UNIVERSIDADE FEDERAL DE SÃO CARLOS

CENTRO DE CIÊNCIAS EXATAS E DE TECNOLOGIA
PROGRAMA DE PÓS-GRADUAÇÃO EM CIÊNCIA DA COMPUTAÇÃO

AVALIAÇÃO DE GERADORES AUTOMÁTICOS DE DADOS DE TESTE COM ÊNFASE NO TESTE DE MUTAÇÃO

FILIPE SANTOS ARAUJO

ORIENTADOR: PROF. DR. AURI MARCELO RIZZO VINCENZI

São Carlos – SP
Março/2021

UNIVERSIDADE FEDERAL DE SÃO CARLOS

CENTRO DE CIÊNCIAS EXATAS E DE TECNOLOGIA
PROGRAMA DE PÓS-GRADUAÇÃO EM CIÊNCIA DA COMPUTAÇÃO

AVALIAÇÃO DE GERADORES AUTOMÁTICOS DE DADOS DE TESTE COM ÊNFASE NO TESTE DE MUTAÇÃO

FILIPE SANTOS ARAUJO

Dissertação apresentada ao Programa de Pós-Graduação em Ciência da Computação da Universidade Federal de São Carlos, como parte dos requisitos para a obtenção do título de Mestre em Ciência da Computação, área de concentração: Engenharia de software
Orientador: Prof. Dr. Auri Marcelo Rizzo Vincenzi

São Carlos – SP
Março/2021

UNIVERSIDADE FEDERAL DE SÃO CARLOS

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

Folha de Aprovação

Defesa de Dissertação de Mestrado do candidato Filipe Santos Araujo, realizada em 31/03/2021.

Comissão Julgadora:

Prof. Dr. Auri Marcelo Rizzo Vincenzi (UFSCar)

Prof. Dr. Delano Medeiros Beder (UFSCar)

Prof. Dr. Eduardo Noronha de Andrade Freitas (IFG)

O presente trabalho foi realizado com apoio da Coordenação de Aperfeiçoamento de Pessoal de Nível Superior - Brasil (CAPES) - Código de Financiamento 001.
O Relatório de Defesa assinado pelos membros da Comissão Julgadora encontra-se arquivado junto ao Programa de Pós-Graduação em Ciência da Computação.

RESUMO

Contexto: Com o avanço da presença da tecnologia na vida cotidiana, é crucial garantir a qualidade dos produtos de software, o que é alcançável, parcialmente, por meio da atividade de teste. Porém, o teste é altamente oneroso, sendo que o mesmo representa, em muitos casos, mais da metade do custo total do desenvolvimento de software. Dentre os critérios de teste, destaca-se o teste de mutação por ser capaz de revelar defeitos específicos do software. Embora o teste de mutação seja eficaz, as etapas de geração de dados de teste e de análise de mutantes vivos tornam a atividade inviável pelo fato de, ainda, demandarem intensa intervenção humana. **Objetivos**: Sendo assim, este trabalho visa avaliar quatro geradores automáticos de dados de teste (`EvoSuite`, `JTeXpert`, `Palus` e `Randoop`), com ênfase no teste de mutação, utilizando de métricas como adequação, eficácia e custo. Os três primeiros destes geradores implementam heurísticas para a geração e são avaliados tanto de forma individual, quanto em conjunto, tendo como base um gerador aleatório. **Método**: Tais geradores são comparados por meio da condução de um experimento com 33 produtos de software implementados em Java. A adequação é medida em relação ao critério cobertura de comandos, a eficácia é medida em relação ao teste de mutação, e o custo em relação ao número de casos de teste gerados. Para cada gerador são gerados 10 conjuntos de teste diferentes para cada um dos 33 programas analisados, totalizando 1320 conjuntos de testes distintos. **Resultados**: Os resultados demonstram que de forma individualizada, em geral, não é possível concluir que há diferença entre os geradores de dados de teste analisados. Quando combinados, os geradores que utilizam heurísticas apresentam melhores resultados do que o gerador aleatório. A combinação dos geradores inteligentes com o aleatório traz um incremento 0,2% em relação adequação e um incremento de 4% em relação à eficácia com um incremento significativo no custo. **Conclusão**: De maneira geral, observa-se que existe uma complementariedade entre os geradores de dados de teste e os mesmos devem ser combinados para um conjunto de teste de melhor qualidade em relação ao teste de mutação. Entretanto, mesmo com a união de todos os conjuntos de teste gerados, em torno 25% dos mutantes gerados permanecem vivos, indicando que há espaço para melhoria na qualidade dos geradores de dados de teste utilizados.

Palavras-chave: geração de dados de teste, search-based software testing, teste de mutação

ABSTRACT

Context: With the increasing presence of technology in daily life, it is crucial to guarantee the quality of software products, which can be reached by testing activity. However, testing is highly expensive, once it is responsible for, in most cases, more than half of the software development cost. Among the existing testing criteria, mutation testing is highlighted once it is capable of revealing specific faults in software. Although mutation testing is effective, test data generation and live mutants analysis make the activity unfeasible because it is still required considerable human intervention. **Goals**: Therefore, this paper aims at evaluating four automatic test data generators (`EvoSuite`, `JTeXpert`, `Palus`, and `Randoop`), focusing on mutation testing, by measuring adequacy, effectiveness and cost. The first three of these generators implement heuristics during test data generation and they are evaluated both individually and combined, using as baseline a randomic generator. **Method**: Such generators are compared by conducting an experiment with 33 software products implemented in Java. Adequacy is measured regarding statement coverage, effectiveness regarding mutation score and cost regarding the number of generated test cases. For each generator we generated 10 different test sets for each one of the 33 analysed programs, totaling 1320 distincts test sets. **Results**: Results show, when we analyze the generators individually, it is not possible to conclude there is a difference between them. However, when we combine the heuristic-based generators, it is possible to conclude there is significant improvement when compared to the randomic generator. Moreover, when we combine all generators, including the randomic one, we obtained improvments of 0,2% in statement coverage and 4% in effectiveness, but we also obtained a significant increase in cost. **Conclusion**: In general, we observe there is a complementary aspect between the three automatic test data generators and they should be combined to improve the quality of test sets regarding mutation score. However, even with such combination, there is still 25% of the generated mutants which remain alive, indicating there is still room for improvements on quality of the used generators.

Keywords: test data generation, search-based software testing, mutation testing

LISTA DE FIGURAS

LISTA DE TABELAS

SUMÁRIO

Capítulo 1

INTRODUÇÃO

1.1 Contexto e Motivação

A atividade de teste é imprescindível para garantir a qualidade do produto de software. O teste de software consiste em executar o programa total ou parcialmente e avaliar se o mesmo está correto, ou seja, se o programa faz o que foi proposto a fazer. Para isso, é importante se atentar aos seguintes fatores: i) se o software corresponde aos requisitos, que serviram de guia para o projeto e desenvolvimento; ii) se o programa está preparado para lidar com todos os tipos de entradas, sendo estas corretas e/ou incorretas; iii) se o software executa seus procedimentos no tempo esperado; dentre outros fatores.

Entretanto, o processo de teste demanda custo e esforço consideráveis, devido ao fato da cardinalidade do domínio das possíveis entradas de teste tender ao infinito (Delamaro et al., 2017). Chusho (1987) destaca que o teste consome mais da metade do custo total do desenvolvimento do software e, além disso, Bashir e Nadeem (2018) aponta que mais de 70% desse custo é gasto unicamente pela atividade de geração de dados de teste. Como consequência, essa alta demanda de recursos faz com que prática da atividade de teste seja pouco priorizada ou até mesmo totalmente ignorada na indústria. Por isso, foram criadas técnicas de teste, sendo as principais o teste funcional, estrutural e baseado em defeitos, juntamente com seus respectivos critérios. Esses critérios têm como objetivo definir requisitos de teste que particionam o domínio de entrada em subdomínios. Tais requisitos podem ser utilizados tanto para a avaliação da qualidade de conjuntos de teste quanto para apoiar a construção de conjuntos de teste, auxiliando na decisão de quando parar os testes, tornando, assim, a atividade de teste factível.

Um dos critérios de teste que se destaca é o teste de mutação, que consiste em criar versões possivelmente defeituosas do programa a ser testado, chamadas de "mutantes", com o intuito de verificar se as saídas dos mutantes diferem das saídas do

programa original. A diferença desse critério, que faz parte da técnica de teste baseada em defeitos, para os demais é o fato dos subconjuntos aqui definidos possuírem a capacidade de revelar um defeito específico (Delamaro et al., 2017). Sendo assim, o teste de mutação se baseia no seguinte princípio: se o programa a ser testado está se mostrando correto, ou seja, se este gera saídas corretas para um dado conjunto de testes, então seu respectivo mutante terá que, necessariamente, gerar saídas incorretas para o mesmo conjunto de testes. Quando o programa mutante gera saídas diferentes do programa, esse mutante é dito como "morto", caso contrário, é dito "vivo".

Embora o teste de mutação se apresente como uma alternativa que complementa as técnicas funcional e estrutural, o critério possui algumas limitações que tornam a atividade altamente onerosa. Uma delas é em relação a quantidade de mutantes que são gerados para um simples programa ou trecho de programa. Somado a isso, existe o problema da indecidibilidade envolvendo a identificação dos "mutantes equivalentes", ou seja, o fato de não se saber se um dado mutante vivo pode ser morto por algum dado de teste (Delamaro et al., 2017).

Visando lidar com as limitações inerentes ao teste de mutação, trabalhos como o de Vincenzi et al. (2006) e Souza (2017) apresentam aprimoramentos nas etapas do teste de mutantes: sendo que o primeiro trabalho conta com estratégias como a utilização de restrição de operadores de mutação, priorização de mutantes através de Redes Bayesianas e geração de dados de teste randômica, presentes na abordagem ***Muta****tion Testing* ***pro****cess* (*Muta-Pro*); e também com a utilização de técnicas da área de *Search-based Software Testing*, como Algoritmos Evolutivos, na geração de dados de teste presente na ferramenta ***R****each,* ***I****nfect and* ***P****ropagation to* ***Mut****ation* (*RIP-MuT*), respectivamente.

Além das abordagens mencionadas, ferramentas de teste também são abordadas na literatura. Um estudo realizado por Vincenzi et al. (2016) compara o desempenho de ferramentas que geram dados de teste de forma automatizada com o teste manual, mostrando que o último apresenta escores de mutação superiores. Como exemplo, a ferramenta `EvoSuite`, que por sua vez é a ferramenta de geração automática de teste que mais vence prêmios na *SBST 2017 Tool Competition*, apresenta um escore de mutação médio bem abaixo de 80% para programas Java e também bem abaixo da geração manual de teste (Fraser et al., 2018).

1.2 Objetivos

Sendo assim, a proposta deste trabalho é analisar quatro geradores automáticos de dados de teste (`EvoSuite` Fraser e Arcuri (2011), `JTeXpert` (Sakti et al., 2017),

`Palus` (Zhang, 2011) e `Randoop` (Pacheco e Ernst, 2007)), sendo que as três primeiras utilizam heurísticas de busca na geração de dado de teste que melhor se adéqua ao programa passado como entrada, e o último gerador (`Randoop`), usado como *baseline*, se baseia na geração aleatória de dados de teste. Os geradores são avaliados como base em três critérios: i) adequação, com o uso da métrica de cobertura de código; ii) eficácia, com base no teste de mutação; e iii) custo, com base no número de casos de teste[1] gerados pelas ferramentas. O intuito dessa avaliação é saber o quão próximas as ferramentas automáticas, que são o estado da arte em geradores de dados de teste para Java, estão de atingirem 100% de escore de mutação. De acordo com a revisão bibliográfica realizada, o uso de heurísticas de otimização de busca apresenta resultados promissores quando estas são utilizadas com o teste de mutantes. As estratégias visadas neste trabalho têm como foco avaliar o quanto os conjuntos de teste gerados pelas ferramentas atuais conseguem atingir pontos de mutação, ou seja, o quão longe as ferramentas estão de matar 100% de todos os mutantes gerados.

Como objetivos específicos deste trabalho, tem-se:

- Avaliar quatro geradores automáticos individualmente com ênfase no escore de mutação;
- Avaliar os aspectos complementares entre os geradores inteligentes combinados, comparado com o gerador randômico;
- Avaliar os aspectos complementares entre todos os geradores combinados (inteligentes e randômico).

1.3 Organização do Trabalho

Este capítulo fez a introdução da proposta de pesquisa desta Dissertação de Mestrado, fornecendo o contexto, motivações e objetivos do trabalho. Sendo assim, este trabalho está organizado da seguinte maneira: o Capítulo 2 aborda a Fundamentação Teórica; o Capítulo 3 traz a Revisão Bibliográfica; o Capítulo 4 detalha as etapas do experimento; o Capítulo 5 traz a análise dos dados coletados; e, por fim, o Capítulo 6 apresenta as conclusões e os desdobramentos para trabalhos futuros.

[1] Em geral, geradores automáticos de dados de teste, geram, como o próprio nome diz, “dados de teste”, ou seja, os dados de entrada. Os geradores utilizados neste trabalho geram os chamados “casos de teste de regressão”, que são construídos executando-se os dados de teste na unidade em teste e assumindo que o resultado obtido é o resultado esperado. Desse modo, elas são capazes de construir um caso de teste sem ter certeza que a saída obtida é realmente a correta.

Capítulo 2

FUNDAMENTAÇÃO TEÓRICA

2.1 Considerações Iniciais

Neste capítulo serão abordados conceitos, terminologias e critérios de teste de software, com ênfase no Teste de Mutação e também na área de ***S**earch-**b**ased **S**oftware **T**esting* (*SBST*). Este capítulo está organizado da seguinte maneira: a Seção 2.2 introduz Conceitos e Terminologias de Teste, a Seção 2.3 aborda as Técnicas e Critérios de Teste e, por fim, a Seção 2.4 trata sobre a área *Search-based Software Testing*.

2.2 Conceitos e Terminologias de Teste

O teste de software faz parte de um conjunto de atividades chamadas de "VV&T" (Validação, Verificação e Teste) que tem como propósito garantir a qualidade do produto de software. Em outras palavras, o teste consiste em executar o software ou parte de seu código com o intuito de descobrir erros (Myers et al., 2011).

Embora o termo "erro" seja comumente usado para se referir a problemas no programa, na verdade o termo se refere a um estado inconsistente, oriundo de uma execução do software (IEEE, 1990). Este erro é consequência de um "defeito", que por sua vez diz respeito a definições errôneas dos dados. Sendo assim, o defeito é causado por um "engano", que está relacionado a uma ação humana. Por fim, tem-se o conceito de "falha", que ocorre quando a saída gerada pelo software difere da que era esperada, fruto de um erro como definido anteriormente.

Outros conceitos importantes são os de caso de teste e dado de teste. Segundo Myers et al. (2011), o dado de teste é a entrada do programa, enquanto que o caso de teste é o par ordenado contendo o dado de teste e a saída esperada. Sendo assim, o agrupamento dos casos de teste forma, assim, o conjunto de teste. Uma vez definido o

conjunto de teste, o mesmo é executado para um dado programa e então é verificado se as saídas geradas correspondem com as saídas previamente definidas para cada dado de teste. Se houver alguma divergência entre as saídas, tem-se a ocorrência de uma falha, implicando na existência de defeito(s) no software. Como pode ser observado na Figura 2.1, inicialmente é definido o domínio das entradas de teste $D(P)$ para o programa P, formando o conjunto de teste T. Em seguida, T é aplicado a P e o testador então irá verificar se as saídas geradas estão de acordo com suas especificações $S(P)$. Se as saídas estiverem de acordo com $S(P)$, então P obteve êxito na execução. Caso contrário, P obteve falha.

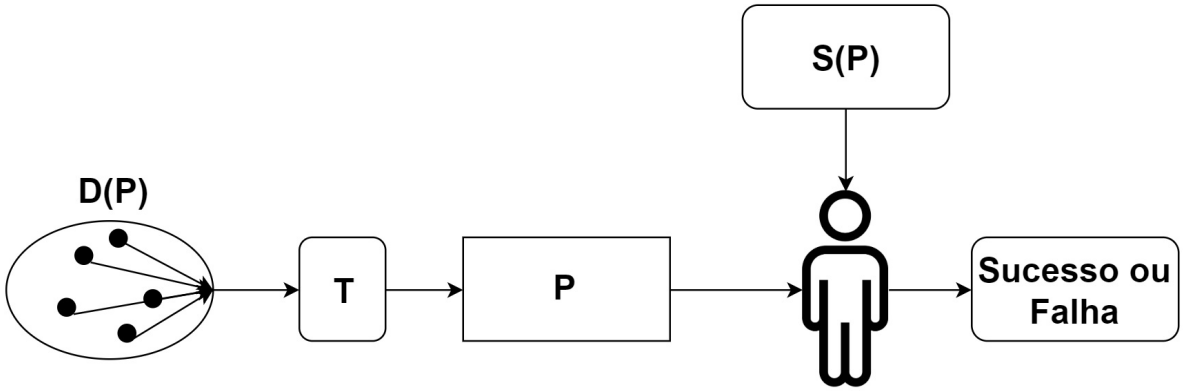

Figura 2.1: Cenário típico da atividade de teste – Adaptado de Delamaro et al. (2017)

Como defeitos podem estar presentes em diferentes níveis do programa, desde definições internas de um módulo até interações entre diferentes módulos, é necessário separar a atividade de teste em fases ou níveis. Por isso, o teste é dividido nas fases de teste de unidade, teste de integração e teste de sistemas. O teste de unidade (ou unitário) diz respeito a análise de pequenas partes do software, como suas classes, módulos e funções e/ou métodos. Como essas partes do software são testadas individualmente, essa fase começa já na primeira iteração do projeto e é conduzida ao longo das demais iterações (Delamaro et al., 2017; Huizinga e Kolawa, 2007, p.4; p.251). Já o teste de integração se preocupa em verificar se as unidades, que foram previamente testadas individualmente, interagem corretamente entre si, por exemplo, via chamadas de procedimentos (Ould e Unwin, 1986, p.71). Por fim, é realizado o teste de sistema, que tem por objetivo validar se tudo o que foi implementando está de acordo com os requisitos definidos (Delamaro et al., 2017).

2.3 Técnicas e Critérios de Software

Apesar da extrema importância da atividade de teste, esta, por sua vez, é altamente dispendiosa. Harrold (2000) e Chusho (1987) destacam que o teste consome mais de cinquenta por cento do custo presente no desenvolvimento de software. Um dos motivos disso é o fato de que é impossível contemplar todo o domínio dos possíveis casos de teste, uma vez que tal domínio pode ser infinito. Por isso, faz-se necessária a utilização de técnicas de teste para que sejam definidos os subdomínios dos casos de teste (Delamaro et al., 2017). Essas técnicas se dividem em Teste Funcional (ou Teste de Caixa-preta), Estrutural (ou Teste de Caixa-branca) e Baseado em Defeitos, sendo que esta última possui o Teste de Mutação como principal critério (Delamaro et al., 2017).

2.3.1 Teste Funcional

Também conhecido como teste de caixa-preta, o teste funcional é uma técnica de projeto de casos de teste que envolve a análise das saídas geradas por um produto de software, dadas suas respectivas entradas. Nessa técnica, detalhes relacionados à implementação do software ficam em segundo plano enquanto que a funcionalidade fruto dessa implementação recebe um maior destaque.

Dessa forma, se a saída gerada por um determinado dado de teste difere da saída esperada pelo caso de teste, tem-se um indício de um ou mais defeitos no programa. Dentre os critérios de teste funcionais, dois mais conhecidos e utilizados são o critério Particionamento de Equivalência e o critério Análise de Valor Limite. Para mais informações sobre critérios de teste funcional o leitor interessado pode consultar (Delamaro et al., 2017).

2.3.2 Teste Estrutural

Já no teste estrutural, também conhecido como teste de caixa-branca, os detalhes de implementação do software são colocados em primeiro plano e, assim, derivam os casos de teste. Sendo assim, as instruções, desvios e caminhos do programa são exercitados pelos casos de teste (Delamaro et al., 2017). Para isso, as rotinas dentro de um programa podem ser vistas pelo teste de caixa-branca como um grafo de fluxo de controle (GFC), sendo que cada instrução é representada por um vértice (nó) enquanto que as transferências de controle entre as instruções são representadas pelas arestas do grafo (Leung e White, 1989). Os nós podem representar tanto instruções

simples, como declaração e uso de variáveis; quanto estruturas mais complexas, como estruturas de repetição e de tomadas de decisão.

Aprofundando-se mais na estrutura do GFC, este representa um dado programa P da forma $GFC(P) = (N, A, e)$, sendo N o conjunto dos nós, A o conjunto de arestas e e sendo o nó de entrada ($e \in N$) (Delamaro et al., 2017). Sendo assim, pode-se definir o conceito de "caminho", que é uma sequência finita de nós interligados por arestas, de modo que o último nó desta sequência seja alcançável partindo do primeiro nó. Logo, um "caminho simples" é um caminho no qual todos os seus nós são distintos, com exceção dos nós inicial e final que podem ser iguais. Caso os nós inicial e final sejam também distintos, o caminho é dito "caminho livre de laço". Por fim, uma sequência de nós é considerada um "caminho completo" quando seu nó inicial corresponde ao nó de entrada e também seu nó final corresponde ao nó de saída do GFC.

Para exemplificar esses conceitos, a Listagem 2.1 mostra o código da função `main` do programa `Identifier` e seu respectivo GFC é mostrado na Figura 2.2. O programa `Identifier` diz se um dado identificador é válido ou não com base na sua cadeia de caracteres. Assim, um identificador é dito válido se, ao mesmo tempo: *(i)* a string começar necessariamente com uma letra; *(ii)* a string conter apenas letras ou dígitos e; *(iii)* o comprimento (`length`) da string deve ser maior ou igual que 1 (um) caractere e menor ou igual a 6 (seis). Caso contrário, o identificador não é válido.

```
1                   int main()
2  /* 1*/       {
3  /* 1*/           char achar;
4  /* 1*/           int length, valid_id;
5  /* 1*/           length = 0;
6  /* 1*/           valid_id = 1;
7  /* 1*/           printf("Identificador: ");
8  /* 1*/           achar = fgetc(stdin);
9  /* 1*/           valid_id = valid_s(achar);
10 /* 1*/           if(valid_id)
11 /* 2*/           {
12 /* 2*/               length = 1;
13 /* 2*/           }
14 /* 3*/           achar = fgetc(stdin);
15 /* 4*/           while(achar != '\n')
16 /* 5*/           {
17 /* 5*/               if(!(valid_f(achar)))
18 /* 6*/               {
19 /* 6*/                   valid_id = 0;
20 /* 6*/               }
21 /* 7*/               length++;
22 /* 7*/               achar = fgetc(stdin);
23 /* 7*/           }
24 /* 8*/           if(valid_id && (length >= 1) && (length <= 6))
25 /* 9*/           {
26 /* 9*/               printf("Valido\n");
```

```
27 /*  9*/            }
28 /*10*/            else
29 /*10*/            {
30 /*10*/                printf("Invalido\n");
31 /*10*/            }
32 /*11*/        }
```

Listagem de Código 2.1: Programa Identifier (função main) – Adaptado de Delamaro et al. (2017)

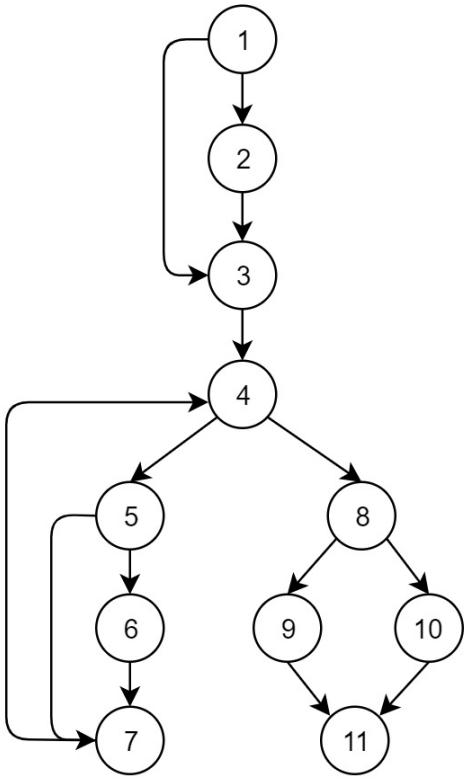

Figura 2.2: GFC do programa Identifier (função main) – Adaptado de Delamaro et al. (2017)

Os números presentes em forma de comentário em cada linha do código na Listagem 2.1 indicam qual vértice do GFC, ilustrado na Figura 2.2, que aquele comando pertence. O primeiro comando `if` do programa (linha 10) introduz um desvio de fluxo de execução, que pode ser seguido pelo arco (1,2) caso a condição seja válida, ou pelo arco (1,3) caso contrário. De forma semelhante, o comando `while` também apresenta um desvio do fluxo de execução, percorrendo o arco (4,5) caso a condição da iteração seja verdadeira, ou pelo arco (4,8), caso seja falsa.

Sendo assim, os casos de testes serão gerados com o intuito de exercitar os componentes do GFC de acordo com qual critério teste que estará sendo utilizado. Esses critérios são normalmente classificados em três grupos: Critérios Baseados em Complexidade; Critérios Baseados em Fluxo de Dados; e Critérios Baseados em Fluxo

de Controle, sendo que este último possui maior relevância neste trabalho. Os critérios mencionados serão abordados nas subseções a seguir.

Teste Baseado em Fluxo de Controle

Nos critérios baseados em fluxo de controle, os casos de teste propostos se preocupam exclusivamente em exercitar os elementos presentes do GFC relativos aos comandos e desvios do programa. Exemplos de coberturas que aplicam esses critérios são: *(i)* Todos-Nós; *(ii)* Todos-Arcos (Todas-Arestas); e *(iii)* Todos-Caminhos. No critério Todos-Nós, o objetivo é percorrer, pelo menos uma vez, todos os vértices do GFC. Já no critério Todos-Arcos, o foco está em executar os desvios do programa, representados pelas arestas do GFC, pelo menos uma vez. Por fim, no critério Todos-Caminhos, todos os caminhos do grafo são percorridos pelos dados de teste.

Um exemplo de aplicação da técnica baseada em fluxo de controle é o "DO-178B, Software Consideration in Airbone Systems and Equipment Certification", que são diretrizes para serem seguidas no desenvolvimento de software de segurança crítica para sistemas de transporte aéreo (RTCA/EUROCAE, 1992). Segundo o DO-178B, a etapa de verificação de um sistema crítico de voo tem como rigor atingir 100% de cobertura do critério Todas-Arestas. Sendo assim, ao aplicar o teste estrutural, é possível encontrar deficiências nos casos de teste, requisitos inadequados ou códigos estranhos dentro do software. A versão mais atual da norma, "DO-178C" tornou as exigências de teste estrutural mais rígidas ao exigir, por exemplo, o cumprimento do critério Cobertura de Condições-Decisões Múltiplas (do inglês, *Multiple Condition-Decision Coverage - MCDC*) (Moy et al., 2013).

Teste Baseado em Fluxo de Dados

Embora o critério baseado em fluxo de controle seja bastante utilizado, o mesmo não se mostra preciso ao revelar defeitos mais simples no programa (Delamaro et al., 2017). Sendo assim, o critério baseado em fluxo de dados visa aumentar o rigor do teste estrutural, com a geração de requisitos de teste baseados nas definições e referências às variáveis e, consequentemente, identificando dependências nos dados do programa a ser testado (Delamaro et al., 2017; Ural e Yang, 1988). Esses critérios são classificados em dois grupos: (i) critérios propostos por Rapps e Weyuker, que são Todas-Definições, Todos-Usos, Todos-Du-Caminhos, dentre outros; e (ii) os critérios Potencias-Usos (Maldonado, 1991).

2.3.3 Teste de Mutação

O Teste de Mutação (ou Teste de Mutantes), como mencionado na Seção 2.3, é um critério da Técnica de Teste Baseada em Defeitos. Essa técnica tem como propósito revelar defeitos por meio da injeção dos mesmos no programa ou trecho de código. Segundo DeMillo et al. (1978), a ideia por trás desse critério é o fato dos programadores explorarem a vantagem que eles têm de desenvolver programas que estão perto de estarem corretos, o que faz com que os programadores possuam uma ideia sobre quais tipos de erros são normalmente cometidos por eles (DeMillo et al., 1978).

Na primeira fase, o conjunto de teste é primeiramente testado com o programa original e em seguida as saídas geradas são comparadas com o que era esperado da especificação do programa. Uma vez que ambas as saídas estão de acordo, diz-se que o programa está correto. Caso contrário, o mesmo é considerado defeituoso e, assim, o teste de mutação é encerrado nessa fase e o programa deve ser corrigido. Observa-se que é necessário ainda nessa fase que exista um testador que saiba decidir se o programa está de fato correto ou o conjunto de teste não é bom o suficiente para revelar defeitos (Vincenzi et al., 2006). Toda vez que o programa em teste sofre qualquer alteração, a segunda fase, de geração de mutantes, deverá ser refeita.

A segunda fase do teste de mutantes consiste na geração dos mutantes, que são versões alternativas do programa a ser testado com o intuito de obter saídas diferentes das do programa original ao testar ambas as versões com as mesmas entradas. Os mutantes diferem do programa original devido a pequenas alterações realizadas pelos operadores de mutação. Como exemplo, na Tabela 2.1 são mostrados três possíveis mutantes para dados trechos de programas, que foram gerados pela aplicação de três operadores de mutação: (i) *u-ORRN*, (ii) *u-OPPO* e (iii) *u-OCNG*. Mais informações sobre os operadores de mutação apoiados pelas ferramenta Proteum/IM podem ser encontradas no relatório de Jorge et al. (2002).

Tabela 2.1: Exemplos de Mutantes (adaptado de Jorge et al. (2002))

Operador	Trecho de Código Original	Trecho de Código Mutante	Função

u-ORRN	`if(i < lp){`
	`  goto teste;`
	`}`
		`if(i >= lp){`
		`  goto teste;`
		`}`
			Substitui um operador relacional por outro operador relacional.

u-OPPO	`if(nfiles <= eargv){`
	`  j++;`
	`}`
		`if(nfiles <= eargv){`
		`  ++j;`
		`}`
			Usa o incremento/decremento prefixo ao invés de incremento/decremento posfixo.

u-OCNG	`if(_argc == 2){`
	`  goto xlong;`
	`}`
		`if(!(argc == 2)){`
		`  goto xlong;`
		`}`
			Inverte condicionais.

Uma vez o programa original tendo passado nos testes e os mutantes gerados, estes mutantes agora são testados na terceira fase com o conjunto de teste e suas saídas são comparadas com as do programa original. Se a saída do programa mutante for diferente do original, então diz-se que esse mutante foi "morto" pelo caso de teste. Caso contrário, diz-se que o mutante está vivo e então é utilizado outro caso de teste para tentar matá-lo novamente. Observa-se que, do ponto de vista do teste de mutação, matar um mutante equivale a eliminar a possibilidade do produto em teste possuir o defeito modelado pelo mutante.

Na quarta e última fase do teste de mutação são analisados os mutantes vivos, ou seja, mutantes que não foram mortos por algum caso de teste. Quando isso ocorre, existem duas possibilidades: (i) os mutantes vivos são identificados como "mutantes equivalentes", ou seja, não existem casos de teste que façam com que o mutante gere uma saída diferente do programa original; ou (ii) o conjunto de teste não é bom o suficiente para matar os mutantes que ainda se mantiveram vivos, sendo necessário melhorar os casos de teste. No fim da análise dos mutantes, é calculado o escore de mutação (do inglês, *mutation score*), que é a razão entre a quantidade de mutantes mortos e o total de mutantes gerados, subtraindo-se destes os mutantes equivalentes. Esse cálculo é demonstrado na Fórmula 2.1, adaptado de Delamaro et al. (2017).

$$ms(P, T) = \frac{DM(P, T)}{M(P) - EM(P)} \tag{2.1}$$

onde:

- P: Programa a ser testado
- T: Conjunto de casos de teste
- $DM(P, T)$: número de mutantes mortos pelo conjunto de casos de teste T;
- $M(P)$: número total de mutantes gerados a partir do programa P; e
- $EM(P)$: número de mutantes gerados que são equivalentes a P.

Os principais problemas presentes na atividade de teste de mutação estão presentes na fases de geração e análise dos mutantes, na segunda e quarta fases, respectivamente. A segunda fase enfrenta o alto custo de geração dos mutantes devido a quantidade de operadores de mutação existentes, possibilitando a criação de vários mutantes por meio de combinações entre os operadores. Estudos mais recentes nessa linha buscam a identificação dos chamados "Mutantes Minimais" (Ammann et al., 2014), descritos na próxima seção. A quarta fase enfrenta o problema da indecidibilidade, seja na geração de dados de teste para percorrer um determinado caminho do GFC, seja na identificação de mutantes vivos que são equivalentes ao programa original (Vincenzi et al., 2006), bem como os custos associados a isso.

2.3.4 Mutantes Minimais

Proposto por Ammann et al. (2014), os "Mutantes Minimais" dizem respeito aos mutantes que são considerados indispensáveis para a atividade de teste de mutação, visando a obtenção de um conjunto de teste adequado ao Teste de Mutação. Seja M o conjunto de todos os mutantes para um dado programa P, T o conjunto de teste de P, m um mutante pertencente a M e t um teste pertencente a T. Uma vez feitas essas definições, o raciocínio por trás dos mutantes minimais é o seguinte: ao se testar P sem considerar os mutantes desnecessários, o resultado obtido deve claramente evidenciar o mesmo resultado que seria obtido ao se testar P com o conjunto de mutantes completo M (Ammann et al., 2014).

Logo, é observado que o conceito de mutantes minimais está diretamente relacionado com o conceito de conjunto de teste minimal. Seja T^m o conjunto de teste minimal de P, isso será verdadeiro se e somente se um caso de teste t_i, pertencente a T, quando removido do conjunto total de teste T, o mesmo não mantém o escore de mutação considerando M e T.

2.4 Search-based Software Testing

***S**earch-**b**ased **S**oftware **T**esting* (*SBST*) é uma subárea de ***S**earch-**b**ased **S**oftware **E**ngineering* (*SBSE*). Sendo assim, *SBSE* trata de transformar os problemas presentes dentro da área de Engenharia de Software em problemas de busca, e propõe soluções baseadas em técnicas de busca meta-heurística para resolvê-los. De forma análoga, a área de *SBST* consiste em transformar a atividade de teste em um problema de busca por casos de teste que se adéquem a algum critério de teste (de Freitas et al., 2010; Harman e Jones, 2001), utilizando também de técnicas de otimização de busca meta-heurística. Um dos focos dessas técnicas de busca é a geração automática de dados de teste (McMinn, 2011). As técnicas de otimização mais utilizadas são o Algoritmo Genético, a Subida da Encosta (do inglês, *Hill Climbing*) e a Têmpera Simulada, que serão abordadas nas seções a seguir.

2.4.1 Algoritmo Genético

Baseado no conceito de evolução proposto por Darwin (Sadeghi et al., 2014), Algoritmos Genético (AG) é uma meta-heurística utilizada na otimização de problemas de busca. Nessa técnica, cada potencial solução para um dado problema é denominado um "indivíduo" ou "cromossomo", sendo o conjunto destes chamado de "população". A

população, por sua vez, é o produto de cada iteração do processo evolutivo, formando assim gerações de população (McMinn, 2011). A Figura 2.3 ilustra as etapas do AG.

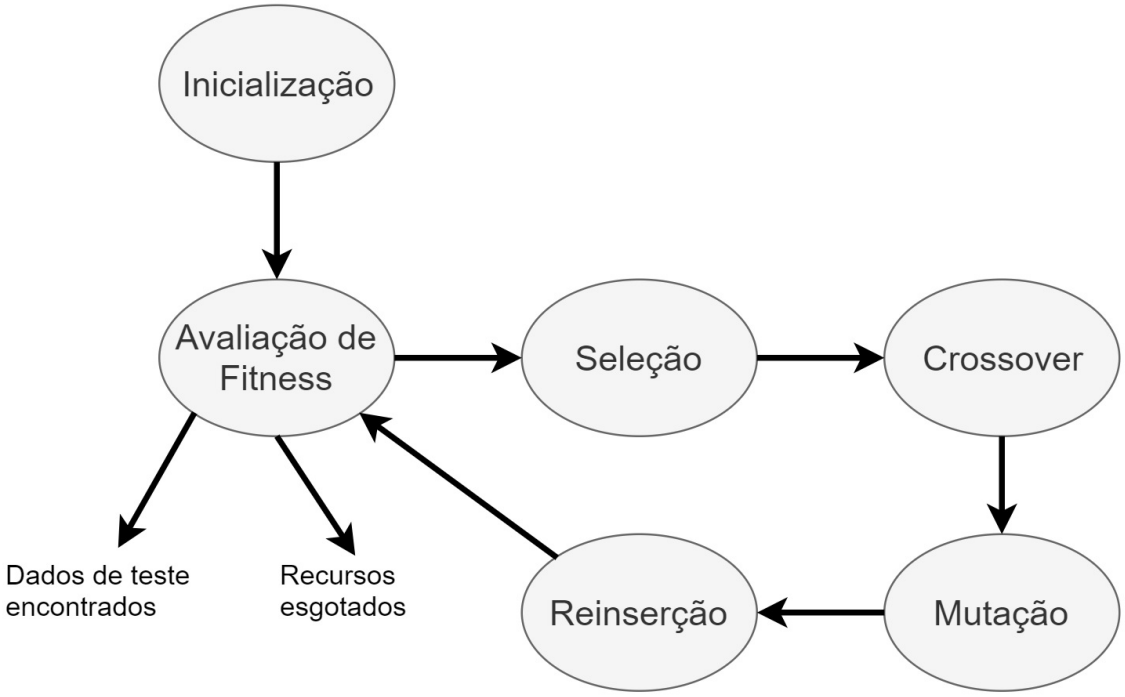

Figura 2.3: Principais passos do Algoritmo Genético – Adaptado de McMinn (2011)

Sendo assim, durante o processo de iteração do AG, cada indivíduo passa por transformações estocásticas para gerar novos indivíduos, como a mutação (biológica) e a recombinação genética (do inglês, *crossover*). Na mutação, novos indivíduos (indivíduos *offspring*) são gerados por meio de mudanças em um único indivíduo, de acordo com a probabilidade de mutação previamente definida. Na etapa de *crossover*, novos indivíduos são formados pela combinação de partes de pares de indivíduos. Logo após, a cada indivíduo da nova geração é atribuído um valor de *fitness*, indicando o quão perto esse indivíduo está da solução do problema de busca, e os indivíduos que possuírem o valor de *fitness* mais adequado serão selecionados na iteração seguinte.

Embora os AGs sejam amplamente usados na literatura, os mesmos podem se deparar com o problema da solução local ótima ao invés da global, dependendo do quão longe a função de *fitness* enxerga o espaço de soluções. Outras desvantagens existentes estão relacionadas à dificuldade de escalabilidade dos AGs quando o problema vai se tornando mais complexo e também ao fato de não se saber se a melhor solução encontrada é de fato a melhor solução para o problema, e não somente a melhor solução entre as demais encontradas.

2.4.2 Subida da Encosta

A técnica de otimização Subida da Encosta (do inglês, *Hill Climbing*) consiste em encontrar a solução local ótima para um dado problema de busca (Souza, 2017). O nome se dá a uma analogia de se "subir uma colina" com o objetivo de se chegar ao seu topo. Em outras palavras, a heurística percorre o espaço amostral de soluções candidatas para um dado problema, partindo de uma solução inicial, e avalia se a próxima solução é melhor que a atual. Se o problema de busca for de maximização do valor da função de *fitness*, o algoritmo para quando encontra o primeiro valor menor que o já encontrado. O mesmo ocorre quando o problema é de minimização, com a diferença de que a heurística irá parar quando encontrar o primeiro valor maior do que o já encontrado. A Figura 2.4 ilustra o funcionamento do *Hill Climbing*.

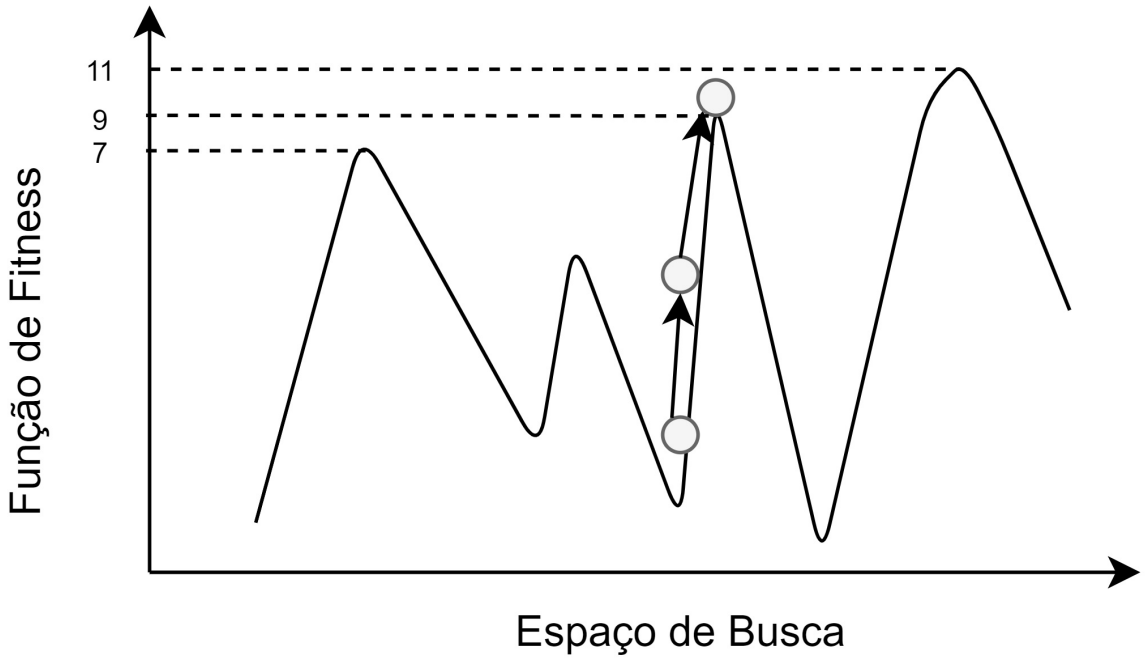

Figura 2.4: Processo de busca do algoritmo Hill Climbing – Adaptado de de Freitas et al. (2010)

Nesse exemplo demonstrado, a heurística Subida da Encosta irá considerar que o maior valor *fitness* é o nove e irá parar, sendo que se a busca continuasse, o valor onze seria enxergado e, consequentemente, seria considerado o valor máximo da função *fitness*. Portanto, embora a meta-heurística seja simples, o mesmo, assim como os AGs, é limitado a encontrar a melhor solução local ao invés da global. Isso significa que se o ponto de partida escolhido não for bom, a heurística não terá um retorno satisfatório do máximo valor *fitness* do problema (de Freitas et al., 2010).

2.4.3 Têmpera Simulada

Na Têmpera Simulada, assim como na Subida da Encosta, abordado na Seção 2.4.2, ocorre a busca pela solução global ótima por meio de "caminhadas" pelo espaço amostral de potenciais soluções para um dado problema. Segundo McMinn (2004), o nome faz analogia com a técnica do processo químico de aquecimento e resfriamento controlado de materiais para aumentar seu tamanho e reduzir seus defeitos. Sendo assim, a Têmpera Simulada corrige o problema da busca cair na solução ótima local da Subida da Encosta ao adotar uma abordagem probabilística no tratamento das soluções que vai encontrando. Fazendo isso, mesmo que o ponto de partida da busca seja visivelmente tendencioso à solução local ótima, o algoritmo permite a adoção de soluções que não parecem virtuosas no momento, mas que poderão mudar o rumo da caminhada para a solução global ótima.

2.5 Considerações Finais

Neste capítulo foram abordados os principais conceitos e definições que envolvem a área de teste de software, com ênfase no teste de mutação. Em paralelo, também foram abordadas estratégias de busca meta-heurísticas que são usadas para diminuir o custo da geração de dados de teste para os critérios de teste. Embora achem bons resultados, as heurísticas podem cair no problema da solução ótima local, mas contribuem para o esforço na geração de dados de teste visando a satisfação de determinado critério de teste.

Capítulo 3

REVISÃO BIBLIOGRÁFICA

3.1 Considerações Iniciais

Este capítulo descreve os trabalhos relacionados ao tema de pesquisa desta qualificação. Sendo assim, este capítulo está organizado da seguinte maneira: a Seção 3.2 aborda a estratégia de revisão adotada, a Seção 3.3 detalha as fases do Mapeamento Sistemático (MS), a Seção 3.4 aborda dois trabalhos de geração de dados para o teste de mutação através de uma Revisão Narrativa e, por fim, a Seção 3.5 trata de ferramentas utilizadas na atividade de teste.

3.2 Estratégia de Revisão Bibliográfica

Zhang et al. (2010) destaca que a maioria das pesquisas realizadas em *SBSE* estão relacionadas com a otimização da atividade de teste e depuração. A Figura 3.1 mostra o gráfico com as porcentagens dos estudos publicados em *SBSE*.

Como pode ser observado no gráfico, enquanto que as demais áreas apresentam, individualmente, no máximo 10% de trabalhos realizados, os trabalhos na área de teste apresentam mais de 50%, o que demonstra um grande foco de atenção em *SBST*.

Considerando esse cenário, a revisão bibliográfica deste trabalho será constituídas de duas partes: *(i)* foi feita uma atualização do Mapeamento Sistemático (MS) realizado por Souza (2017) em novembro de 2016, considerando os estudos posteriores a essa data; e *(ii)* foi feita uma Revisão Narrativa (Rother, 2007), na qual alguns trabalhos foram manualmente escolhidos de acordo com suas respectivas familiaridades com o tema desta proposta de trabalho.

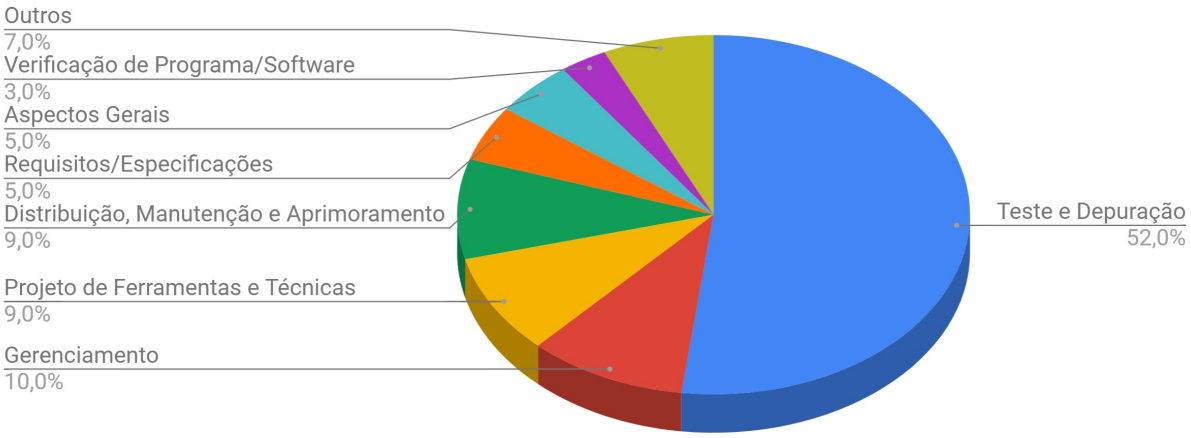

Figura 3.1: Porcentagem de artigos publicados entre 1976 e 2010 – Adaptado de Zhang et al. (2010)

3.3 Mapeamento Sistemático

O objetivo do Mapeamento Sistemático é fazer o levantamento de estudos na literatura que evidenciam a relevância de um determinado tema. Neste caso, o tema em questão é a geração de dados de teste para o teste de mutação. Sendo assim, o MS é composto pelas seguintes fases: (*i*) Planejamento, no qual é definida uma string de busca a fim de se obter os estudos que envolvem o tema proposto; (*ii*) Condução, na qual são lidos o título e abstract/resumo dos estudos e, em seguida, são aplicados os critérios de inclusão/exclusão nos mesmos; e (*iii*) Análise, na qual são lidas a introdução e conclusão dos estudos restantes e, novamente, são aplicados os critérios de inclusão/exclusão nos mesmos. As fases mencionadas serão detalhadas nas seções as seguir.

3.3.1 Planejamento do Mapeamento Sistemático

O planejamento do MS foi realizado de forma semelhante a Souza (2017). Para isso, foi usada o *Parsifal*, que é uma ferramenta online que auxilia na revisão sistemática no contexto da área de Engenharia de Software. Sendo assim, *Parsifal* implementa a técnica *Population, Intervention, Comparison and Outcomes* (*PICO*), que será utilizada nesse planejamento. A técnica *PICO* é detalhada a seguir:

- **População:** critério de teste de mutação
- **Intervenção:** técnicas para a geração de dados de teste.
- **Comparação:** não se aplica nesse MS.

- **Resultados:** visão dos estudos primários que apresentam a utilização de técnicas para geração de dados de teste no contexto do teste de mutação

A partir da definição dos termos acima, foi definida a questão de pesquisa baseada em Souza (2017):

- **QP$_1$:** Quais técnicas têm sido utilizadas para geração de dados de teste no contexto de teste de mutação?

3.3.2 Estratégia de Busca

Uma vez definida a questão de pesquisa, a *string* de busca é então criada com base nas mesmas. Os elementos que compõem a *string* são: (*i*) palavras-chaves de acordo com as questões de pesquisa; (*ii*) sinônimos das palavras-chaves; (*iii*) operador booleano "*OR*" fazendo a disjunção entre os sinônimos; e (*iv*) operador booleano "*AND*" para fazer a conjunção entre as palavras-chaves. A Figura 3.2 ilustra a estratégia descrita.

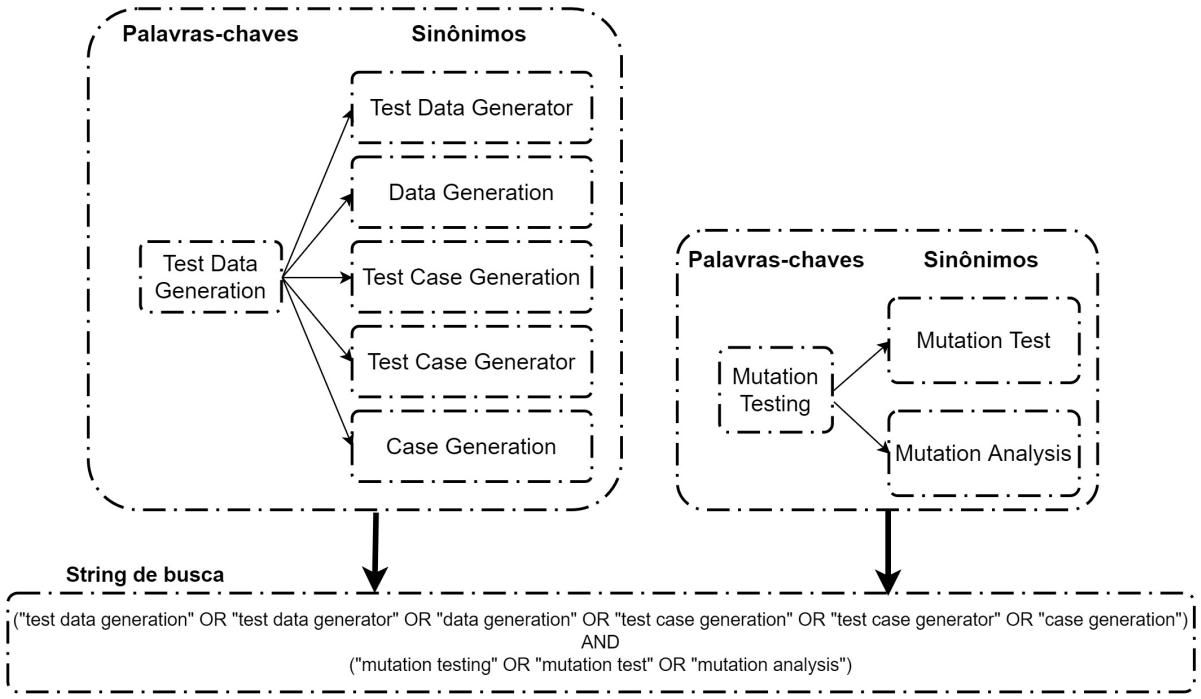

Figura 3.2: *String* de Busca das questões de pesquisa – Adaptado de Souza (2017)

Depois de criada a *string* de busca, as mesmas foram executadas nas bases de dados, de forma semelhante a Souza (2017). A Tabela 3.1 mostra as bases de dados utilizadas neste trabalho, juntamente com seus respectivos endereços.

Tabela 3.1: Bases de Dados – Adaptado de Souza (2017)

Bases de Dados	Endereço
ACM Digital Library	`www.portal.acm.org`
Compendex	`www.engineeringvillage.com`
IEEE Xplore	`www.ieeexplore.ieee.org`
ScienceDirect	`www.sciencedirect.com`
Scopus	`www.scopus.com`
Springer	`www.springerlink.com`
Web of Knowledge	`www.isiknowledge.com`

3.3.3 Critérios para Seleção dos Estudos

Com a estratégia de busca definida, é preciso então avaliar os estudos que serão retornados quando a *string* de busca for executada nas bases de dados. Para isso, foram estabelecidos critérios de inclusão e exclusão que serão aplicados durante as fases 2 e 3 do MS. Os critérios são listados a seguir:

- **Critérios de Inclusão:**
  - **CI_1:** Estudos primários que apresentam pelo menos uma técnica/abordagem para geração de dados de teste para o teste de mutação.
- **Critérios de Exclusão:**
  - **CE_1:** Estudos primários que não mencionam as técnicas de geração de dados de teste no abstract;
  - **CE_2:** Estudos primários repetidos, neste caso, o estudo menos completo será excluído;
  - **CE_3:** Estudos primários que não estejam escritos em inglês ou português;
  - **CE_4:** Estudos primários que não sejam *full paper* ou *short paper* (por exemplo, pôsters, tutoriais, relatórios técnicos, teses e dissertações);
  - **CE_5:** Estudos primários com texto ou resultados incompletos;
  - **CE_6:** Estudos anteriores a 2017.

3.3.4 Extração dos Dados

Na fase de extração, foi coletado os nomes das técnicas de geração automática de dados utilizadas e como elas foram utilizadas.

3.3.5 Execução do Mapeamento Sistemático

A *string* base foi adaptada para as diferentes máquinas de busca como mostrado na Figuras 3.3, 3.4, 3.5, 3.6, 3.7, 3.8 e 3.9.

TITLE-ABS-KEY(("test data generation" **OR** "test data generator" **OR** "data generation" **OR** "test case generation" **OR** "test case generator" **OR** "case generation") **AND** ("mutation testing" **OR** "mutation test" **OR** "mutation analysis"))

Figura 3.3: *String* de Busca da Scopus – Elaborado pelo autor

```
+(
(+(acmdlTitle:("test data generation") acmdlTitle:("test data generator") acmdlTitle:("data
generation") acmdlTitle:("test case generation") acmdlTitle:("test case generator")
acmdlTitle:("case generation")) +(acmdlTitle:("mutation testing") acmdlTitle:("mutation
test") acmdlTitle:("mutation analysis")))

(+(recordAbstract:("test data generation") recordAbstract:("test data generator")
recordAbstract:("data generation") recordAbstract:("test case generation")
recordAbstract:("test case generator") recordAbstract:("case generation"))
+(recordAbstract:("mutation testing") recordAbstract:("mutation test")
recordAbstract:("mutation analysis")))

(+(keywords.author.keyword:("test data generation") keywords.author.keyword:("test data
generator") keywords.author.keyword:("data generation") keywords.author.keyword:("test
case generation") keywords.author.keyword:("test case generator")
keywords.author.keyword:("case generation")) +(keywords.author.keyword:("mutation
testing") keywords.author.keyword:("mutation test") keywords.author.keyword:("mutation
analysis")))
)
```

Figura 3.4: *String* de Busca da ACM – Elaborado pelo autor

(((("test data generation" **OR** "test data generator" **OR** "data generation" **OR** "test case generation" **OR** "test case generator" **OR** "case generation") **AND** ("mutation testing" **OR** "mutation test" **OR** "mutation analysis")) WN KY)

Figura 3.5: *String* de Busca da Compendex – Elaborado pelo autor

Souza (2017) relata que foi necessário fazer a inversão da ordem da *string* de busca na base *IEEE Xplore* com o intuito de obter um maior número de estudos. Neste estudo,

(("test data generation" **OR** "test data generator" **OR** "data generation" **OR** "test case generation" **OR** "test case generator" **OR** "case generation")
AND
("mutation testing" **OR** "mutation test" **OR** "mutation analysis"))

Figura 3.6: *String* de Busca da Science@Direct – Elaborado pelo autor

(("test data generation" **OR** "test data generator" **OR** "data generation" **OR** "test case generation" **OR** "test case generator" **OR** "case generation")
AND
("mutation testing" **OR** "mutation test" **OR** "mutation analysis"))

Figura 3.7: *String* de Busca da Web of Knowledge – Elaborado pelo autor

(("Document Title":"test data generation" **OR** "Document Title":"test data generator" **OR** "Document Title":"data generation" **OR** "Document Title":"test case generation" **OR** "Document Title":"test case generator" **OR** "Document Title":"case generation")
AND
("Document Title":"mutation testing" **OR** "Document Title":"mutation test" **OR** "Document Title":"mutation analysis"))

OR

(("Abstract":"test data generation" **OR** "Abstract":"test data generator" **OR** "Abstract":"data generation" **OR** "Abstract":"test case generation" **OR** "Abstract":"test case generator" **OR** "Abstract":"case generation")
AND
("Abstract":"mutation testing" **OR** "Abstract":"mutation test" **OR** "Abstract":"mutation analysis"))

OR

(("Author Keywords":"test data generation" **OR** "Author Keywords":"test data generator" **OR** "Author Keywords":"data generation" **OR** "Author Keywords":"test case generation" **OR** "Author Keywords":"test case generator" **OR** "Author Keywords":"case generation")
AND
("Author Keywords":"mutation testing" **OR** "Author Keywords":"mutation test" **OR** "Author Keywords":"mutation analysis"))

Figura 3.8: *String* de Busca da IEEE Xplore – Elaborado pelo autor

a inversão da string não apresentou alterações com relação ao número de estudos retornados pela base de dados.

Após a execução das *strings*, 977 estudos foram retornados ao todo, sendo que nos final do processo de seleção foram retornados 10 estudos. A Figura 3.10 ilustra os passos da execução do mapeamento sistemático.

("test data generation" **OR** "test data generator" **OR** "data generation" **OR** "test case generation" **OR** "test case generator" **OR** "case generation")
AND
("mutation testing" **OR** "mutation test" **OR** "mutation analysis")

Figura 3.9: *String* de Busca da Springer – Elaborado pelo autor

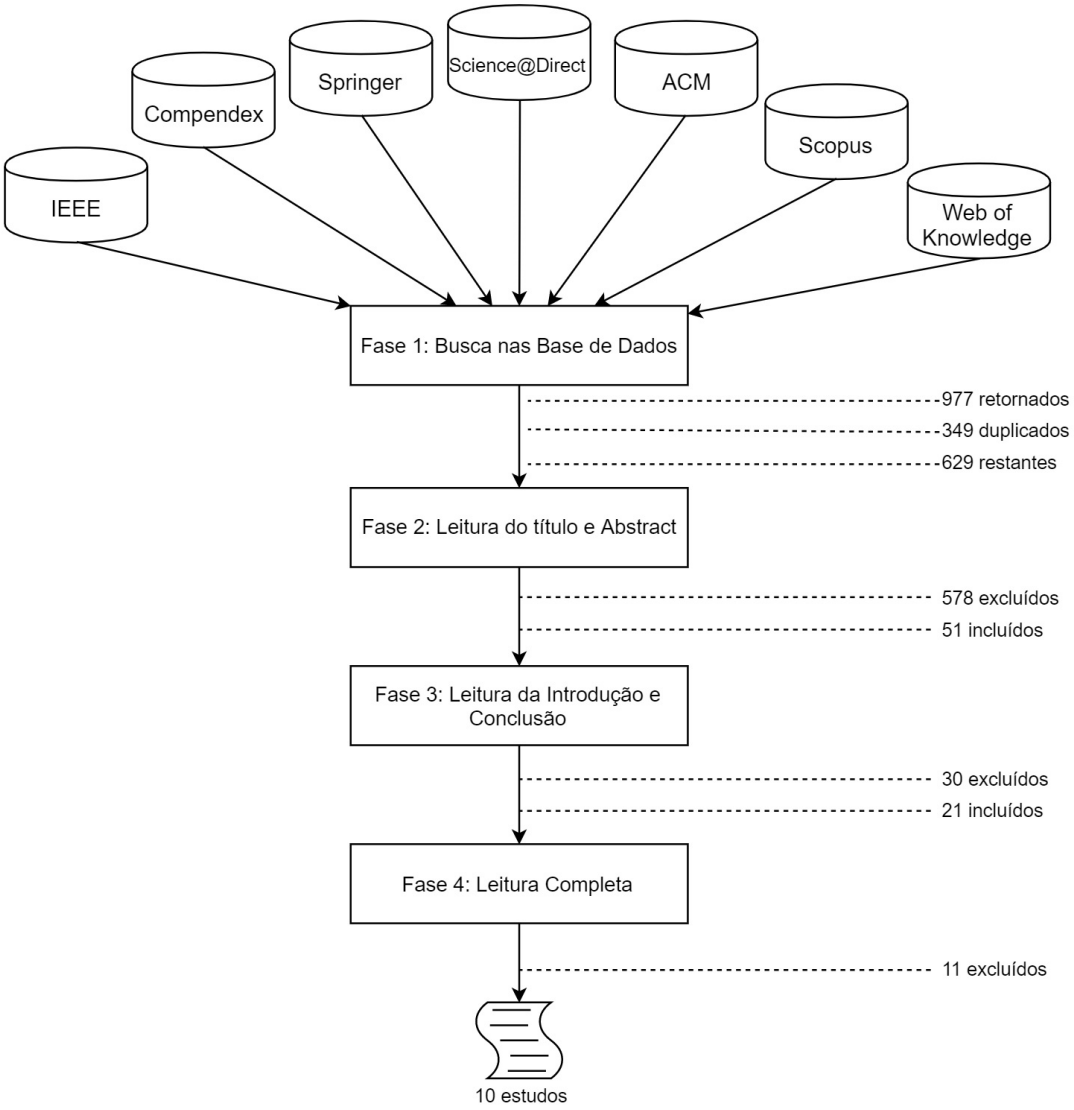

Figura 3.10: Fases do Mapeamento Sistemático – Elaborado pelo autor

3.3.6 Discussão e Análise dos Estudos do MS

Liu e Chen (2017a) e Liu e Chen (2017b) utilizam AG juntamente com a clusterização K-means para redução do número de mutantes. Sua abordagem consiste em três etapas: *(i)* geração preliminar dos dados de teste com base na linguagem WADL (*Web Application Description Language*), utilizando os critérios de particionamento de equivalência e análise de valor limite (Seção 2.3.1); *(ii)* design dos operadores de mutação baseados em WADL, sucedido do uso de AG que suporta clusterização

K-means para eliminar os mutantes redundantes e; *(iii)* seleção de dados de teste com base na capacidade dos mesmos de matar os mutantes selecionados na etapa anterior. Como resultados, o número de mutantes e, principalmente, o número de dados de teste gerados caíram drasticamente.

Khan et al. (2017) apresentam uma abordagem híbrida do AG, que envolve a utilização do critério de teste Todos-caminhos (Seção 2.3.2), com o teste de mutação. O processo evolutivo de geração de dados de teste é pautado pelo escore de mutação e se segue até cobrir todos os caminhos do GFC ou até atingir o limite de iterações definido. Os resultados mostraram que o AG híbrido supera as técnicas randômicas e o AG padrão. O trabalho de Chuaychoo e Kansomkeat (2017) segue uma abordagem semelhante, com a diferença de utilizar o cálculo *Branch Distance* como guia da função de *fitness*.

Chaudhary e Kumar (2017) utilizam a técnica *Artificial Bee Colony-Penguin Search Optimization* (*ABC-PeSO*), que é a fusão das técnicas de otimização Colônia Artificial de Abelhas e a Otimização de Busca dos Pinguins. Ambas as heurísticas se baseiam no comportamento biológico de pinguins e abelhas na busca por alimento.

Bashir e Nadeem (2018) utilizam três variações do AG na geração de dados de teste para a linguagem Java e as compara com o teste randômico. Os mutantes são gerados pelo gerador de mutantes, que aplica os operadores de mutação fornecidos pelo usuário, podendo estes ser relacionados à programação estruturada ou orientada a objetos. De forma semelhante, Rani et al. (2019) também utilizam AG para a geração de dados de teste para os mutantes que são gerados pelo gerador *MuJava*, que utiliza operadores de mutação de deleção.

Um outro método utilizado na geração de dados de teste é a Programação Dinâmica. Em Jatana et al. (2019), os mutantes são gerados pela ferramenta *Milu* e os dados de teste são gerados de forma randômica. Num segundo momento, os dados de testes são executados nos mutantes gerados, sendo armazenados numa matriz os dados de testes e os respectivos mutantes mortos por eles.

Ghiduk e El-Zoghdy (2018) utilizam AG na geração de dados de teste para o teste de mutação para programação concorrente. Nessa abordagem, os mutantes gerados possuem mais de um defeito inserido em seu código para exercitar os aspectos concorrentes do mesmo.

O trabalho de Bashir e Nadeem (2017) utiliza uma variação do AG na geração de dados de teste para o teste de mutação para programas orientados a objetos. Essa adaptação proposta do AG foi implementada na ferramenta eMuJava e envolve modificações nas etapas de *crossover* e mutação dos genes a fim de considerar o estado atual do objeto instanciado no programa mutante. Os resultados mostram um melhor desempenho da abordagem proposta em relação à qualidade dos dados

de teste gerados juntamente com o custo do processo quando comparada ao teste randômico, ao AG em si e também à ferramenta `EvoSuite`.

3.4 Revisão Narrativa

Nesta seção será realizada a Revisão Narrativa (Rother, 2007), composta de estudos primários e secundários de trabalhos envolvendo a geração automatizada de dados de teste para os critérios de teste baseados em fluxo de controle e também para o teste de mutação.

3.4.1 Geração de Dados de Teste para Teste de Fluxo de Controle

A revisão bibliográfica realizada por McMinn (2004) apresenta estudos que envolvem estratégias de geração automática de dados para o teste de caixa-branca. As principais estratégias são: (*i*) a geração de dados de teste estrutural estática e dinâmica, (*ii*) Têmpera Simulada (do inglês, *Simulated Annealing*) e (*iii*) Algoritmo Evolutivo, sendo que essa última tem se mostrado uma tendência maior que as demais através do uso do AG.

Na geração de dados estrutural estática, o foco está puramente na análise dos caminhos presentes na estrutura do programa, não necessitando que o mesmo seja executado. Um exemplo dessa abordagem é a técnica Execução Simbólica (do inglês, *Symbolic Execution*), a qual analisa os possíveis caminhos presentes na estrutura do programa e então atribui valores simbólicos a fim e exercitá-los sem necessariamente executar o programa. Já na geração de dados estrutural dinâmica, o programa é executado através de técnicas de busca ou simplesmente teste aleatório a fim de exercitar a estrutura do programa. Embora o teste aleatório seja muito utilizado, McMinn (2004) destaca que essa técnica deixa a desejar por não percorrer caminhos que somente são percorridos com dados de teste específicos.

Na aplicação das metaheurísticas Têmpera Simulada e Algoritmo Genético, ambas abordadas nas Seções 2.4.3 e 2.4.1, a busca metaheurística tem sido guiada pelo cálculo da distância entre ramos (do inglês, *Branch Distance*) a fim de percorrer todas as arestas e/ou nós do GFC.

Embora técnicas de otimização tenham sido usadas no teste estrutural, existem algumas limitações quando programas não numéricos são levados em consideração, como programas contendo *strings* e outros tipos de estruturas de dados. Outra limitação é com relação a programa orientados a objetos, devido a dificuldade da heurística de

geração de dados de testes de lidar com o estado objetos instanciados a partir de classes herdeiras.

3.4.2 Geração de Dados de Teste para o Teste de Mutação

Nesta seção é abordado um trabalho, oriundo da Revisão Narrativa, diretamente relacionado com a proposta deste trabalho.

RIP-MuT

A abordagem ***R**each, **I**nfect and **P**ropagation to **Mu**tation **T**esting* (*RIP-MuT*), é baseada na técnica de *Hill Climbing*, abordado na Seção 2.4.2. Seu objetivo é automatizar a geração de dados de teste para o teste de mutação e também sugerir, dentre os mutantes que ainda estão vivos, quais são os equivalentes. A Figura 3.11 apresenta uma visão geral da abordagem *RIP-MuT*, a qual é dividida nos módulos de geração de dados de teste e sugestão de mutantes equivalentes.

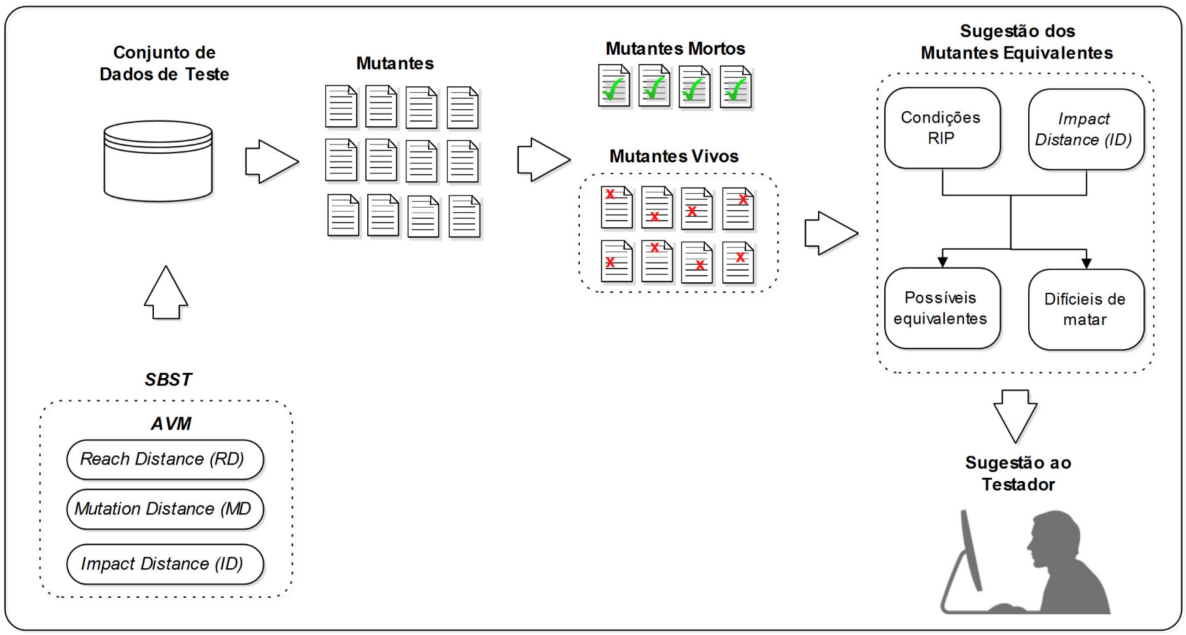

Figura 3.11: Visão geral da RIP-MuT – Adaptado de Souza (2017)

Sendo assim, inicialmente são gerados os dados de teste através da utilização do método *Alternating Variable Method* (*AVM*), baseado no algoritmo *Hill Climbing* e que implementa as métricas *Reach Distance* (*RD*), *Mutation Distance* (*MD*) e *Impact Distance* (*ID*). A métrica *RD* é responsável por alcançar o ponto de mutação do programa. Em seguida, a métrica *MD* irá exercitar o ponto de mutação com o intuito de causar uma diferença na execução do mutante em relação ao programa original. Por fim, a métrica *ID* é responsável por fazer essa diferença de execução ser propagada para a saída do programa (Souza, 2017). Portanto, com a utilização dessas métricas

de *SBST*, é possível medir a adequação dos dados de teste com o auxílio de uma função de *fitness* e assim melhorá-los incrementalmente a fim de se obter uma solução ótima (Souza, 2017).

3.5 Ferramentas de Geração de Dados de Teste

Como já abordado neste trabalho, as diferentes técnicas e critérios de teste buscam sistematizar a atividade de teste a fim de torná-la viável. Para que isso seja não só possível como também prático, é imprescindível a utilização de ferramentas que dão suporte para a aplicação dos critérios de teste, como o teste de mutação e o teste estrutural. Sendo assim, Vincenzi et al. (2016) faz uma análise dos aspectos complementares das ferramentas de geração automática de dados de teste e o teste manual. Dentre as ferramentas analisadas, destacam-se a `EvoSuite`, `JTeXpert`, `Randoop` e a `Palus`.

A `EvoSuite`, proposta por Fraser e Arcuri (2011), é uma ferramenta de teste para a linguagem Java que usa técnicas de *SBST*, como algoritmos evolutivos e a aplicação da abordagem dinâmica *Symbolic Execution*. A `Palus` combina a abordagem dinâmica e estática para, num primeiro momento, criar um modelo de chamada de métodos e então avaliar quais métodos são dependentes entre si. A partir disso, essas informações servem de guia para uma geração randômica de dados de teste (Zhang, 2011). Já a ferramenta `Randoop` (Pacheco e Ernst, 2007) utiliza uma abordagem randômica na geração de dados de teste para métodos em Java. De acordo com Fraser et al. (2018), a `EvoSuite` tem se mostrado a ferramenta com maior desempenho, vencendo pelo menos as três últimas edições da *SBST Tool Competition*, uma competição anual que acontece no *The Search-Based Software Testing (SBST) Workshop*, por obter a maior pontuação dentre as outras ferramentas competidoras.

Um dos destaques dessa competição é o fato da `EvoSuite` ter vencido as demais ferramentas, na edição de 2018, obtendo médias de cobertura de ramos de 55,4% e de escore de mutação de 46,5%, o que são medidas muito baixas quando comparadas com o teste realizado de maneira manual. Outra ferramenta que também se destacou na *SBST Tool Competition* é a `JTeXpert`, ficando em segundo lugar na edição de 2017 (Sakti et al., 2017). O diferencial da `JTeXpert` é a presença de um construtor de teste candidatos, que analisa no código quais chamadas de métodos são relevantes para o teste, fazendo com que a ferramenta não gaste esforço desnecessário durante a geração de casos de teste.

3.6 Considerações Finais

Como pode ser observado neste capítulo, a atividade de teste e depuração otimizados vem resultando em boa parte dos trabalhos na literatura, como mostrado em Zhang et al. (2010).

Os estudos levantados no MS realizado neste trabalho, que foi uma atualização do MS realizado por Souza (2017), mostraram a utilização de técnicas como Programação Dinâmica, Colônia Artificial de Abelhas, Otimização de Busca dos Pinguins, mas com destaque maior para a técnica AG e suas variações, esta última representando a maior parte dos estudos realizados de 2017 até a data de hoje.

Os estudos levantados por McMinn (2004) apresentam estratégias de *SBST* aplicados à técnica de teste estrutural como o intuito de se saber quais são as características necessárias para se gerar dados de teste que exercitem os elementos cruciais do programa que está sendo testado. Com relação ao teste de mutação, Vincenzi et al. (2006) mostram que a principal causa do alto custo presente nesse critério está na dificuldade de automatizar o processo de geração de dados de teste capazes de matar mutantes específicos e também identificar aqueles que são equivalentes.

Apesar das diversas estratégias contarem com o auxílio de ferramentas, como a `EvoSuite`, alguns desafios com relação ao baixo escore de mutação e cobertura de código ainda persistem essas ferramentas são comparados à geração de dados de teste manual (Fraser et al., 2018).

Portanto, o teste de mutação, apesar de contar com o auxílio de ferramentas, ainda exige um grande esforço computacional e humano para ser realizado, sendo que um dos motivos é fato de ser difícil automatizar o processo de geração de dados de teste para atingir 100% de escore de mutação (Vincenzi et al., 2006).

Capítulo 4

EXPERIMENTO

4.1 Considerações Iniciais

Neste capítulo serão apresentados os passos adotados durante o experimento. Este capítulo está organizado da seguinte maneira: a Seção 4.2 aborda a definição do experimento, a Seção 4.3 o planejamento e a Seção 4.4 a operação do experimento.

4.2 Definição do Experimento

Neste trabalho foi utilizado o modelo “Goal-Question-Metric” (GQM) (Basili et al., 1994) para definir o estudo experimental, como está elencado a seguir:

- Objeto de Estudo: os conjuntos de teste gerados automaticamente.
- Propósitos:
    1. Avaliar individualmente os conjuntos de teste gerados por quatro ferramentas de geração automatizadas inteligentes e randômica.
    2. Avaliar os aspectos complementares entre os conjuntos de teste gerados por ferramentas de geração inteligentes combinadas e randômica.
    3. Avaliar os aspectos complementares entre os conjuntos de teste gerados por ferramentas de geração inteligentes e randômica combinadas.
- Foco de Qualidade: adequação, eficácia e o custo dos casos de teste, avaliados pelos critérios de cobertura de linhas de código, teste de mutação (Andrews et al., 2005, 2006) e também número de casos de teste presente no conjunto de teste.
- Perspectiva: ponto de vista dos pesquisadores.

- Contexto: os pesquisadores definiram o experimento considerando um conjunto de programas contido no domínio de estruturas de dados e então realizaram o estudo. O estudo envolve um participante (o primeiro autor) que trabalhou num conjunto de objetos, em outras palavras, esse é um estudo de variação multi-objeto.

4.3 Planejamento do Experimento

Nesta etapa de planejamento, as hipóteses e variáveis do estudo foram definidas. Logo após, foi criado o plano de experimento para guiar ambas a condução e análise dos dados obtidos. A seguir, é apresentado uma visão global das atividades principais que compõem o plano de experimento.

4.3.1 Seleção do Contexto

Como definido nos objetivos na Seção 1.2, o objetivo primário deste experimento é investigar os aspectos complementares entre os casos de teste gerados automaticamente por ferramentas inteligentes e uma randômica. Para isso foram investigados as métricas de cobertura de linhas de código, o escore de mutação e também o tamanho de cada conjunto de teste, como mencionado no foco de qualidade na Seção 4.2.

Neste experimento foram selecionados 33 programas Java, dos quais 32 são dos trabalhos de Souza et al. (2012) e Vincenzi et al. (2016), e adicionalmente foi incluído o programa “Identifier”, que é parte de um compilador. Estes programas, em geral, implementam estruturas de dados simples que são conhecidas na literatura (Ziviani, 2011).

Portanto, este experimento foi classificado como um estudo offline, realizado por um aluno de mestrado, que visa a identificação e comparação das métricas de adequação, eficácia e custo dos dados de teste gerados automaticamente num contexto particular.

4.3.2 Formulação das Hipóteses

O objetivo deste estudo é avaliar os aspectos complementares dos conjuntos de teste gerados automaticamente por ferramentas inteligentes e randômicas, buscando evidências que possam definir novas hipóteses para trabalhos futuros.

Com o intuito de definir as hipóteses deste trabalho, foram medidos: i) a adequação do conjunto de teste, considerando o critério de cobertura de código, ii) a eficácia, considerando o escore de mutação, e iii) o custo considerando o número de casos

de testes no conjunto de teste. É importante dizer que essas medidas podem ser influenciadas por: (i) ferramentas de teste utilizadas e (ii) tamanho e complexidade dos programas testados.

Desse modo, foram definidas as hipóteses a serem investigadas como pode ser visto na Tabela 4.1.

Tabela 4.1: Hipóteses formalizadas – Adequação, Eficácia e Custo

Hipótese Nula	**Hipótese Alternativa**
Não existe diferença de adequação entre os conjuntos de teste gerados por ferramentas inteligentes (*All Smart – AllS*) e aleatórias (*Random – R*). $H1_0$: $Adequacao(AllS) = Adequacao(R)$	Existe diferença de adequação entre os conjuntos de teste gerados por ferramentas inteligentes (*AllS*) e aleatórias (*R*). $H1_1 : Adequacao(AllS) \neq Adequacao(R)$
Não existe diferença de eficácia entre os conjuntos de teste gerados por ferramentas inteligentes (*AllS*) e aleatórias (*R*). $H2_0 : Eficacia(AllS) = Eficacia(R)$	Existe diferença de eficácia entre os conjuntos de teste gerados por ferramentas inteligentes (*AllS*) e aleatórias (*R*). $H2_1 : Eficacia(AllS) \neq Eficacia(R)$
Não existe diferença de custo entre os conjuntos de teste gerados por ferramentas inteligentes (*AllS*) e aleatórias (*R*). $H3_0 : Custo(AllS) = Custo(R)$	Existe diferente de custo entre os conjuntos de teste gerados por ferramentas inteligentes (*AllS*) e aleatórias (*R*). $H3_1 : Custo(AllS) \neq Custo(R)$

4.3.3 Seleção das Variáveis

Nesta etapa, foram definidas as variáveis independentes e dependentes do experimento, baseada no contexto e nas hipóteses formalizadas.

Variáveis Independentes. Variáveis que podem ser manipuladas ou controladas durante um experimento são chamadas de variáveis independentes (Wohlin et al., 2012). Neste trabalho, as variáveis independentes definidas foram: i) as ferramentas de teste adotadas; ii) os critérios de teste usados; iii) o tamanho e complexidade dos programas em teste e; iv) os conjuntos de teste. A última variável citada é o único fator de interesse neste experimento. Portanto, os conjuntos de teste são tratados de duas formas neste trabalho: gerados por ferramentas de automação inteligentes e aleatória. As outras variáveis do experimento foram ajustadas para não interferir com os resultados obtidos.

Variáveis Dependentes. Já as variáveis dependentes são aquelas que permitem observar os efeitos da manipulação das variáveis independentes (Wohlin et al., 2012). Neste trabalho, as variáveis dependentes definidas foram: i) a cobertura de código, que descreve a adequação dos conjuntos de teste; ii) o escore de mutação, que mede a eficácia, ou seja, a habilidade que os testes têm de detectar defeitos comumente conhecidos (mutantes) (Andrews et al., 2005) e; iii) a quantidade de casos de teste gerados, que descreve o custo do conjunto de teste.

Além das métricas citadas, outras também foram coletadas, como detalhado na Subseção 4.4.2.

4.3.4 Design do Experimento

A análise dos resultados é diretamente impactada pelo design do experimento, sendo um dos impactos a minimização da influência de fatores adversos nos resultados. Como estabelecido anteriormente, a qualidade dos conjuntos de teste é o único foco de interesse deste trabalho.

Na Figura 4.1 está ilustrada a estratégia de avaliação dos conjuntos de teste adotada neste trabalho. Para cada programa *Prog* foram gerados 40 conjuntos de teste, 10 para cada uma das ferramentas utilizadas: `EvoSuite` (E), `JTeXpert` (J), `Palus` (P) (Zhang, 2011) e `Randoop` (R).

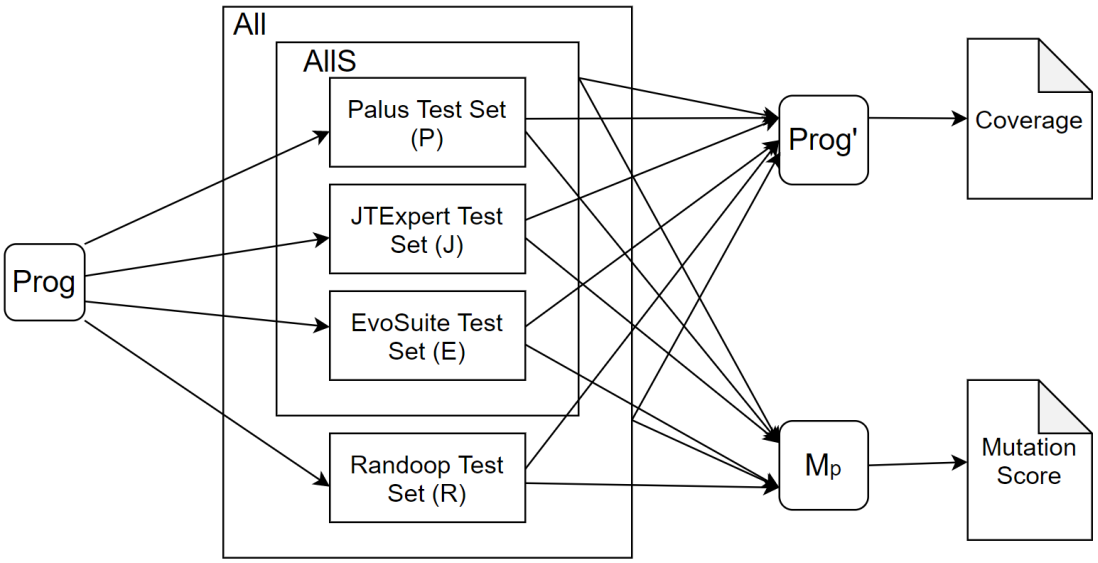

Figura 4.1: Estratégia de avaliação dos conjuntos de teste.

A abordagem seguida é similar a utilizada por Vincenzi et al. (2016), combinando conjuntos de teste para avaliar os aspectos complementares entre eles. Neste trabalho, além dos conjuntos de teste já citados, foram criados mais dois conjuntos oriundos de combinações dos conjuntos mencionados acima: *AllS* (abreviado do inglês, "All Smart", representa os conjuntos de teste de todos os geradores "inteligentes", ou seja, que usam heurísticas) e *All* (do inglês, "Todos", representando a união de todos os conjuntos de teste gerados). O primeiro, *AllS*, representa a união de todos os conjuntos de teste gerados pelas ferramentas que utilizam heurísticas na geração de casos de teste, ou seja, $(E \cup J \cup P)$. De modo semelhante, o outro conjunto, *All*, representa a união de todos os conjuntos de teste gerados neste experimento, ou seja, $(E \cup J \cup P \cup R)$.

Para gerar os conjuntos de teste, cada ferramenta gerou 10 repetições de geração dos conjuntos de teste utilizando dez sementes (do inglês, "seeds") diferentes e, então,

foram medidos a cobertura de código e o escore de mutação. Para isso, a ferramenta Pitest (PIT) (Coles, 2015) foi usada para calcular e gerar relatórios das métricas após cada execução dos conjuntos de teste.

Embora este experimento seja similar ao de Vincenzi et al. (2016), como já mencionado, no caso da ferramenta `Randoop`, nesta abordagem não foi fixado o máximo número de casos de testes gerados para cada programa de acordo o máximo número de casos de teste gerados pelas outras ferramentas. Ao invés disso, o tempo de geração padrão da `Randoop` foi diminuído pela metade, ou seja, para trinta segundos, uma vez que a ferramenta gera uma grande quantidade de casos de teste em seu tempo padrão.

4.4 Operação do Experimento

A operação do experimento inclui o preparo de artefatos e ferramentas, a execução das atividades definidas no planejamento (Seção 4.3) e a validação dos dados coletados durante a execução (Subseção 4.4.2).

4.4.1 Preparo

Como já mencionado na Subseção 4.3.1, foram utilizados 33 programas Java nos quais foram executados os conjuntos de teste gerados pelas ferramentas automatizadas e então foram realizadas as combinações dos conjuntos de teste como explicado na Subseção 4.3.4.

Antes do experimento ser executado, alguns ajustes foram realizados no conjunto de programas. Primeiramente, foi criado um projeto Maven para cada um dos programas usando o Eclipse (Foundation, 2015), uma vez que as ferramentas de geração automatizada podem ser chamadas através de scripts do Maven (Foundation, 2016). Em seguida, foi realizada uma padronização dos nomes dos conjuntos de teste com o intuito de relacioná-los com a ferramenta que os geraram. Esses ajustes possibilitaram uma facilidade de execução dos scripts em Python, que serão mencionados no parágrafo seguinte.

Sendo assim, foram escritos scripts em Python para chamar o Maven com o propósito de: i) compilar os programas e conjuntos de teste, ii) executar os conjuntos de teste, iii) calcular a cobertura de código e o escore de mutação e iv) gerar os relatórios de teste. Na Tabela 4.2 estão apresentadas as ferramentas e suas respectivas versões usadas, como também seus propósitos neste experimento.

Tabela 4.2: Versão e propósito das ferramentas

Ferramenta	Versão	Propósito
JavaNCSS	32.53	Computação de Métricas Estáticas
EvoSuite	1.0.6	Geração de teste
JTeXpert	1.4	Geração de teste
Palus	0.20	Geração de teste
Randoop	4.2.4	Geração de teste
Pitest	1.3.2	Teste de cobertura e mutação
Eclipse	4.14.0	Ambiente de desenvolvimento integrado
Maven	3.6.3	Builder da aplicação
JUnit	4.12	Framework para teste unitário
Python	2.7.18	Linguagem do script

Para a realização do experimento, for utilizado um laptop com as seguintes especificações:

- Modelo: Dell Inspiron;
- SO: Linux Ubuntu 20.04 LTS 64 bits;
- Processador: Core i7;
- RAM: 32 GB;
- Disco rígido: 1 TB.

4.4.2 Execução

O primeiro passo para a execução dos programas é o registro das métricas de cada um deles, como está apresentado na Tabela 5.1. Para cada programa, as seguintes métricas foram catalogadas:

- Linhas de código, descartando os comentários (NCSS);
- Número de Complexidade Ciclomática (CCN);
- Média da Complexidade Ciclomática (CCA);
- Quantidade de casos de teste em cada conjunto de teste: `EvoSuite` (E), `JTeXpert` (J), `Palus` (P) e `Randoop` (R);
- Quantidade de requisitos demandados para a cobertura de código;
- Quantidade de mutantes gerados considerando todos os operadores de mutação disponíveis na ferramenta PIT.

Uma informação importante é que as ferramentas de geração foram executadas utilizando suas configurações padrão, com exceção da `Randoop`, como já mencionado na Subseção 4.3.4.

4.4.3 Validação dos Dados

Para ter garantia de que os dados esperados serão de fato obtidos dos scripts, todos os geradores de teste automáticos foram executados manualmente nos programas. Em um segundo momento, os dados obtidos pelos scripts foram comparados com os obtidos manualmente. Por fim, nenhuma diferença foi encontrada ao comparar os dados, o que aumentou a confiança nos scripts e então a coleta de dados foi iniciada para os 33 programas.

4.5 Considerações Finais

Neste capítulo foram abordados as etapas seguidas durante o experimento. Com base no modelo GQM, foi possível então guiar o experimento através da definições: i) do objeto de estudo, que são os quatro geradores automáticos; ii) das hipóteses, que questionam os aspectos complementares entre os geradores com base na adequação, eficácia e custo a nível de unidade; e iii) do contexto, que envolve os programas Java. A partir disso, foi descrita a operação do experimento, detalhando as etapas de execução das ferramentas automáticas e, consequentemente, a coleta dos dados gerados.

Capítulo 5

DADOS E DISCUSSÃO

5.1 Considerações Iniciais

Neste capítulo serão discutidos os dados coletados durante o experimento. Este capítulo está organizado da seguinte maneira: a Seção 5.2 aborda a análise dos dados coletados e a Seção 5.3 trata das ameças à validade deste experimento.

5.2 Análise dos Dados

Inicialmente, é necessário caracterizar os programas utilizados neste experimento. Na Tabela 5.1 pode ser visto que trata-se de programas de fácil compreensão, os quais implementam de 1 a 3 classes (1,5 em média), 6,1 métodos em média, somando em torno de 40,2 linha de código (LOC). A coluna CCM corresponde à complexidade ciclomática máxima encontrada nos métodos, enquanto a coluna CCA corresponde à complexidade ciclomática média de cada um dos programas. A CCM varia de 2 para 9, atingindo média de 4,9. Já a CCA varia de 1,3 até 9, obtendo em média 3,3 nos programas. As duas últimas colunas da Tabela 5.1 apresentam o número de requisitos demandados pelos critérios de cobertura de código (#Req) e de teste de mutação (#Mut). No caso do teste de mutantes, foram usados todos os operadores de mutação implementados na PIT, como já mencionado na Subseção 4.4.2.

Na Tabela 5.2 estão presentes os dados coletados sobre a cobertura de código e o escore de mutação por cada conjunto de teste. Considerando os conjuntos de teste individualmente, do melhor para o pior em termos de cobertura de código, tem-se J, E, R e P. Com relação ao escore de mutação, R atingiu o maior valor, seguido por J, E e P.

Tabela 5.1: Informação dos programas Java

ID	Program	#Classes	#Methods	NCSS	CCM	CCA	#Req	#Mut
1	Max	1	1	8	3	3,0	4	14
2	MaxMin1	1	1	13	4	4,0	8	21
3	MaxMin2	1	1	14	4	4,0	8	21
4	MaxMin3	1	1	32	9	9,0	16	61
5	Sort1	1	1	11	4	4,0	10	21
6	FibRec	1	1	8	2	2,0	6	12
7	FibIte	1	1	8	2	2,0	6	12
8	MaxMinRec	1	1	26	5	5,0	13	41
9	Mergesort	1	2	22	6	4,0	16	56
10	MultMatrixCost	1	1	18	6	6,0	14	75
11	ListArray	1	4	20	3	1,8	12	29
12	ListAutoRef	2	4	23	2	1,3	12	21
13	StackArray	1	5	20	3	1,8	12	27
14	StackAutoRef	2	5	27	3	1,4	17	27
15	QueueArray	1	5	24	3	2,0	19	40
16	QueueAutoRef	2	5	32	3	1,6	23	32
17	Sort2	2	7	74	6	3,4	49	141
18	HeapSort	1	9	59	5	2,7	40	116
19	PartialSorting	1	10	62	5	2,5	42	120
20	BinarySearch	1	4	32	8	3,5	21	55
21	BinaryTree	2	11	85	7	3,0	48	145
22	Hashing1	2	10	61	5	2,1	35	88
23	Hashing2	2	12	88	7	3,2	51	162
24	GraphMatAdj	1	9	60	5	2,9	42	134
25	GraphListAdj1	3	16	66	4	1,6	34	95
26	GraphListAdj2	2	14	88	6	2,2	51	113
27	DepthFirstSearch	3	16	65	4	1,6	33	94
28	BreadthFirstSearch	3	16	65	4	1,6	33	94
29	Graph	3	16	65	4	1,6	33	94
30	PrimAlg	1	5	40	7	2,6	31	71
31	ExactMatch	1	4	55	8	6,3	40	205
32	AproximateMatch	1	1	24	7	7,0	19	88
33	Identifier	1	3	30	9	7,7	22	114
Média (M)		1,5	6,1	40,2	4,9	3,3	24,8	73,9
Desvio Padrão (DP)		0,7	5,2	25,4	2,0	2,0	14,7	50,3

Ao observar o padrão, *E* possui mais testes que atingem 100% de cobertura de código, seguido por *J*, *R* e *P*. Apenas nos programas 9, 10, 24 e 31 as demais ferramentas de geração automatizada atingiram melhores resultados se tratando da cobertura de código. Com exceção do programa 9, os demais mencionados estão entre os programas mais complexos e os geradores tiveram desempenho superior que a *E*. Com relação ao escore de mutação, *E* não atingiu 100% em nenhum programa, enquanto que os demais conjuntos atingiram 100% em dois programas (6 e 7).

Como pode ser observado na Tabela 5.3, todas as ferramentas geraram, em média, pelo menos um caso de teste para cada programa. Não obstante, devido a alguma incompatibilidade, *E* falha ao obter cobertura de código e escore de mutação para o programa 10. O mesmo acontece com o conjunto de teste da `Palus` (*P*) no programa 30. *P* também falha ao obter o escore de mutação para os programas 1 e 10, como

mostrado na Tabela 5.2. Até o momento, não foi identificado o motivo para estas falhas citadas acima. Futuramente, será realizada uma investigação deste fato com o intuito de melhorar estas ferramentas.

Tabela 5.2: Médias de Cobertura de Código e Escore de Mutação por Conjunto de Teste

ID	Cobertura por Conjunto de Teste (%)						Escore de Mutação por Conjunto de Teste (%)					
	E	*P*	*J*	*R*	*AllS*	*All*	*E*	*P*	*J*	*R*	*AllS*	*All*
1	100.0	75.0	100.0	100.0	100.0	100.0	44.3	0.0	27.9	50.0	49.3	64.3
2	100.0	98.8	100.0	100.0	100.0	100.0	29.0	28.1	83.8	52.4	83.8	83.8
3	100.0	96.3	100.0	100.0	100.0	100.0	29.0	26.7	84.3	52.4	84.3	84.3
4	100.0	68.8	100.0	93.8	100.0	100.0	21.3	10.6	79.7	35.5	79.7	79.8
5	100.0	61.0	100.0	70.0	100.0	100.0	56.0	28.0	50.5	30.0	77.5	78.5
6	100.0	100.0	100.0	100.0	100.0	100.0	92.5	100.0	100.0	100.0	100.0	100.0
7	100.0	100.0	100.0	100.0	100.0	100.0	92.5	100.0	100.0	100.0	100.0	100.0
8	100.0	96.2	100.0	100.0	100.0	100.0	23.1	26.4	76.0	48.1	82.4	83.1
9	91.9	95.0	100.0	100.0	100.0	100.0	19.8	32.5	52.9	95.5	64.5	95.5
10	0.0	92.9	100.0	92.9	100.0	100.0	0.0	26.6	34.9	30.4	45.3	45.6
11	100.0	91.7	91.7	98.3	100.0	100.0	68.1	61.3	57.4	76.8	73.5	78.1
12	100.0	100.0	92.5	100.0	100.0	100.0	69.6	73.9	70.0	82.6	74.8	83.9
13	100.0	91.7	91.7	96.7	100.0	100.0	63.5	77.1	67.4	77.4	81.3	81.3
14	100.0	100.0	100.0	100.0	100.0	100.0	55.2	81.6	69.7	83.9	84.2	85.5
15	100.0	94.7	93.2	99.5	100.0	100.0	79.8	72.1	65.6	82.8	87.9	90.5
16	100.0	100.0	83.5	100.0	100.0	100.0	66.2	67.6	53.0	73.0	78.1	82.4
17	100.0	59.8	100.0	66.7	100.0	100.0	44.6	21.7	52.4	24.5	68.2	68.8
18	100.0	57.5	81.0	73.8	100.0	100.0	67.1	27.6	31.0	31.0	73.5	73.9
19	100.0	57.6	100.0	73.3	100.0	100.0	75.7	23.1	54.3	30.7	84.4	84.4
20	100.0	23.8	60.5	76.2	100.0	100.0	64.7	0.0	23.0	33.7	65.4	70.4
21	88.8	12.5	81.3	54.2	88.8	88.8	93.8	1.3	38.6	17.5	93.8	93.8
22	100.0	99.1	96.9	100.0	100.0	100.0	65.1	71.3	48.2	82.5	84.3	93.9
23	100.0	96.0	94.6	98.1	100.0	100.0	61.2	51.0	48.1	79.6	74.8	86.6
24	19.0	86.0	89.5	96.2	89.5	96.2	7.0	51.5	44.2	68.4	53.1	69.0
25	100.0	95.9	98.8	100.0	100.0	100.0	59.3	58.1	57.8	76.8	74.5	81.9
26	98.4	90.2	93.3	95.3	99.0	99.2	60.1	55.7	54.7	68.3	74.6	78.1
27	100.0	96.4	100.0	100.0	100.0	100.0	64.4	54.8	56.5	76.7	76.7	81.0
28	100.0	93.3	100.0	100.0	100.0	100.0	64.4	55.4	53.3	76.7	76.3	82.1
29	100.0	93.9	100.0	100.0	100.0	100.0	64.4	55.3	59.2	75.6	76.2	81.4
30	100.0	0.0	71.9	28.1	100.0	100.0	30.3	0.0	31.8	1.3	42.4	42.4
31	90.0	100.0	100.0	100.0	100.0	100.0	31.3	55.3	34.0	57.6	56.8	58.6
32	100.0	100.0	100.0	100.0	100.0	100.0	44.2	39.5	29.1	39.5	51.6	51.6
33	100.0	95.9	100.0	90.9	100.0	100.0	46.3	64.6	41.8	51.8	75.1	75.8
M	93.6	82.4	94.6	91.0	99.3	99.5	53.1	45.4	55.5	59.5	74.2	78.2
DP	21.7	26.0	9.2	16.3	2.6	2.0	23.5	27.5	19.5	25.3	14.1	14.1

Na Tabela 5.3 são apresentados os dados em relação ao número de conjuntos de teste gerados. A coluna “S” apresenta os casos de teste gerados com sucesso, coluna “I” os casos de teste que foram ignorados, a coluna “F” os casos de teste que tiveram falha na geração e a coluna “To” os caso de testes gerados no total. Como esperado, *R* teve a maior quantidade de casos de teste, em média, seguido por *P*, *J* e *E*. É observado que *R* gerou o máximo número de casos de teste, 3283,7, para o programa 32 e o mínimo de 34 casos de teste para o programa 5.

Tabela 5.3: Média do Número de Testes Gerados por Gerador

	Evo				Palus				JTExpert				Randoop				AllS			
ID	S	I	F	To	S	I	F	To	S	I	F	To	S	I	F	To	S	I	F	To
1	3,3	0,0	0,0	3,3	6,0	0,0	0,0	6,0	3,0	0,0	0,0	3,0	2198,9	0,0	0,0	2198,9	12,3	0,0	0,0	12,3
2	4,0	0,0	0,0	4,0	21,7	0,0	0,0	21,7	2,7	0,0	0,0	2,7	1404,8	0,0	0,0	1404,8	28,4	0,0	0,0	28,4
3	4,0	0,0	0,0	4,0	28,8	0,0	0,0	28,8	3,9	0,0	0,0	3,9	1308,2	0,0	0,0	1308,2	36,7	0,0	0,0	36,7
4	7,4	0,0	0,0	7,4	40,0	0,0	0,0	40,0	8,2	0,0	0,0	8,2	1481,6	0,0	0,0	1481,6	55,6	0,0	0,0	55,6
5	2,4	0,0	0,0	2,4	29,8	0,0	0,0	29,8	3,0	0,0	0,0	3,0	1947,2	0,0	0,0	1947,2	35,2	0,0	0,0	35,2
6	2,0	0,0	0,0	2,0	24,8	0,0	0,0	24,8	1,6	0,0	0,0	1,6	34,0	0,0	0,0	34,0	28,4	0,0	0,0	28,4
7	2,0	0,0	0,0	2,0	28,5	0,0	0,0	28,5	1,4	0,0	0,0	1,4	34,0	0,0	0,0	34,0	31,9	0,0	0,0	31,9
8	3,0	0,0	0,0	3,0	29,5	0,0	0,0	29,5	4,7	0,0	0,0	4,7	2127,5	0,0	0,0	2127,5	37,2	0,0	0,0	37,2
9	2,1	0,0	0,0	2,1	16,8	0,0	0,0	16,8	2,8	0,0	0,0	2,8	1846,3	0,0	0,0	1846,3	21,7	0,0	0,0	21,7
10	2,0	0,0	0,0	2,0	28,9	0,0	0,0	28,9	3,2	0,0	0,0	3,2	1902,7	0,0	0,0	1902,7	34,1	0,0	0,0	34,1
11	4,4	0,0	0,0	4,4	59,9	0,0	0,0	59,9	2,9	0,0	0,2	3,1	853,5	0,0	0,0	853,5	67,2	0,0	0,2	67,4
12	3,0	0,0	0,0	3,0	68,2	0,0	0,0	68,2	2,7	0,0	0,1	2,8	668,4	0,0	0,0	668,4	73,9	0,0	0,1	74,0
13	3,9	0,0	0,0	3,9	77,1	0,0	0,0	77,1	4,0	0,0	0,1	4,1	768,1	0,0	0,0	768,1	85,0	0,0	0,1	85,1
14	3,0	0,0	0,0	3,0	72,9	0,0	0,0	72,9	2,9	0,2	0,4	3,5	677,0	0,0	0,0	677,0	78,8	0,2	0,4	79,4
15	4,1	0,0	0,4	4,5	67,3	0,0	0,0	67,3	4,2	0,0	0,1	4,3	374,6	0,0	0,0	374,6	75,6	0,0	0,5	76,1
16	2,9	0,0	0,2	3,1	78,6	0,0	0,0	78,6	3,9	0,0	0,1	4,0	382,6	0,0	0,0	382,6	85,4	0,0	0,3	85,7
17	11,9	0,0	0,0	11,9	38,0	0,0	0,0	38,0	16,1	0,0	0,0	16,1	2379,1	0,0	0,0	2379,1	66,0	0,0	0,0	66,0
18	10,1	0,0	0,3	10,4	136,6	0,0	0,0	136,6	10,3	0,1	0,0	10,4	564,7	0,0	0,0	564,7	157,0	0,1	0,3	157,4
19	11,8	0,0	0,1	11,9	95,5	0,0	0,0	95,5	12,9	0,0	0,1	13,0	370,9	0,0	0,0	370,9	120,2	0,0	0,2	120,4
20	5,4	0,0	0,0	5,4	1,0	0,0	0,0	1,0	2,6	0,1	0,1	2,8	573,2	0,0	0,0	573,2	9,0	0,1	0,1	9,2
21	11,1	0,0	1,8	12,9	1,4	0,0	0,0	1,4	12,2	0,1	0,0	12,3	1565,7	0,0	0,0	1565,7	24,7	0,1	1,8	26,6
22	4,2	0,0	0,8	5,0	96,4	0,0	0,0	96,4	6,5	0,2	0,3	7,0	562,9	12,9	0,5	576,3	106,8	0,2	1,4	108,4
23	7,9	0,0	0,3	8,2	83,3	0,0	0,0	83,3	9,6	0,1	0,3	10,0	378,2	1,9	0,1	380,2	100,8	0,1	0,6	101,5
24	1,0	0,0	0,0	1,0	35,5	0,0	0,0	35,5	8,4	0,1	0,1	8,6	406,4	0,0	0,0	406,4	44,9	0,1	0,1	45,1
25	8,8	0,0	0,0	8,8	83,5	0,0	0,0	83,5	14,9	0,1	0,0	15,0	1196,5	0,0	0,0	1196,5	107,2	0,1	0,0	107,3
26	9,5	0,7	0,0	10,2	79,7	0,0	0,0	79,7	13,2	0,0	0,2	13,4	369,9	0,0	0,0	369,9	102,3	0,7	0,3	103,3
27	8,9	0,0	0,0	8,9	111,8	0,0	0,0	111,8	13,9	0,0	0,0	13,9	1230,0	0,0	0,0	1230,0	134,6	0,0	0,0	134,6
28	8,9	0,0	0,0	8,9	121,3	0,0	0,0	121,3	14,0	0,0	0,2	14,2	1135,7	0,0	0,0	1135,7	144,2	0,0	0,2	144,4
29	8,9	0,0	0,0	8,9	128,2	0,0	0,0	128,2	14,0	0,0	0,0	14,0	974,6	0,0	0,0	974,6	151,1	0,0	0,0	151,1
30	4,1	0,0	0,0	4,1	1,0	0,0	0,0	1,0	2,8	0,0	0,0	2,8	60,0	0,0	0,0	60,0	7,9	0,0	0,0	7,9
31	11,0	0,0	0,0	11,0	313,4	0,0	0,0	313,4	15,4	0,0	0,0	15,4	3163,0	0,0	0,0	3163,0	339,8	0,0	0,0	339,8
32	3,6	0,0	0,0	3,6	316,4	0,0	0,0	316,4	3,7	0,0	0,0	3,7	3283,7	0,0	0,0	3283,7	323,7	0,0	0,0	323,7
33	6,5	0,0	0,0	6,5	102,8	0,0	0,0	102,8	5,1	0,0	0,0	5,1	1125,1	0,0	0,0	1125,1	114,4	0,0	0,0	114,4
M	5,7	0,0	0,1	5,8	73,5	0,0	0,0	73,5	7,0	0,0	0,1	7,1	1131,8	0,4	0,0	1132,3	86,1	0,1	0,2	86,4
DP	3,4	0,1	0,3	3,5	73,3	0,0	0,0	73,3	4,9	0,1	0,1	4,9	851,5	2,3	0,1	851,2	76,6	0,1	0,4	76,6

Na Tabela 5.3, a ferramenta *P* gerou 316,4 testes em média no caso máximo e 1 teste em média no caso mínimo. *J* gerou, em média, 16,1 e 1,6 testes no máximo e mínimo, respectivamente. Por fim, *E* gerou 12,9 e 1 testes nos casos máximo e mínimo, respectivamente.

Considerando as três hipóteses estabelecidas, baseadas na cobertura de código, escore de mutação (Tabela 5.2) e quantidade de conjuntos de teste gerada (Tabela 5.3), foi aplicado a norma de Shapiro-Wilk (Devore, 2019; Othman et al., 2015). Após essa aplicação, foi concluído que os dados coletados não possuem uma distribuição normalizada com um nível de confiança de 95%, ou seja, valor-p $\leq$ 0,05. Portanto, foi utilizado um teste não-paramétrico, o teste U de Mann-Whitney (Devore, 2019), para verificar a diferença entre os grupos dos conjuntos de teste gerados pelas ferramentas inteligentes e aleatória, considerando um nível de confiança de 95% ($\alpha = 0,05$). Nas Tabelas 5.4 até 5.6 estão apresentados os resultados dessas normalizações para cada par de conjuntos de teste. Uma vez que múltiplas hipóteses, foi aplicado o método de correção Holm-Bonferroni que resultou na coluna "valor-p corrigido" (Devore, 2019; Othman et al., 2015).

5.2.1 Análise da Adequação

Em média, *J* obteve uma cobertura de 94,6% com um desvio padrão (*DP*) de 9,2%. *E* obteve uma cobertura de 93,6% e *DP* de 21,7%. *R* obteve 91% de cobertura e 16,3 de *DP*; e *P* obteve cobertura de 82,4% e *DP* de 26%. Entretanto, existem situações específicas nas quais um conjunto de teste tem melhor performance que o outro, o que pode ser observado nas células que estão coloridas de cinza na Tabela 5.2.

Na Tabela 5.4 está apresentado o teste de Wilcoxon aplicado no critério de cobertura de código dos conjuntos de teste gerados pelas ferramentas, comparando os testes gerados de forma randômica e inteligente, individualmente e combinado, para todos os 33 programas.

Tabela 5.4: Adequação: cobertura de código

Par de Conjuntos	Valor-p	Valor-p Corrigido
R-E	0.1386833	0.2773666
R-P	4.299318e-05	0.0001719727
R-J	0.3603752	0.3603752
R-AllS	0.001757725	0.005273174

A estatística sugere que, em se tratando da comparação dos conjuntos de teste *R* com *E* ($R - E$) e dos conjuntos de teste de *R* com *J* ($R - J$), não há diferença estatística em termos de adequação porque o valor-p corrigido está acima de 0,05. Por isso, a hipótese nula $H1_0$ deve ser aceita. Já em relação a comparação $R - P$, existe diferença estatística porque o valor-p corrigido está abaixo de 0,05, indicando que a

hipótese alternativa $H1_1$ deve ser aceita, e o conjunto de teste *R* se mostra com melhor adequação em relação a *P*.

5.2.2 Análise da Eficácia

A mesma comparação foi feita com o escore de mutação obtido pelos conjuntos de teste, como mostrado na Tabela 5.5. De forma semelhante à comparação anterior, foi observado que apenas na comparação de *R* com *P* é evidenciado que há diferença estatística significante, resultando no aceite da hipótese alternativa $H2_1$. Com relação aos demais conjuntos de teste, o valor-p corrigido de *E* e *J* está acima de 0,05, sugerindo o aceite da hipótese nula $H2_0$ pois não existe diferença estatística significativa em relação à eficácia. Novamente, em relação à eficácia, o conjunto de teste *R* se mostra com melhor eficácia do que o conjunto *P*. O escore médio do conjunto *R* foi de 59,5 contra 45,4 de escore médio do conjunto *P*.

Tabela 5.5: Eficácia: escore de mutação

Par de Conjuntos	Valor-p	Valor-p Corrigido
R-E	0.1077912	0.2155824
R-P	8.069576e-06	3.22783e-05
R-J	0.5371801	0.5371801
R-AllS	0.002673522	0.008020565

5.2.3 Análise do Custo

Com relação à análise estatística do custo, como esperado, *R* obteve, em média, 1132,3 casos de teste, que é uma quantidade consideravelmente elevada quando comparada aos demais conjuntos de testes, mencionados a seguir. *P* gerou 73,5 casos de teste em média, *J* gerou 7,1 e *E* gerou 5,8. Particularmente para *E* e *J*, a quantidade de casos de testes gerados foi considerada razoável no caso de um testador desejar verificar se os testes gerados estão corretos. Apesar do fato de *P* possuir mais de 10 vezes a quantidade de testes que *J*, ainda assim *P* é consideravelmente menor que *R*, uma vez que este possui em média mais de 13 vezes a quantidade de testes gerados que *P*. Além disso, o desvio padrão obtido por *E* e *J* são menores que os obtidos por *P* e *R*.

Por fim, é mostrado na Tabela 5.6 o teste de Wilcoxon na quantidade de casos de teste gerada pelas ferramentas inteligentes e randômica para todos os 33 programas. Pode ser observado que em todas as comparações realizadas, o valor-p corrigido está consideravelmente abaixo de 0,05, indicando que a hipótese alternativa $H3_1$ deve ser aceita. Esse resultado obtido não é surpreendente porque, como já mencionado anteriormente, não foi limitada a quantidade de casos de teste durante a geração pela ferramenta `Randoop` de acordo com a quantidade de casos de teste gerada pelas

demais ferramentas inteligentes. Nesse sentido, os conjuntos *E*, *J*, e *P* são de menor custo quando comparado ao conjunto *R*.

Tabela 5.6: Custo: número de casos de teste

Par de Conjuntos	Valor-p	Valor-p Corrigido
R-E	5.6418e-07	5.6418e-07
R-P	2.328306e-10	9.313226e-10
R-J	2.328306e-10	9.313226e-10
R-AllS	2.328306e-10	9.313226e-10

5.2.4 Aspectos Complementares dos Conjuntos de Teste

Depois de realizada a análise dos conjuntos de teste gerados pela ferramenta `Randoop` em relação aos demais, observando tanto as estatísticas quanto resultados individuais dos testes, foram realizadas combinações incrementais desses conjuntos de teste analisados com o intuito de observar se existe um aspecto complementar entre os conjuntos de modo a verificar o quanto eles podem melhorar a adequação e eficácia. Para isso, foram criados mais dois conjuntos de teste: i) *AllS* (abreviado do inglês *All Smart*, todas as inteligentes), representando a combinação dos conjuntos de teste *E*, *J* e *P*; e ii) *All* ("todas"), representando todos os conjuntos de teste, ou seja, adicionando *R* à combinação anterior *AllS*.

Conjunto AllS

Pode ser visto na Tabela 5.2 que o conjunto *AllS*, se tratando da métrica de cobertura de código, obteve 3 de 33 programas abaixo de 100% e atingiu uma média de 99,3%, superando *R*, além de ter um desvio padrão menor. Também, ao observar o respectivo valor-p corrigido na Tabela 5.4, pode ser observado que a respectiva comparação se encontra abaixo de 0,05, indicando que a hipótese alternativa $H1_1$ deve ser aceita, mas agora a favor do *AllS*.

Com relação ao escore de mutação, pode ser observado na Tabela 5.2 que *AllS* atingiu uma média de 100% apenas em 2 dos 33 programas, o que também foi obtido pelos conjuntos *P*, *J* e *R*. Por outro lado, o conjunto *AllS* obteve uma média geral de 74,2%, que é consideravelmente maior que a média obtida por *R*, além de *AllS* ter obtido um desvio padrão menor. Além disso, pode ser observado na Tabela 5.5 que o valor-p corrigido está abaixo de 0,05, indicando que existe diferença estatística significante e, então, a hipótese alternativa $H2_1$ deve ser aceita, também a favor do *AllS*.

Sobre a quantidade de casos de teste gerados, o conjunto *AllS* gerou 86,4 testes em média enquanto o conjunto *R* gerou 1132,3, mais de 13 vezes maior, como pode ser observado na Tabela 5.3. Consequentemente, na Tabela 5.6 pode ser visto que o

respectivo valor-p corrigido está abaixo de 0,05, indicando que a hipótese alternativa $H3_1$ deve ser aceita, ou seja, existe diferença estatística significante entre os conjuntos *R* e *AllS*, com vantagem para o *AllS*.

Portanto, com relação à adequação, eficácia e custo, pode ser concluído com base nas Tabelas 5.4 até 5.6 que existe uma melhora significativa do conjunto *AllS* em relação à *R*, indicando que as todas as hipóteses alternativas devem ser aceitas, a favor do conjunto *AllS*.

Conjunto All

Como já mencionado, um último passo de combinação foi realizado, que foi a união de todos os conjuntos analisados previamente, formando o conjunto *All*. Na Tabela 5.2 pode ser visto que apenas 3 dos 33 programas estão abaixo de 100% em relação à cobertura de código, o que também já foi observado no conjunto *AllS*, embora *All* tenha obtido uma maior cobertura em 2 desses programas (24 e 26). Além disso, *All* atingiu uma média de 99,5% de cobertura de código, um aumento pouco expressivo quando comparado com *AllS*. Em relação ao escore de mutação, o conjunto *All* também teve 2 dos 33 programas que atingiram 100%, o que foi observado em todos os demais conjuntos com exceção de *E*, mas a média total foi ligeiramente superior à *AllS*.

Por fim, ao analisar a quantidade de casos de testes de *All*, pode ser evidenciado na Tabela 5.3 que ao adicionar os casos de teste de *R* no conjunto *AllS* tem-se uma quantidade ainda maior de testes no total.

5.3 Ameaças à Validade

É crucial estar atento às potenciais ameaças à validade de uma pesquisa. A seguir estão apontadas as ameaças internas, externas, de construção e conclusão deste trabalho.

Interna. Foi identificada que a etapa de Seleção de Contexto (Seção 4.3.1) como uma potencial ameaça uma vez que não foi realizada uma seleção randômica do conjunto de programas, apesar dos mesmos serem tradicionalmente usados na literatura. Uma outra ameaça à validade interna identificada diz respeito às ferramentas utilizadas, uma vez que as mesmas podem conter defeitos e consequentemente levar a resultados incorretos. Para minimizar a ameaça, as ferramentas de geração foram executadas dez vezes com diferentes sementes para cada programa.

Externa. neste foram utilizadas as ferramentas `EvoSuite`, `Randoop`, `JTeXpert` e `Palus` para a geração automática de dados de teste. É possível que usando diferentes ferramentas ocasionem em diferentes resultados. Apesar disso, as ferramentas utilizadas são consideradas o estado da arte para código Java.

Construção. neste trabalho foi aplicada mutação à nível de unidade como um modelo de defeitos para medir a eficácia. Embora Andrews et al. (2005) afirma a relação entre mutação e defeitos reais, usando diferentes ferramentas de mutação podem levar a diferentes resultados quanto à eficácia.

Conclusão. o conjunto de programas deste trabalho, composto por 33 programas simples em Java, pode ser relativamente pequeno, o que pode levar a diferentes resultados se for aumentado tanto a quantidade de programas e o tamanho dos mesmos. Apesar disso, o foco deste trabalho são testes unitários, e os métodos dos programas selecionados possuem complexidade suficiente, representativa de uma grande variedade de aplicações Java em nível de método.

5.4 Considerações Finais

Este capítulo trata da análise dos dados coletados neste experimento. Num primeiro momento, é observado que, com relação aos conjuntos de teste individualmente analisados, não há comprovação estatística quanto a superioridade dos conjuntos gerados pelas ferramentas inteligentes em relação ao conjunto de teste gerado randomicamente (R) quanto à adequação e eficácia. Já em relação ao custo, os conjuntos inteligentes se mostram ser consideravelmente menos custosas que R.

Com relação aos conjunto gerados pelas combinações dos conjuntos inteligentes ($AllS$) e também randômico (All), é observado que $AllS$ supera estatisticamente o conjunto R em todas as métricas. Apesar disso, é importante notar que os 74,2% de mutantes mortos pelo conjunto $AllS$, corresponde a uma quantidade de 629 mutantes que ainda permanecem vivos (seja pelo fato de não serem alcançados pelas ferramentas ou serem estes equivalentes). Já o conjunto All se mostrou um aumento considerável de aumento de custo em troca de uma ligeira melhora nas métricas de adequação e eficácia em relação à $AllS$.

Tais aspectos evidenciam a complementariedade das ferramentas de teste, além da necessidade de evolução das mesmas visando a maximizar a detecção de defeitos modelados por operadores de mutação.

Capítulo 6

CONCLUSÃO

6.1 Considerações Iniciais

Neste capítulo será apresentada a conclusão deste trabalho. Este capítulo está organizado da seguinte maneira: a Seção 6.2 apresenta as definições do experimento e, por fim, a Seção 6.3 apresenta os trabalhos futuros.

6.2 Contribuições do Trabalho

Neste trabalho foi realizado uma investigação sobre os aspectos complementares entre conjuntos de teste gerados por ferramentas inteligentes e randômicas em 33 programas que implementam estruturas de dados. Para isso, foram utilizados quatro geradores automáticos de dados de teste com suas configurações padrão, exceto pela ferramenta `Randoop` na qual foi reduzido pela metade o tempo de geração por causa do número elevado de dados de teste que a ferramenta gera.

Ao fazer uma avaliação, num primeiro momento, de cada conjunto de testes gerado pelas ferramentas com relação à adequação, o conjunto da `EvoSuite` (*E*) obteve tanto a maior média de cobertura de código quanto a maior quantidade de programas com 100% de cobertura em média, seguido pelo conjunto da `JTeXpert` (*J*). O conjunto da `Randoop` (*R*) obteve maior média de cobertura que o conjunto da `Palus` (*P*). Embora *E* tenha sido destacado acima, a análise estatística sugere que não há diferença de adequação entre *R*, *E* e *J*.

Com relação à eficácia, *R* obteve o maior escore de mutação na maioria dos programas, tendo uma média de 59,5%. Os outros conjuntos atingiram um maior escore de mutação em 15 programas, sendo *J* o segundo melhor conjunto de teste por ter obtido uma média de 55,5% de cobertura. Entretanto, a análise estatística mostra

que não há diferença significante entre *R-E* e *R-J*, o que é perceptível também ao observar que os respectivos escores de mutação são muito próximos um do outro.

Com relação ao custo dos conjuntos de teste gerados pelas ferramentas inteligentes e randômica, foi a única métrica na qual a análise estatística sugeriu que a hipótese alternativa $H3_1$ deve ser aceita para todos os casos de comparação. A razão disso é que *R* gera uma grande quantidade de casos de teste mesmo quando o tempo de geração da ferramenta `Randoop` é decrementado pela metade.

Num segundo momento, foi realizada uma investigação sobre os aspectos complementares dos mesmos conjuntos já citados anteriormente, mas agora combinados. A ideia foi combinar todos os conjuntos de teste gerados pelas ferramentas que usam alguma estratégia na geração baseada em heurísticas (denominado como *AllS*), e então comparar com o conjunto gerado pela ferramenta randômica, `Randoop`. Feita a comparação, foi concluído que *AllS* obteve resultados significativamente superiores que *R* em todos os aspectos investigados: adequação, eficácia e custo.

Por fim, foi realizada uma última investigação na qual foi comparado o conjunto anterior *AllS* com uma nova combinação que inclui todos os conjuntos de testes gerados por todas as ferramentas, chamado *All*. Neste conjunto é observado que o acréscimo dos casos de teste de *R* agrega uma melhora pouco expressiva na adequação e eficácia, ao custo de aumentar significativamente a quantidade de testes do conjunto. Sendo assim, é preciso que a aplicação do teste randômico seja integrada junto a uma estratégia de minimização de conjuntos de teste a fim de evitar redundância e manter o custo razoável.

Como já mencionado no Capítulo 5, foi constatado que os geradores automáticos evoluíram, mas ainda existem cerca de 26% dos mutantes que os conjuntos gerados pelas ferramentas não foram capazes de detectar. Apesar de não existir uma solução geral para o problema, faz-se necessário realizar um estudo qualitativo para entender se é possível matar operadores de mutação específicos dos mutantes vivos.

Uma última contribuição deste trabalho foi que o mesmo gerou um artigo científico aceito e apresentado, conforme detalhado abaixo:

- Artigo: How far are we from testing a program in a completely automated way, considering the mutation testing criterion at unit level?
- Autores: Filipe Santos Araujo e Auri Marcelo Rizzo Vincenzi
- Evento: XIX Simpósio Brasileiro de Qualidade de Software (SBQS)
- Ano: 2020
- DOI: https://doi.org/10.1145/3439961.3439977

6.3 Trabalhos Futuros

Como trabalho futuro, o experimento será estendido para programas maiores com o intuito de investigar aspectos de qualidade de cada gerador de teste automático e as razões pelas quais os mesmos acabam falhando ao gerar testes para alguns programas e/ou métodos. Diferentes combinações de geradores devem ser investigadas. É preciso estabelecer uma estratégia de teste incremental combinando diferentes geradores automáticos de teste visando reduzir o número de defeitos antes do release do software. A ideia é manter uma alta cobertura de código em partes críticas da aplicação de forma totalmente automatizada.

Também é importante que sejam implementadas estratégias de minimização para reduzir a sobreposição entre os testes gerados pelas ferramentas de geração randômicas e inteligentes, contribuindo, assim, para otimizar o teste de regressão. No caso do teste de mutação, é necessário ter uma ferramenta que permita a geração de uma *killing matrix* (uma matriz que relaciona qual caso de teste matou cada mutante). Sendo assim, baseado nessa *killing matrix*, é possível implementar facilmente uma estratégia de minimização para manter um alto escore de mutação com a menor quantidade de casos de teste possível.

Por fim, considerando os dados coletados oriundos apenas dos geradores de teste inteligentes, aproximadamente 0,7% do código fonte continua sem ser coberto, e 26% dos mutantes continuam vivos. Diante disso, com o intuito de descobrir casos de teste específicos que sejam capazes de atingir essa parte do código não coberta para então matar esses mutantes ainda vivos, é necessário desenvolver uma ferramenta de instrumentação seletiva. Essa ferramenta receberia como entrada partes específicas do código e geraria uma outra versão do programa, instrumentando apenas aquelas linhas de código que ainda não foram cobertas ou que possuem mutantes vivos. Desse modo, mesmo usando um gerador randômico de teste, apenas seriam selecionados casos de teste que atingiriam essas partes muito específicas do código, contribuindo para a melhoria do conjunto de teste gerado automaticamente.

REFERÊNCIAS

AMMANN, P.; DELAMARO, M. E.; OFFUTT, J. Establishing theoretical minimal sets of mutants. In: *2014 IEEE Seventh International Conference on Software Testing, Verification and Validation*, IEEE, p. 21–30, 2014.

ANDREWS, J. H.; BRIAND, L. C.; LABICHE, Y. Is mutation an appropriate tool for testing experiments? In: *XXVII International Conference on Software Engineering – ICSE'05*, ACM Press, event-place: St. Louis, MO, USA, p. 402–411, 2005.

ANDREWS, J. H.; BRIAND, L. C.; LABICHE, Y.; NAMIN, A. S. Using mutation analysis for assessing and comparing testing coverage criteria. *IEEE Transactions on Software Engineering*, v. 32, n. 8, p. 608–624, 2006.

BASHIR, M. B.; NADEEM, A. Improved genetic algorithm to reduce mutation testing cost. *IEEE Access*, v. 5, p. 3657–3674, 2017.

BASHIR, M. B.; NADEEM, A. An experimental tool for search-based mutation testing. In: *2018 International Conference on Frontiers of Information Technology (FIT)*, p. 30–34, 2018.

BASILI, V. R.; CALDIERA, G.; ROMBACH, H. D. Goal question metric (GQM) approach. In: *Encyclopedia of Software Engineering*, v. 2, John Wiley & Sons, Inc., p. 528–532, bibtex*[chapter=Goal Question Metric Paradigm], 1994.

CHAUDHARY, J.; KUMAR, M. Optimal test case generation in mutation testing-a hybrid artificial bee colony-penguin search optimization (abc-peso) approach. *Journal of Engineering and Applied Sciences*, v. 12, n. Specialissue3, p. 6635–6648, https://www.scopus.com/inward/record.uri?eid=2-s2.0-85039775386&doi=10.3923%2fjeasci.2017.6635.6648&partnerID=40&md5=e55e26575f875c505b4b23b39ffaabfc, 2017.

CHUAYCHOO, N.; KANSOMKEAT, S. Path coverage test case generation using genetic algorithms. *Journal of Telecommunication, Electronic and Computer Engineering*, v. 9, n. 2-2, p. 115–119, https://www.scopus.com/inward/record.uri?eid=2-s2.0-85032984223&partnerID=40&md5=cc322cff029f7498879ddfc2dc46ad9b, 2017.

CHUSHO, T. Test data selection and quality estimation based on the concept of essential branches for path testing. *IEEE Transactions on Software Engineering*, v. SE-13, n. 5, p. 509–517, 1987.

COLES, H. Pitest: real world mutation testing. Disponível em: http://pitest.org/. Acesso em: 04/07/2016. bibtex*[howpublished=Página Web], 2015.

DELAMARO, M.; JINO, M.; MALDONADO, J. *Introdução ao teste de software*. Elsevier Brasil, 2017.

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.

DEVORE, J. L. *Probabilidade e estatística para engenharia e ciências*. Cengage, 656 p., 2019.

FOUNDATION, A. S. Apache maven project. Disponível em: https://maven.apache.org/. Acesso em: 04/07/2016 bibtex*[howpublished=Página Web], 2016.

FOUNDATION, E. Eclipse ide. Disponível em: https://eclipse.org/mars/. Acesso em: 04/07/2016 bibtex*[howpublished=Página Web], 2015.

FRASER, G.; ARCURI, A. Evosuite: automatic test suite generation for object-oriented software. In: *Proceedings of the 19th ACM SIGSOFT symposium and the 13th European conference on Foundations of software engineering*, ACM, p. 416–419, 2011.

FRASER, G.; ROJAS, J. M.; ARCURI, A. Evosuite at the sbst 2018 tool competition. In: *2018 IEEE/ACM 11th International Workshop on Search-Based Software Testing (SBST)*, p. 34–37, 2018.

DE FREITAS, F. G.; MAIA, C. L. B.; DE CAMPOS, G. A. L.; DE SOUZA, J. T. Otimização em teste de software com aplicação de metaheurísticas. *Revista de Sistemas de*, v. 5, p. 3–13, 2010.

GHIDUK, A. S.; EL-ZOGHDY, S. F. Chomk: Concurrent higher-order mutants killing using genetic algorithm. *Arabian Journal for Science and Engineering*, v. 43, n. 12, p. 7907–7922, https://doi.org/10.1007/s13369-018-3226-y, 2018.

HARMAN, M.; JONES, B. F. Search-based software engineering. *Information and Software Technology*, v. 43, n. 14, p. 833 – 839, http://www.sciencedirect.com/science/article/pii/S0950584901001896, 2001.

HARROLD, M. J. Testing: A roadmap. In: *Proceedings of the Conference on The Future of Software Engineering*, ICSE '00, New York, NY, USA: ACM, http://doi.acm.org/10.1145/336512.336532, p. 61–72, 2000 (*ICSE '00*, v.).

HUIZINGA, D.; KOLAWA, A. *Automated defect prevention: best practices in software management*. John Wiley & Sons, 2007.

IEEE Ieee standard glossary of software engineering terminology. *IEEE Std 610.12-1990*, p. 1–84, 1990.

JATANA, N.; SURI, B.; KUMAR, P. Mutation testing-based test suite reduction inspired from warshall's algorithm. In: HODA, M. N.; CHAUHAN, N.; QUADRI, S. M. K.; SRIVASTAVA, P. R., eds. *Software Engineering*, Singapore: Springer Singapore, p. 357–364, 2019.

JORGE, R.; VINCENZI, A.; DELAMARO, M.; MALDONADO, J. Relatório dos operadores de mutação implementados nas ferramentas proteum e proteum/im. *ICMC-Usp, São Carlos, SP*, 2002.

KHAN, R.; AMJAD, M.; SRIVASTAVA, A. K. Generation of automatic test cases with mutation analysis and hybrid genetic algorithm. In: *2017 3rd International Conference on Computational Intelligence Communication Technology (CICT)*, p. 1–4, 2017.

LEUNG, H. K. N.; WHITE, L. Insights into regression testing (software testing). In: *Proceedings. Conference on Software Maintenance - 1989*, p. 60–69, 1989.

LIU, J.; CHEN, W. Fatdog: Hadoop based test data generation tool for restful web service. In: *2017 IEEE 41st Annual Computer Software and Applications Conference (COMPSAC)*, p. 280–281, 2017a.

LIU, J.; CHEN, W. Optimized test data generation for restful web service. In: *2017 24th Asia-Pacific Software Engineering Conference (APSEC)*, p. 683–688, 2017b.

MALDONADO, J. C. *Critérios Potenciais Usos: Uma Contribuição ao Teste Estrutural de Software*. PhD Thesis, DCA/FEE/UNICAMP, Campinas, SP, 1991.

MCMINN, P. Search-based software test data generation: a survey: Research articles. *Software testing, Verification and reliability*, v. 14, p. 105–156, 2004.

MCMINN, P. Search-based software testing: Past, present and future. In: *2011 IEEE Fourth International Conference on Software Testing, Verification and Validation Workshops*, p. 153–163, 2011.

MOY, Y.; LEDINOT, E.; DELSENY, H.; WIELS, V.; MONATE, B. Testing or formal verification: Do-178c alternatives and industrial experience. *IEEE software*, v. 30, n. 3, p. 50–57, 2013.

MYERS, G. J.; SANDLER, C.; BADGETT, T. *The art of software testing*. John Wiley & Sons, 2011.

OTHMAN, A.; KESELMAN, H.; WILCOX, R. Assessing normality: Applications in multi-group designs. *Malaysian Journal of Mathematical Sciences*, v. 9, p. 53–65, 2015.

OULD, M. A.; UNWIN, C. *Testing in software development*. Cambridge University Press, 1986.

PACHECO, C.; ERNST, M. D. Randoop: feedback-directed random testing for java. In: *OOPSLA Companion*, p. 815–816, 2007.

RANI, S.; DHAWAN, H.; NAGPAL, G.; SURI, B. Implementing time-bounded automatic test data generation approach based on search-based mutation testing. In: PANIGRAHI, C. R.; PUJARI, A. K.; MISRA, S.; PATI, B.; LI, K.-C., eds. *Progress in Advanced Computing and Intelligent Engineering*, Singapore: Springer Singapore, p. 113–122, 2019.

ROTHER, E. T. Revisã sistemática X revisÃO narrativa. *Acta Paulista de Enfermagem*, v. 20, p. v – vi, `http://www.scielo.br/scielo.php?script=sci_arttext&pid=S0103-21002007000200001&nrm=iso`, 2007.

RTCA/EUROCAE *Software considerations in airborne systems and equipment certification*. Relatóro Técnico DO-178B/ED12B, Radio Technical Commission for Aeronautics – RTCA & European Organization for Civil Aviation Equipment – EUROCAE, Washington, D.C., EUA, 1992.

SADEGHI, J.; SADEGHI, S.; NIAKI, S. T. A. Optimizing a hybrid vendor-managed inventory and transportation problem with fuzzy demand: an improved particle swarm optimization algorithm. *Information Sciences*, v. 272, p. 126–144, 2014.

SAKTI, A.; PESANT, G.; GUÉHÉNEUC, Y. Jtexpert at the sbst 2017 tool competition. In: *2017 IEEE/ACM 10th International Workshop on Search-Based Software Testing (SBST)*, p. 43–46, 2017.

SOUZA, F. C. M. *Uma abordagem para geração de dados de teste para o teste de mutação utilizando técnicas baseadas em busca*. PhD Thesis, Universidade de São Paulo, 2017.

SOUZA, S. R. S.; PRADO, M. P.; BARBOSA, E. F.; MALDONADO, J. C. An Experimental Study to Evaluate the Impact of the Programming Paradigm in the Testing Activity. *CLEI Electronic Journal*, v. 15, n. 1, p. 1–13, paper 3 bibtex*[publisher=scielouy], 2012.

URAL, H.; YANG, B. A structural test selection criterion. *Information Processing Letters*, v. 28, n. 3, p. 157 – 163, http://www.sciencedirect.com/science/article/pii/0020019088901627, 1988.

VINCENZI, A. M. R.; BACHIEGA, T.; DE OLIVEIRA, D. G.; DE SOUZA, S. R. S.; MALDONADO, J. C. The complementary aspect of automatically and manually generated test case sets. In: *Proceedings of the 7th International Workshop on Automating Test Case Design, Selection, and Evaluation*, A-TEST 2016, New York, NY, USA: ACM, p. 23–30, 2016 (*A-TEST 2016*, v.).

VINCENZI, A. M. R.; SIMÃO, A. S.; DELAMARO, M. E.; MALDONADO, J. C. Muta-pro: Towards the definition of a mutation testing process. *Journal of Brazilian Computer Society*, v. 12, n. 2, p. 49–61, 2006.

WOHLIN, C.; RUNESON, P.; HÖST, M.; OHLSSON, M. C.; REGNELL, B.; WESSLÉN, A. *Experimentation in software engineering*. New York, NY, USA: Springer Heidelberg, 2012.

ZHANG, S. Palus: A hybrid automated test generation tool for java. In: *Proceedings of the 33rd International Conference on Software Engineering*, ICSE '11, New York, NY, USA: Association for Computing Machinery, p. 1182–1184, 2011 (*ICSE '11*, v.). Disponível em https://doi.org/10.1145/1985793.1986036

ZHANG, Y.; HARMAN, M.; MANSOURI, A. The SBSE repository: A repository and analysis of authors and research articles on search based software engineering. crestweb.cs.ucl.ac.uk/resources/sbse_repository/, 2010.

ZIVIANI, N. *Project of algorithms with java and c++ implementations*. Cengage Learning, (in Portuguese), 2011.