 58.
- AHMED, M. A.; HERMADI, I. GA-based multiple paths test data generator. *Computers & Operations Research*, v. 35, n. 10, p. 3107 – 3124, 2008. ISSN 0305-0548. Part Special Issue: Search-based Software Engineering. Citado 2 vezes nas páginas 43 e 51.
- ALEB, N.; KECHID, S. Automatic test data generation using a genetic algorithm. In: SPRINGER. *International Conference on Computational Science and Its Applications*. [S.l.], 2013. p. 574–586. Citado 4 vezes nas páginas 43, 47, 51 e 59.
- ALESIO, S. D. et al. Combining genetic algorithms and constraint programming to support stress testing of task deadlines. *ACM Transactions on Software Engineering and Methodology (TOSEM)*, ACM, v. 25, n. 1, p. 4, 2015. Citado 4 vezes nas páginas 42, 56, 57 e 59.
- ALETI, A.; GRUNSKE, L. Test data generation with a Kalman filter-based adaptive genetic algorithm. *Journal of Systems and Software*, v. 103, p. 343 – 352, 2015. ISSN 0164-1212. Citado 6 vezes nas páginas 37, 43, 46, 56, 57 e 58.
- ALETI, A.; MOSER, I. A systematic literature review of adaptive parameter control methods for evolutionary algorithms. *ACM Comput. Surv.*, ACM, New York, NY, USA, v. 49, n. 3, p. 56:1–56:35, out. 2016. ISSN 0360-0300. Citado na página 37.
- ALLI, S. et al. A systematic review of the application and empirical investigation of search-based test case generation. *IEEE Transactions on Software Engineering*, v. 36, n. 6, p. 742–762, 2010. ISSN 00985589. Citado 4 vezes nas páginas 16, 28, 29 e 39.
- ALSHRAIDEH, M.; MAHAFAZAH, B. A.; AL-SHARAEH, S. A multiple-population genetic algorithm for branch coverage test data generation. *Software Quality Journal*, Springer, v. 19, n. 3, p. 489–513, 2011. Citado 6 vezes nas páginas 44, 53, 54, 58, 59 e 70.
- ALSMADI, I. Using genetic algorithms for test case generation and selection optimization. In: *Electrical and Computer Engineering (CCECE), 2010 23rd Canadian Conference on*. [S.l.: s.n.], 2010. p. 1–4. ISSN 0840-7789. Citado 4 vezes nas páginas 42, 47, 55 e 59.
- ARORA, A.; SINHA, M. State based test case generation using VCL-GA. In: *IEEE. Issues and Challenges in Intelligent Computing Techniques (ICICT), 2014 International Conference on*. [S.l.], 2014. p. 661–665. Citado 4 vezes nas páginas 42, 47, 54 e 59.

² De acordo com a Associação Brasileira de Normas Técnicas. NBR 6023.

- BARBOSA, E. F. et al. Teste estrutural. In: DELAMARO, M.; MALDONADO, J.; JINO, M. (Ed.). *Introdução ao teste de software*. 1. ed. Rio de Janeiro: Campus, 2007. ISBN 9788535226348. Citado 2 vezes nas páginas 21 e 22.
- BASHIR, M.; BANURI, S. Automated model based software test data generation system. In: *Emerging Technologies, 2008. ICET 2008. 4th International Conference on*. [S.l.: s.n.], 2008. p. 275–279. Citado na página 16.
- BETTS, K.; PETTY, M. Automated search-based robustness testing for autonomous vehicle software. *Modelling and Simulation in Engineering*, Hindawi Publishing Corporation, v. 2016, 2016. ISSN 16875591. Citado 4 vezes nas páginas 17, 42, 49 e 59.
- BOUCHACHIA, A. An immune genetic algorithm for software test data generation. In: *Hybrid Intelligent Systems, 2007. HIS 2007. 7th International Conference on*. [S.l.: s.n.], 2007. p. 84–89. Citado 7 vezes nas páginas 28, 44, 47, 56, 57, 58 e 59.
- BUDD, T. A.; ANGLUIN, D. Two notions of correctness and their relation to testing. *Acta Inf.*, Springer-Verlag New York, Inc., Secaucus, NJ, USA, v. 18, n. 1, p. 31–45, mar. 1982. ISSN 0001-5903. Disponível em: <http://dx.doi.org/10.1007/BF00625279>. Citado na página 24.
- CAO, Y.; HU, C.; LI, L. Search-based multi-paths test data generation for structure-oriented testing. In: *Proceedings of the First ACM/SIGEVO Summit on Genetic and Evolutionary Computation*. New York, NY, USA: ACM, 2009. (GEC '09), p. 25–32. ISBN 978-1-60558-326-6. Citado 4 vezes nas páginas 43, 51, 58 e 59.
- CHEN, Y.; ZHONG, Y. Automatic path-oriented test data generation using a multi-population genetic algorithm. In: *Natural Computation, 2008. ICNC '08. Fourth International Conference on*. [S.l.: s.n.], 2008. v. 1, p. 566–570. Citado 5 vezes nas páginas 44, 46, 53, 54 e 58.
- CHEN, Y.; ZHONG, Y. Experimental study on GA-based path-oriented test data generation using branch distance function. In: *Intelligent Information Technology Application, 2009. IITA 2009. Third International Symposium on*. [S.l.: s.n.], 2009. v. 1, p. 216–219. Citado 2 vezes nas páginas 43 e 51.
- CONCI, A.; AZEVEDO, E.; LETA, F. R. *Computação gráfica: teoria e prática*. Rio de Janeiro: Elsevier, 2008. ISBN 9788535223293. Citado 2 vezes nas páginas 25 e 27.
- DAVE, M.; AGRAWAL, R. Search based techniques and mutation analysis in automatic test case generation: A survey. In: *Advance Computing Conference (IACC), 2015 IEEE International*. [S.l.: s.n.], 2015. p. 795–799. Citado 2 vezes nas páginas 28 e 39.
- DEEPAK, A.; SAMUEL, P. An evolutionary multi population approach for test data generation. In: *Nature Biologically Inspired Computing, 2009. NaBIC 2009. World Congress on*. [S.l.: s.n.], 2009. p. 1451–1456. Citado 5 vezes nas páginas 44, 47, 53, 54 e 58.
- DELAMARO, M.; MALDONADO, J.; JINO, M. *Introdução ao teste de software*. 1. ed. Rio de Janeiro: Campus, 2007. ISBN 9788535226348. Citado 5 vezes nas páginas 16, 17, 20, 21 e 27.
- DEMILLO, R. A.; LIPTON, R. J.; SAYWARD, F. G. Hints on test data selection: Help for the practicing programmer. *Computer*, v. 11, n. 4, p. 34–41, 1978. ISSN 0018-9162. Citado 2 vezes nas páginas 23 e 24.

DERDERIAN, K. et al. A case study on the use of genetic algorithms to generate test cases for temporal systems. In: SPRINGER. *International Work-Conference on Artificial Neural Networks*. [S.l.], 2011. p. 396–403. Citado 4 vezes nas páginas 42, 47, 49 e 59.

DYER, D. *Watchmaker*: framework for evolutionary computation. 2010. Disponível em: <https://watchmaker.uncommons.org/>. Citado na página 65.

EL-SERAFY, A. et al. Enhanced genetic algorithm for MC/DC test data generation. In: *Innovations in Intelligent Systems and Applications (INISTA), 2015 International Symposium on*. [S.l.: s.n.], 2015. p. 1–8. Citado 5 vezes nas páginas 43, 47, 56, 58 e 59.

FABBRI, S. C. P. F.; VICENZI, A. M. R.; MALDONADO, J. C. Teste funcional. In: DELAMARO, M.; MALDONADO, J.; JINO, M. (Ed.). *Introdução ao teste de software*. 1. ed. Rio de Janeiro: Campus, 2007. ISBN 9788535226348. Citado na página 21.

FAN, X. et al. Test data generation with a hybrid genetic tabu search algorithm for decision coverage criteria. In: . [S.l.]: Proceedings of Science (PoS), 2015. v. 12-13-September-2015. Citado 4 vezes nas páginas 44, 56, 57 e 58.

FISCHER, M.; TÖNJES, R. Generating test data for black-box testing using genetic algorithms. In: IEEE. *Proceedings of 2012 IEEE 17th International Conference on Emerging Technologies & Factory Automation (ETFA 2012)*. [S.l.], 2012. p. 1–6. Citado 4 vezes nas páginas 32, 42, 53 e 59.

FOGEL, L. J.; OWENS, A. J.; WALSH, M. J. *Artificial Intelligence through Simulated Evolution*. [S.l.]: John Wiley & Sons, 1966. Citado na página 29.

GARG, D.; GARG, P. Basis path testing using sga & hga with exlb fitness function. In: A.K., M. J. S. (Ed.). [S.l.]: Elsevier, 2015. v. 70, p. 593–602. ISSN 18770509. Citado 4 vezes nas páginas 44, 56, 57 e 59.

GHIDUK, A. S.; GIRGIS, M. R. Using genetic algorithms and dominance concepts for generating reduced test data. *Informatica*, v. 34, n. 3, 2010. Citado 4 vezes nas páginas 43, 51, 52 e 58.

GHIDUK, A. S.; HARROLD, M. J.; GIRGIS, M. R. Using genetic algorithms to aid test-data generation for data-flow coverage. In: *Software Engineering Conference, 2007. APSEC 2007. 14th Asia-Pacific*. [S.l.: s.n.], 2007. p. 41–48. ISSN 1530-1362. Citado 5 vezes nas páginas 43, 47, 51, 58 e 59.

GOLDBERG, D. E. *Genetic Algorithms in Search, Optimization and Machine Learning*. 13. ed. [S.l.]: Addison-Wesley Publishing Company, 1989. Citado 5 vezes nas páginas 29, 30, 33, 34 e 35.

GONG, D.; ZHANG, W.; YAO, X. Evolutionary generation of test data for many paths coverage based on grouping. *Journal of Systems and Software*, v. 84, n. 12, p. 2222 – 2233, 2011. ISSN 0164-1212. Citado 4 vezes nas páginas 45, 53, 54 e 59.

GONZALEZ, R.; WOODS, R. *Processamento de imagens digitais*. 3. ed. Upper Saddle River: Pearson Prentice Hall, 2008. Citado 3 vezes nas páginas 25, 26 e 27.

GOOGLE SCHOLAR. Google Scholar. 2017. Disponível em: <http://scholar.google.com.br/>. Acesso em: 24/02/2016. Citado na página 38.

GUPTA, N.; ROHIL, M. Improving GA based automated test data generation technique for object oriented software. In: *Advance Computing Conference (IACC), 2013 IEEE 3rd International*. [S.l.: s.n.], 2013. p. 249–253. Citado na página 39.

HANH, L.; BINH, N.; TUNG, K. Applying the meta-heuristic algorithms for mutation-based test data generation for simulink models. In: . [S.l.]: Association for Computing Machinery, 2014. v. 04-05-December-2014, p. 102–109. ISBN 9781450329309. Citado 6 vezes nas páginas 23, 43, 47, 56, 57 e 59.

HANH, L. T. M.; BINH, N. T.; TUNG, K. T. A novel fitness function of metaheuristic algorithms for test data generation for simulink models based on mutation analysis. *Journal of Systems and Software*, Elsevier Inc., v. 120, p. 17–30, 2016. ISSN 01641212. Citado 6 vezes nas páginas 17, 23, 43, 47, 50 e 59.

HERMADI, I.; AHMED, M. A. Genetic algorithm based test data generator. In: IEEE. *Evolutionary Computation, 2003. CEC'03. The 2003 Congress on*. [S.l.], 2003. v. 1, p. 85–91. Citado 5 vezes nas páginas 43, 47, 51, 52 e 59.

HOLLAND, J. H. *Adaptation in Natural and Artificial Systems*. [S.l.]: MIT Press, 1975. Citado 3 vezes nas páginas 29, 33 e 34.

HUANG, S.; COHEN, M. B.; MEMON, A. M. Repairing GUI test suites using a genetic algorithm. In: *Software Testing, Verification and Validation (ICST), 2010 Third International Conference on*. [S.l.: s.n.], 2010. p. 245–254. Citado 4 vezes nas páginas 42, 47, 50 e 59.

IEEE XPLORE DIGITAL LIBRARY. Ieee Xplore Digital Library. 2017. Disponível em: <http://ieeexplore.ieee.org/>. Acesso em: dezembro de 2016. Citado 3 vezes nas páginas 38, 39 e 122.

JAMEEL, T. et al. Test image generation using segmental symbolic evaluation for unit testing. In: *Software Engineering, Artificial Intelligence, Networking and Parallel/Distributed Computing (SNPD), 2014 15th IEEE/ACIS International Conference on*. [S.l.: s.n.], 2014. p. 1–6. Citado 2 vezes nas páginas 17 e 38.

JI, H.; SUN, J. Automatic test data generation based on SA-QGA. In: *Computer and Information Science (ICIS), 2012 IEEE/ACIS 11th International Conference on*. [S.l.: s.n.], 2012. p. 611–615. Citado 5 vezes nas páginas 44, 56, 57, 58 e 59.

JIN, R.; JIANG, S.; ZHANG, H. Generation of test data based on genetic algorithms and program dependence analysis. In: IEEE. *Cyber Technology in Automation, Control, and Intelligent Systems (CYBER), 2011 IEEE International Conference on*. [S.l.], 2011. p. 116–121. Citado 3 vezes nas páginas 43, 47 e 51.

KHAN, R.; AMJAD, M. Automatic test case generation for unit software testing using genetic algorithm and mutation analysis. In: *2015 IEEE UP Section Conference on Electrical Computer and Electronics (UPCON)*. [S.l.: s.n.], 2015. p. 1–5. Citado 4 vezes nas páginas 43, 47, 50 e 59.

KHAN, R.; AMJAD, M.; SRIVASTAVA, A. K. Optimization of automatic generated test cases for path testing using genetic algorithm. In: *2016 Second International Conference on Computational Intelligence Communication Technology (CICT)*. [S.l.: s.n.], 2016. p. 32–36. Citado 6 vezes nas páginas 17, 43, 47, 51, 52 e 58.

KHANDELWAL, J.; TOMAR, P. Approach for automated test data generation for path testing in aspect-oriented programs using genetic algorithm. In: IEEE. *Computing, Communication & Automation (ICCCA), 2015 International Conference on*. [S.l.], 2015. p. 854–858. Citado 3 vezes nas páginas 45, 46 e 55.

KIFETEW, F. M. et al. Orthogonal exploration of the search space in evolutionary test case generation. In: *Proceedings of the 2013 International Symposium on Software Testing and Analysis*. New York, NY, USA: ACM, 2013. (ISSTA 2013), p. 257–267. ISBN 978-1-4503-2159-4. Citado 5 vezes nas páginas 44, 47, 56, 57 e 58.

KITCHENHAM, B. Guidelines for performing systematic literature reviews in software engineering. Durham, UK, 2007. Citado na página 38.

KOLEEJAN, C.; XUE, B.; ZHANG, M. Code coverage optimisation in genetic algorithms and particle swarm optimisation for automatic software test data generation. In: . [S.l.]: Institute of Electrical and Electronics Engineers Inc., 2015. p. 1204–1211. ISBN 9781479974924. Citado 4 vezes nas páginas 45, 55, 58 e 70.

KOZA, J. R. *Genetic Programming: On the programming of computers by means of natural selection*. [S.l.]: MIT Press, 1992. Citado 2 vezes nas páginas 29 e 33.

LAKHOTIA, K.; HARMAN, M.; MCMINN, P. A multi-objective approach to search-based test data generation. In: *Proceedings of the 9th Annual Conference on Genetic and Evolutionary Computation*. New York, NY, USA: ACM, 2007. (GECCO '07), p. 1098–1105. ISBN 978-1-59593-697-4. Citado 5 vezes nas páginas 44, 46, 51, 57 e 58.

LEFTICARU, R.; IPATE, F. Automatic state-based test generation using genetic algorithms. In: *synasc*. [S.l.: s.n.], 2007. v. 7, p. 188–195. Citado 4 vezes nas páginas 42, 47, 54 e 59.

LI, B. et al. Automatic test case selection and generation for regression testing of composite service based on extensible BPEL flow graph. In: *Software Maintenance (ICSM), 2010 IEEE International Conference on*. [S.l.: s.n.], 2010. p. 1–10. ISSN 1063-6773. Citado na página 38.

LIJUAN, W.; YUE, Z.; HONGFENG, H. Genetic algorithms and its application in software test data generation. In: IEEE. *Computer Science and Electronics Engineering (ICCSEE), 2012 International Conference on*. [S.l.], 2012. v. 2, p. 617–620. Citado 4 vezes nas páginas 45, 47, 55 e 58.

LIN, P. et al. Test case generation based on adaptive genetic algorithm. In: IEEE. *Quality, Reliability, Risk, Maintenance, and Safety Engineering (ICQR2MSE), 2012 International Conference on*. [S.l.], 2012. p. 863–866. Citado 4 vezes nas páginas 42, 56, 57 e 59.

LINDEN, R. *Algoritmos Genéticos: Uma importante ferramenta da Inteligência Computacional*. 2. ed. Rio de Janeiro: Brasport, 2008. 75 p. ISBN 9788574523736. Citado na página 36.

LIU, D.; WANG, X.; WANG, J. Automatic test case generation based on genetic algorithm. *Journal of Theoretical and Applied Information Technology*, v. 48, n. 1, p. 411–416, 2013. Citado 2 vezes nas páginas 45 e 55.

- LUKE, S. *ECJ: A Java-based Evolutionary Computation Research System*. 2017. Disponível em: <https://cs.gmu.edu/~eclab/projects/ecj/>. Citado na página 65.
- MA, Y.-S.; OFFUTT, J. *Description of muJava's Method-level Mutation Operators*. 2005. Atualizado em jul. 2016. Disponível em: <https://cs.gmu.edu/~offutt/mujava/mutopsMethod.pdf>. Acesso em: 11 nov. 2016. Citado na página 66.
- MA, Y.-S.; OFFUTT, J.; KWON, Y. R. muJava: An automated class mutation system: Research articles. *Software Testing, Verification and Reliability*, John Wiley and Sons Ltd., Chichester, UK, v. 15, n. 2, p. 97–133, jun. 2005. ISSN 0960-0833. Citado na página 66.
- MANIKUMAR, T.; KUMAR, A. J. S.; MARUTHAMUTHU, R. Automated test data generation for branch testing using incremental genetic algorithm. *Sadhana - Academy Proceedings in Engineering Sciences*, Springer India, v. 41, n. 9, p. 959–976, 2016. ISSN 02562499. Citado 4 vezes nas páginas 43, 56, 57 e 58.
- MANN, M. et al. Automatic goal-oriented test data generation using a genetic algorithm and simulated annealing. In: *2016 6th International Conference - Cloud System and Big Data Engineering (Confluence)*. [S.l.: s.n.], 2016. p. 83–87. Citado 5 vezes nas páginas 17, 44, 47, 51 e 52.
- MAYAN, J.; RAVI, T. Test case optimization using hybrid search technique. In: . [S.l.]: Association for Computing Machinery, 2014. v. 10-11-October-2014. ISBN 9781450329088. Citado 5 vezes nas páginas 44, 47, 56, 57 e 58.
- MEFFERT, K. *JGAP: Java Genetic Algorithms and Genetic Programming package*. 2015. Disponível em: <http://jgap.sf.net/>. Citado na página 65.
- MILLER, J.; REFORMAT, M.; ZHANG, H. Automatic test data generation using genetic algorithm and program dependence graphs. *Information and Software Technology*, v. 48, n. 7, p. 586 – 605, 2006. ISSN 0950-5849. Citado 5 vezes nas páginas 44, 47, 56, 57 e 58.
- MITCHELL, M. *An Introduction to Genetic Algorithms*. 5. ed. Cambridge, Massachusetts: MIT Press, 1999. Citado 10 vezes nas páginas 28, 29, 30, 31, 32, 33, 34, 35, 36 e 37.
- MYERS, G. J. *The Art of Software Testing*. [S.l.]: Wiley, 1979. Citado na página 20.
- NARCISO, E. N. *Seleção de casos de teste para sistemas de processamento de imagens utilizando conceitos de CBIR*. 128 f. Dissertação (Mestrado em Sistemas de Informação) — Escola de Artes, Ciências e Humanidades da Universidade de São Paulo, São Paulo, 2013. Citado na página 38.
- NUNES, F. L. S. Processamento de imagens médicas para sistemas de auxílio ao diagnóstico. In: RODELLO, I. A.; BREGA, J. R. F.; KALINKA, R. L. J. (Ed.). *Introdução ao Teste de Software*. 1. ed. Marília (SP): Escola Regional de Informática São Paulo – Oeste, 2006. p. 83–137. Citado na página 25.
- NÚÑEZ, A. et al. Using genetic algorithms to generate test sequences for complex timed systems. *Soft Computing*, Springer, v. 17, n. 2, p. 301–315, 2013. Citado 5 vezes nas páginas 17, 42, 47, 49 e 59.

OFFUTT, A. J.; ROTHERMEL, G.; ZAPF, C. An experimental evaluation of selective mutation. In: *Proceedings of the 15th International Conference on Software Engineering*. Los Alamitos, CA, USA: IEEE Computer Society Press, 1993. (ICSE '93), p. 100–107. Citado na página 25.

PACHAURI, A.; SRIVASATAVA, G. Towards a parallel approach for test data generation for branch coverage with genetic algorithm using the extended path prefix strategy. In: IEEE. *Computing for Sustainable Global Development (INDIACom), 2015 2nd International Conference on*. [S.l.], 2015. p. 1786–1792. Citado 2 vezes nas páginas 43 e 56.

PARTHIBAN, M.; SUMALATHA, M. R. GASE – an input domain reduction and branch coverage system based on genetic algorithm and symbolic execution. In: *Information Communication and Embedded Systems (ICICES), 2013 International Conference on*. [S.l.: s.n.], 2013. p. 429–433. Citado 3 vezes nas páginas 44, 47 e 51.

PEDRINI, H.; SCHWARTZ, W. *Análise de imagens digitais: princípios, algoritmos e aplicações*. 1. ed. São Paulo: Thomson Learning, 2008. ISBN 9788522105953. Citado na página 16.

PENG, X.; LU, L. User-session-based automatic test case generation using GA. *International Journal of Physical Sciences*, Academic Journals, v. 6, n. 13, p. 3232–3245, 2011. Citado 4 vezes nas páginas 42, 47, 55 e 59.

PERENHA, R. A. *Sistema de Processamento de imagens para reconhecimento de impressão digital*. 62 f. Monografia (Graduação em Ciência da Computação) — Centro Universitário Eurípides de Marília, Marília, SP, 2003. Citado na página 69.

PINTO, G. H. L.; VERGILIO, S. R. A multi-objective genetic algorithm to test data generation. In: *Tools with Artificial Intelligence (ICTAI), 2010 22nd IEEE International Conference on*. [S.l.: s.n.], 2010. v. 1, p. 129–134. ISSN 1082-3409. Citado 3 vezes nas páginas 44, 47 e 51.

PRESSMAN, R. S. *Software Engineering: A Practitioner's Approach*. 7. ed. [S.l.]: McGraw-Hill, 2010. Citado na página 23.

RANI, S.; SURI, B. An approach for test data generation based on genetic algorithm and delete mutation operators. In: IEEE. *Advances in Computing and Communication Engineering (ICACCE), 2015 Second International Conference on*. [S.l.], 2015. p. 714–718. Citado 4 vezes nas páginas 43, 47, 50 e 59.

RAPPS, S.; WEYUKER, E. J. Data flow analysis techniques for test data selection. In: IEEE COMPUTER SOCIETY PRESS. *Proceedings of the 6th international conference on Software engineering*. Tokio, Japan, 1982. p. 272–278. Citado na página 22.

RECHENBERG, I. *Evolutionstrategie: Optimierung technischer systeme nach prinzipien der biologischen evolution*. [S.l.]: Frommann-Holzboog Verlag, 1973. Citado na página 28.

RODRIGUES, D. S. et al. Using genetic algorithms in test data generation: a critical systematic mapping. *ACM Computing Surveys*, 2017. Submetido, em fase de revisão. Citado 2 vezes nas páginas 65 e 108.

SCHWEFEL, H. *Evolutionstrategie und numerische Optimierung*. Tese (Doutorado) — Technische Universität Berlin, 1975. Citado na página 28.

SCIENCE DIRECT. Science Direct. 2017. Disponível em: <http://www.sciencedirect.com/>. Acesso em: dezembro de 2016. Citado 3 vezes nas páginas 38, 39 e 122.

SCOPUS. Scopus. 2017. Disponível em: <http://www.scopus.com/>. Acesso em: dezembro de 2016. Citado 3 vezes nas páginas 38, 39 e 122.

SHARMA, N.; PASALA, A.; KOMMINENI, R. Generation of character test input data using GA for functional testing. In: *Software Engineering Conference (APSEC), 2012 19th Asia-Pacific*. [S.l.: s.n.], 2012. v. 2, p. 87–94. ISSN 1530-1362. Citado 3 vezes nas páginas 42, 50 e 59.

SHIMIN, L.; ZHANGANG, W. Genetic algorithm and its application in the path-oriented test data automatic generation. *Procedia Engineering*, v. 15, p. 1186 – 1190, 2011. ISSN 1877-7058. CEIS 2011. Citado 4 vezes nas páginas 44, 51, 58 e 59.

SHUAI, B. et al. Software vulnerability detection based on code coverage and test cost. In: . [S.l.]: Institute of Electrical and Electronics Engineers Inc., 2015. p. 317–321. Citado 5 vezes nas páginas 44, 51, 52, 58 e 59.

SILVA, L. S.; VAN SOMEREN, M. Evolutionary testing of object-oriented software. In: *Proceedings of the 2010 ACM Symposium on Applied Computing*. New York, NY, USA: ACM, 2010. (SAC '10), p. 1126–1130. ISBN 978-1-60558-639-7. Citado 4 vezes nas páginas 17, 45, 55 e 59.

SOLTANI, M.; PANICHELLA, A.; DEURSEN, A. V. Evolutionary testing for crash reproduction. In: . [S.l.]: Association for Computing Machinery, Inc, 2016. p. 1–4. Citado 4 vezes nas páginas 44, 51, 52 e 59.

SOMMERVILLE, I. *Engenharia de Software*. 8. ed. São Paulo: Addison-Wesley, 2007. Citado na página 20.

SUN, J. H.; JIANG, S. J. An approach to automatic generating test data for multi-path coverage by genetic algorithm. In: *Natural Computation (ICNC), 2010 Sixth International Conference on*. [S.l.: s.n.], 2010. v. 3, p. 1533–1536. Citado 3 vezes nas páginas 44, 46 e 51.

TAN, X.; LONGXIN, C.; XIUMEI, X. Test data generation using annealing immune genetic algorithm. In: IEEE. *INC, IMS and IDC, 2009. NCM'09. Fifth International Joint Conference on*. [S.l.], 2009. p. 344–348. Citado 3 vezes nas páginas 44, 56 e 57.

TIAN, T.; GONG, D. Test data generation for path coverage of message-passing parallel programs based on co-evolutionary genetic algorithms. *Automated Software Engg.*, Kluwer Academic Publishers, Hingham, MA, USA, v. 23, n. 3, p. 469–500, set. 2016. ISSN 0928-8910. Citado 4 vezes nas páginas 45, 53, 54 e 59.

TONELLA, P. Evolutionary testing of classes. *SIGSOFT Softw. Eng. Notes*, ACM, New York, NY, USA, v. 29, n. 4, p. 119–128, jul. 2004. ISSN 0163-5948. Citado 2 vezes nas páginas 45 e 55.

USAOLA, M. P.; MATEO, P. R. Mutation testing cost reduction techniques: A survey. *IEEE Software*, v. 27, n. 3, p. 80–86, May 2010. Citado na página 25.

VARSHNEY, S.; MEHROTRA, M. Search-based test data generator for data-flow dependencies using dominance concepts, branch distance and elitism. *Arabian Journal for Science and Engineering*, Springer, v. 41, n. 3, p. 853–881, 2016. Citado 4 vezes nas páginas 17, 44, 47 e 51.

VERGILIO, S. R.; MALDONADO, J. C.; JINO, M. Geração de dados de teste. In: DELAMARO, M.; MALDONADO, J.; JINO, M. (Ed.). *Introdução ao teste de software*. 1. ed. Rio de Janeiro: Campus, 2007. ISBN 9788535226348. Citado na página 27.

VICENZI, A. M. R. et al. Teste de mutação. In: DELAMARO, M.; MALDONADO, J.; JINO, M. (Ed.). *Introdução ao teste de software*. 1. ed. Rio de Janeiro: Campus, 2007. ISBN 9788535226348. Citado na página 23.

VON ZUBEN, F. J. Taxonomia dos algoritmos evolutivos: Parte I. 2011. Universidade Estadual de Campinas. Disponível em: ftp://ftp.dca.fee.unicamp.br/pub/docs/vonzuben/ia707_1s11/notas_de_aula/topico6_IA707_1s11.pdf. Acesso em: 29/06/2016. Citado 6 vezes nas páginas 28, 29, 33, 34, 35 e 36.

VUDATHA, C. P. et al. Automated generation of test cases from output domain of an embedded system using genetic algorithms. In: IEEE. *Electronics Computer Technology (ICECT), 2011 3rd International Conference on*. [S.l.], 2011. v. 5, p. 216–220. Citado 4 vezes nas páginas 17, 42, 50 e 59.

WANG, Z. et al. Test data generation for derived types in c program. In: *Theoretical Aspects of Software Engineering, 2009. TASE 2009. Third IEEE International Symposium on*. [S.l.: s.n.], 2009. p. 155–162. Citado na página 38.

WILHELMSTÖTTER, F. *Jenetics*: Java Genetic Algorithm Library. 2017. Disponível em: <http://jenetics.io/>. Citado na página 65.

XIBO, W.; NA, S. Automatic test data generation for path testing using genetic algorithms. In: IEEE. *2011 Third International Conference on Measuring Technology and Mechatronics Automation*. [S.l.], 2011. v. 1, p. 596–599. Citado 3 vezes nas páginas 32, 44 e 51.

YANG, S. et al. Rga: A lightweight and effective regeneration genetic algorithm for coverage-oriented software test data generation. *Information and Software Technology*, Elsevier B.V., v. 76, p. 19–30, 2016. ISSN 09505849. Citado 4 vezes nas páginas 45, 47, 53 e 54.

YAO, X.; GONG, D. Genetic algorithm-based test data generation for multiple paths via individual sharing. *Computational intelligence and neuroscience*, Hindawi Publishing Corp., v. 2014, p. 29, 2014. Citado 3 vezes nas páginas 45, 53 e 54.

YAO, X.; GONG, D.; WANG, W. Test data generation for multiple paths based on local evolution. *Chinese Journal of Electronics*, IET Digital Library, v. 24, n. 1, p. 46–51, 2015. Citado 3 vezes nas páginas 44, 51 e 52.

YAO, X.; GONG, D.; ZHANG, G. Constrained multi-objective test data generation based on set evolution. *IET Software*, v. 9, n. 4, p. 103–108, 2015. Citado 4 vezes nas páginas 43, 50, 59 e 70.

ZHANG, N.; WU, B.; BAO, X. Automatic generation of test cases based on multi-population genetic algorithm. *Technology*, v. 10, n. 6, 2015. Citado 4 vezes nas páginas 45, 47, 53 e 54.

ZHANG, W. et al. Evolutionary generation of test data for many paths coverage. In: *2010 Chinese Control and Decision Conference*. [S.l.: s.n.], 2010. p. 230–235. ISSN 1948-9439. Citado 5 vezes nas páginas 45, 53, 54, 58 e 59.

ZHANG, Y.; GONG, D. Evolutionary generation of test data for multiple paths coverage with faults detection. In: *2010 IEEE Fifth International Conference on Bio-Inspired Computing: Theories and Applications (BIC-TA)*. [S.l.: s.n.], 2010. p. 406–410. Citado 4 vezes nas páginas 44, 51, 52 e 59.

ZHOU, X.; ZHAO, R.; YOU, F. EFSM-based test data generation with multi-population genetic algorithm. In: IEEE. *Software Engineering and Service Science (ICSESS), 2014 5th IEEE International Conference on*. [S.l.], 2014. p. 925–928. Citado 3 vezes nas páginas 42, 53 e 59.

Apêndices

Apêndice A – Protocolo do mapeamento sistemático

Esta seção descreve com detalhes o protocolo definido para o mapeamento sistemático apresentado no Capítulo 3.

Objetivo

Obter um panorama do estado da arte da pesquisa em geração de dados de teste por meio de algoritmos genéticos.

Questão de pesquisa

Quais são as principais técnicas de geração de dados de teste de *software* que utilizam algoritmos genéticos?

Intervenção

Geração de dados de teste por meio de algoritmos genéticos.

População

Teste de *software* com dados de testes gerados por algoritmos genéticos.

Resultado

Panorama do estado da arte da pesquisa em geração de dados de teste por meio de algoritmos genéticos.

Aplicação

Pesquisadores das áreas de Engenharia de *Software* que utilizam algoritmos genéticos para geração de dados de teste.

Palavras-chave

Evolutionary test, genetic algorithm, test case generation e test data generation.

Métodos de busca

Busca das palavras-chave escolhidas considerando título, resumo e palavras-chave dos artigos. Os resultados retornados foram avaliados (por meio de seus títulos e resumos) de acordo com critérios de inclusão e exclusão. Foram excluídos os resultados que atenderam a um ou mais critérios de exclusão. Foram incluídos os resultados que atenderam ao menos um dos critérios de inclusão e a nenhum dos critérios de exclusão.

Bases de dados

A busca foi realizada nas bases de dados ACM Digital Library (2017), IEEE Xplore Digital Library (2017), Scopus (2017) e Science Direct (2017).

Critérios de inclusão de estudos primários

- I1** Estudos que apresentem técnicas de geração de dados de testes por meio de algoritmos genéticos.
- I2** Estudos que apresentem as vantagens e as desvantagens dos algoritmos genéticos para a geração de dados de testes.

Critérios de exclusão de estudos primários

- E1** Estudos que não foram escritos em inglês.
- E2** Estudos que não passaram por revisão por pares (capítulos de livros, por exemplo).
- E3** Estudos que apresentem revisões de literatura.
- E4** Estudos indisponíveis na *web*.

Critério de qualidade

Os estudos devem apresentar técnicas de geração de dados de teste que utilizem algoritmos genéticos. Não foram considerados os estudos que abordam algoritmos genéticos, porém não têm como objetivo gerar dados de teste.

Apêndice B – Geração de programas mutantes com a ferramenta *muJava*

A seguir são detalhados os procedimentos referenciados no Capítulo 5 e que foram utilizados para gerar programas mutantes.

Instalação da ferramenta e configuração do ambiente

A ferramenta *muJava* requer a configuração de variáveis de ambiente e a definição de um diretório base. A ferramenta e os procedimentos de instalação e configuração estão disponíveis na internet¹.

Tipos de mutação suportados

Os mutantes gerados pela *muJava* podem incluir defeitos em nível de método ou em nível de classe.

Defeitos em nível de método modificam comandos, operandos e operadores:

- acrescentam, excluem ou trocam operadores aritméticos, lógicos, relacionais, binários ou de atribuição e,
- excluem variáveis, comandos, constantes e operadores (neste caso, excluem também um dos operandos).

Defeitos em nível de classe modificam elementos de sintaxe da linguagem Java relacionados ao paradigma de orientação a objetos:

- alteram o modificador de acesso de variáveis e métodos (encapsulamento),
- alteram sobrecarga e sobrescrita de métodos, chamadas de construtores de superclasses, etc. (herança),
- alteram argumentos de métodos sobrecarregados, alteram tipo no *cast* de variáveis, etc. (polimorfismo) e,
- incluem ou excluem palavras reservadas como *this*, *static*, excluem a inicialização de atributos, etc. (recursos específicos do Java).

¹ *muJava Home Page* <<https://cs.gmu.edu/~offutt/mujava/>>

Programa sob teste

O programa sob teste deve ser representado por uma ou mais classes Java. Os programas mutantes são gerados de acordo com os métodos da(s) classe(s) indicada(s) pelo usuário. Cada um dos métodos será analisado e modificado pelos operadores de mutação previamente selecionados. Entretanto, o método `main` é sempre ignorado pelos operadores de mutação e, portanto, nunca é modificado.

Dadas as características da ferramenta, é importante certificar-se de que os trechos de código de interesse estejam localizados em método(s) específico(s). Aqui são considerados de interesse os trechos de código que têm influência direta no resultado do programa. Caso todo o código-fonte seja de interesse para a geração dos mutantes, o método `main` deverá ser vazio. Caso contrário, é recomendado que trechos de menor importância (como, por exemplo, a gravação de *logs*) estejam em métodos separados. Desta forma, é possível reduzir a quantidade de mutantes equivalentes gerados.

Geração de mutantes

A geração de mutantes é feita a partir de um módulo iniciado via linha de comando do sistema operacional (maiores detalhes podem ser encontrados na página da ferramenta).

A interface deste módulo é representada pela Figura 27. O módulo permite a escolha da classe que representa o programa sob teste, a escolha dos operadores de mutação que serão utilizados e o ajuste do nível de *log* do processo de geração dos mutantes. Além de gerar mutantes, também é possível visualizar e comparar os códigos-fonte dos mutantes e do programa sob teste.

Os mutantes gerados são armazenados em diretórios específicos para cada método. Cada diretório, por sua vez, contém subdiretórios específicos para o código-fonte e para o arquivo `*.class` correspondente.

Figura 27 – Módulo de geração e visualização de mutantes da ferramenta *muJava*

Fonte: Davi Silva Rodrigues, 2017

Teste de mutação

Após a geração dos mutantes, o teste de mutação pode ser realizado pela própria ferramenta a partir do módulo representado pela Figura 28.

A execução dos mutantes exige a construção de uma classe *JUnit*² em que cada método represente um caso de teste. Tanto o código-fonte quanto a classe compilada *JUnit* devem ser armazenados em diretório específico definido pela *muJava*.

Devido à própria natureza dos operadores de mutação, é possível que alguns dos mutantes gerados entrem em *loop* infinito. A *muJava* prevê estas situações e oferece opções de *timeout* customizáveis, entretanto, é necessário que os programas mutantes suportem a interrupção de suas *threads* (ver Capítulo 4).

² *JUnit Framework* <<http://junit.org>>

Após a execução dos mutantes, são exibidos os resultados do teste de mutação: número de mutantes, mutantes vivos, mutantes mortos e *mutation score*.

Figura 28 – Módulo de execução e visualização de mutantes da ferramenta *muJava*

TestCase Runner | **Traditional Mutants Viewer** | **Class Mutants Viewer**

☐ Execute only class mutants
☒ Execute only traditional mutants
☐ Execute all mutants

Class : davi.test.bordas.BorderDetection
 Method : davi.genetic.algorithm.Image_metodoGradiente(davi.genetic.alg...
 TestCase : davi.test.bordas.BorderDetectionTest **RUN**
 Time-Out : 3 seconds 3 s

Op	#	Op	#
AORB	28	IHI	0
AORS	2	IHD	0
AOIU	15	IOD	0
AOIS	64	IOP	0
AODU	0	IOR	0
AODS	0	ISI	0
ROR	17	ISD	0
COR	0	IPC	0
COD	0	PNC	0
COI	3	PMD	0
SOR	0	PPD	0
LOR	0	PCI	0
LOI	18	PCC	0
LOD	0	PCD	0
ASRS	0	PRV	0
SDL	8	OMR	0
VDL	10	OMD	0
CDL	4	OAN	0
ODL	14	JTI	0
		JTD	0
		JSI	0
		JSD	0
		JID	0
		JDC	0
		EOA	0
		EOC	0
		EAM	0
		EMM	0
Total : 183		Total : 0	

Traditional Mutants Result

Live Mutants #	
Killed Mutants #	
Total Mutants #	
Mutant Score	

Live

Killed

Class Mutants Result

Live Mutants #	
Killed Mutants #	
Total Mutants #	
Mutant Score	

Live

Killed

Fonte: Davi Silva Rodrigues, 2017