UNIVERSIDADE TECNOLÓGICA FEDERAL DO PARANÁ

MATHEUS MOREIRA FERREIRA

GERAÇÃO DE DADOS DE TESTE PARA TESTE MUTAÇÃO FRACA UTILIZANDO TÉCNICAS BASEADAS EM BUSCA

DOIS VIZINHOS

2022

MATHEUS MOREIRA FERREIRA

GERAÇÃO DE DADOS DE TESTE PARA TESTE MUTAÇÃO FRACA UTILIZANDO TÉCNICAS BASEADAS EM BUSCA

Generation test data for weak mutation testing using search-based techniques

Trabalho de Conclusão de Curso de Graduação apresentado como requisito para obtenção do título de Bacharel em Engenharia de Software do Curso de Bacharelado em Engenharia de Software da Universidade Tecnológica Federal do Paraná.

Orientador: Prof. Dr. Francisco Carlos Monteiro Souza

DOIS VIZINHOS
2022

MATHEUS MOREIRA FERREIRA

GERAÇÃO DE DADOS DE TESTE PARA TESTE MUTAÇÃO FRACA UTILIZANDO TÉCNICAS BASEADAS EM BUSCA

Trabalho de Conclusão de Curso de Graduação apresentado como requisito para obtenção do título de Bacharel em Engenharia de Software do Curso de Bacharelado em Engenharia de Software da Universidade Tecnológica Federal do Paraná.

Data de aprovação: 01/dezembro/2022

Francisco Carlos Monteiro Souza
Doutorado
Universidade Tecnológica Federal do Paraná

Rafael Alves Paes de Oliveira
Doutorado
Universidade Tecnológica Federal do Paraná

Rodolfo Adamshuk Silva
Doutorado
Universidade Tecnológica Federal do Paraná

DOIS VIZINHOS
2022

RESUMO

Visando minimizar as falhas em sistemas, a atividade de teste de software é uma ótima solução. O teste de mutação é um critério poderoso de teste de software, pois consegue identificar falhas e mensurar a eficácia do conjunto dos dados de teste, criando programas defeituosos que simulam enganos que um programador tende a cometer durante o desen- volvimento. Porém, esta atividade tem um custo computacional caro, que vem da geração de dados de teste os quais possam identificar os mutantes. Sendo assim, neste projeto foi construída uma ferramenta com intuito de contribuir com os programadores e profissionais de teste, ajudando-os na geração automática de dados de teste para programas desenvolvidos em *Python*, utilizando técnicas de busca.

Palavras-chave: teste de software baseado em busca; teste de mutaçã fraca; geraçã de dados de teste.

ABSTRACT

In order to minimize system failures, software testing is a great solution. Mutation testing is a powerful software testing criterion, that can identify flaws and measure the effectiveness of the test data set, creating defective programs that simulate mistakes that a programmer tends to commit during development. However, this activity has an expensive computational cost, which comes from generating test data which can identify the mutants. Therefore, in this project a tool was built in order to contribute with programmers and test professionals, helping them in the automatic generation of test data for programs developed in Python, using search techniques.

Keywords: search-based software testin; weak mutation testing; test data generation.

LISTA DE FIGURAS

LISTA DE TABELAS

SUMÁRIO

1 INTRODUÇÃO

O teste de software compreende como uma atividade de executar um programa com entradas específicas, denominadas de dados de teste. A finalidade do teste é encontrar possíveis erros que ocorreram durante o desenvolvimento. Com a geração dos melhores dados de teste, pode-se revelar a maioria dos defeitos com baixo custo (MALDONADO, 1991). Este processo trata-se da identificação de dados de entrada válidos para um programa segundo os critérios de teste. Apesar disso, essa atividade constitui um problema complexo, pois, existem muitas restrições inerentes às atividades de teste que impossibilitam automatizar completamente essa etapa, como, por exemplo, requisitos de testes que não podem ser cumpridos.

Visando sistematizar esse processo, critérios de teste têm sido definidos, com o importante papel de quantificar as atividades de teste e qualificar os testes em execução. Não somente isso, mas auxiliar na seleção e análise de dados de teste. Dentre os critérios existentes, é importante citar o teste de mutação que tem sido amplamente pesquisado (DELAMARO; MALDONADO; JINO, 2016).

O Teste de Mutação é um critério baseado em defeitos proposto por DeMillo, Lipton e Sayward (1978) para detectar falhas e medir a eficácia dos conjuntos de teste. Ele funciona injetando defeitos simples no programa em teste, visando obter versões defeituosas do programa, chamados programas mutantes. Esses defeitos, são introduzidos com base em um conjunto de operadores de mutação, regras sintáticas que alteram o código-fonte.

No entanto, a realização do teste de mutação pode ocasionar em custos, tais como, a alta quantidade de mutantes produzidos, o número de dados de teste necessários para identificar esses mutantes e a dificuldade de encontra os mutantes equivalentes. Vale destacar que dentre essas questões, uma das mais críticas é a geração de dados de teste, pois quando feita manualmente, torna-se um trabalho exaustivo com tendência a ter erros (MALDONADO, 1991).

O intuito da ferramenta é identificar um conjunto de dados de testes que consiga revelar os defeitos nos programas mutantes de modo que eles falhem, ou seja, distinguir as saídas dos programas mutantes das originais. Se um mutante produz saída diferente do programa original para um determinado teste, o mutante é dito como morto, caso contrário, é chamado de vivo (DELAMARO; MALDONADO; JINO, 2016). É importante salientar que para diferenciar o comportamento do programa original e o mutante, os dados de teste devem atender três condições, conhecidas como Alcançabilidade, Infecção e Propagação.

A atividade de teste, tem recebido muita atenção dos profissionais no cenário de desenvolvimento de software. Entretanto, automação por completo desta atividade ainda é um grande desafio, especialmente em softwares que possuem diversas restrições e manipulam muitos dados. Nesse sentido, o objetivo geral desse trabalho é contribuir com a automatização da geração de dados de teste para mutação fraca. Como se trata de uma atividade complexa, técnicas inteligentes como técnicas de busca e metaheurísticas têm se destacado. Nos últimos anos a Engenharia de Software baseada em Busca tem sido utilizada para resolver diversos problemas,

em particular o teste de software, que consiste em formular problemas de teste, como problemas de busca e otimização (SOUZA, 2017).

Em adicional ao objetivo geral, também foi desenvolvida uma ferramenta para automatização da atividade da geração de dados de teste para o teste de mutação fraca e verificação da mutação forte. O foco dessa ferramenta é o teste de programas escritos na linguagem de programação *Python*, pois segundo dados da Stack Overflow (2019), trata-se de uma linguagem em constante ascensão. Os resultados alcançados no *benchmark* da ferramenta são bastante promissores, visto que para mutação fraca foi alcançado um *score* de mutação médio de 94% para 30 programas e para mutação forte 75%.

O restante desta monografia está organizada da seguinte forma. Os aspectos conceituais são apresentados no Capítulo 2, onde são expostos os principais conceitos sobre teste de software e teste de software baseado em busca. Já no Capítulo 3 é apresentada a revisão de literatura, onde são exibidos estudos relacionados ao contexto deste projeto. No Capítulo 4 é apresentada a proposta deste trabalho. No Capítulo 5 é apresenta o estudo experimental e os resultados obtidos da abordagem proposta e por fim, no Capítulo 6 são apresentadas as considerações finais.

2 ASPECTOS CONCEITUAIS

A atividade de teste de software tem se tornado uma atividade muito importante, pois com ela é possível minimizar os problemas em um programa/sistema antes de enviá-lo para o cliente. Neste contexto, na engenharia de software existem as atividades de "Verificação e Validação", conhecidas pela sigla "V&V", cujo intuito é assegurar que tanto o modo de desenvolvimento quanto o produto final estejam em conforme com o especificado (SILVA, 2013).

Com isso, o teste de software consiste na execução de um software com o intuito da assegurar a conformidade com a sua especificação. Sendo assim, um bom teste de software é aquele que possui ótimo conjunto de casos capaz de detectar incongruências em um sistema ou software (SOUZA, 2017).

Na tarefa de automatização da geração dos dados de teste, há dificuldades que podem ser categorizadas como problemas de busca e otimização, descritas por sua alta complexidade e uma vastidão de soluções. Assim, este processo pode ser classificado com um problema de busca, tendo em vista que o objetivo é selecionar um subconjunto de dados de teste adequados o suficiente para um critério de teste.

Deste modo, este capítulo está organizado visando a apresentação dos conceitos fundamentais referentes ao trabalho de graduação a ser realizado. Na Seção 2.1 está sintetizado os principais conceitos, já terminologias referentes ao estudo e as técnicas e algoritmos baseados em busca são apresentadas na Seção 2.1.

2.1 Teste de Software

O teste de software é um processo que auxilia na minimização das falhas e potencializa a qualidade de um sistema ou programa (DELAMARO; MALDONADO; JINO, 2016). Neste contexto, essa tarefa avalia se o software está consoante a algumas especificações da aplicação, podendo ser elas: requisitos, estórias de usuário, modelagem e código-fonte.

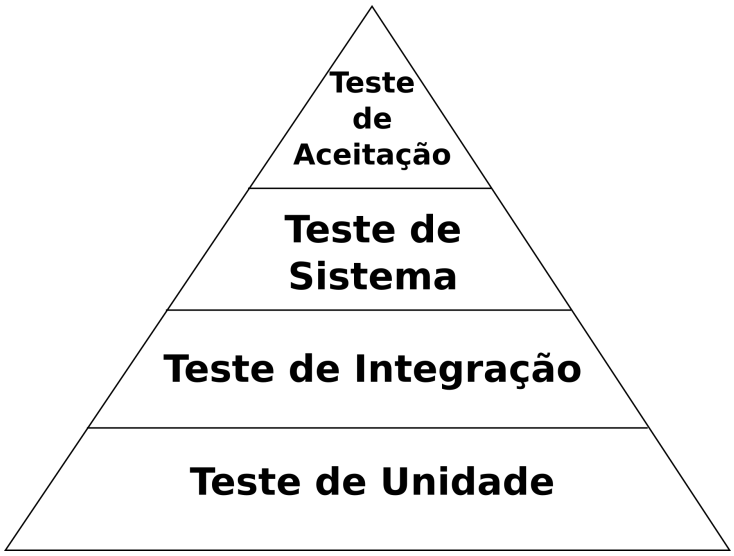
Delamaro, Maldonado e Jino (2016), categorizaram algumas terminologias que podem melhorar a compreensão de teste de software, sendo elas:

- **engano (*mistake*):** ação humana que ocasiona um defeito, podendo ser por meio de uma instrução ou comando incorretos;
- **defeito (*fault*):** um passo, processo ou definição de dados incorretos;
- **erro (*error*):** ocorre durante a execução mediante a um defeito, gerando um estado de inconsistência ou inesperado; e
- **falha (*failure*):** a execução de um erro, a consequência é o estado de falha, ou seja, um resultado obtido diferente do esperado.

Para ocorrer a “falha” é preciso acontecer alcançabilidade, infecção e propagação (do inglês, *Reachability*, *Infection and Propagation* – RIP). Neste contexto, para ter à alcançabilidade (*Reachability*) é necessário que um “defeito” no código não se encontre isolado, ou seja, ele deve ser executado. Tendo executado o ponto defeituoso, existe assim infecção (*Infection*), ou seja, o estado difere do esperado. Por sua vez, propagação (*Propagation*) é quando o estado infectado espalha causando uma saída diferente do esperado.

No teste de software, existem níveis os quais são a representação da aplicação do teste em um ponto do desenvolvimento de software, podendo ser organizados em uma pirâmide (Figura 1). Tendo como base o teste de unidade, este teste é conhecido também como teste de componente (ou módulo), onde se testa a menor parte de um sistema de forma isolada. Já o teste de integração, foca na comunicação entre componentes distintos. Em sequência, o teste de sistema tem como foco o comportamento de um sistema, ou seja, ao executar um sistema de ponta a ponta analisa-se o comportamento do software. E por fim, o teste de aceitação, diferente dos demais, não tem como propósito encontrar defeitos no sistema, mas na visão do usuário final, pretende verificar se o sistema atende às suas necessidades (DELAMARO; MALDONADO; JINO, 2016).

Figura 1 – Pirâmide de teste



Fonte: Adaptado de Cohn (2010).

Com base nas definições, a atividade de teste em sua grande maioria tem como intuito identificar defeitos em um determinado programa/sistemas, com isso, faz-se necessário a geração de bons dados de teste que possam evidenciá-los.

Segundo Pressman e Maxim (2016), é possível testar um produto de software por meio de teste funcional, teste estrutural e teste baseado em defeitos. No teste funcional, o sistema é tratado como uma caixa preta, ou seja, não se tem acesso ao código-fonte. Deste modo, o teste funcional verifica se as entradas de um sistema são adequadas, se as saídas produzidas estão corretas e se tem integridade das informações externas. Já o teste estrutural, é um teste de caixa branca, ou seja, o testador tem conhecimento da estrutura interna do sistema, sendo

assim, para realização dos testes, é necessário considerar a linguagem de programação. Por fim, o teste baseado em defeitos aplica defeito que durante o desenvolvimento o programador pode cometer, deste modo, criando subprogramas utilizados para análise dos casos de teste.

2.1.1 Teste Estrutural

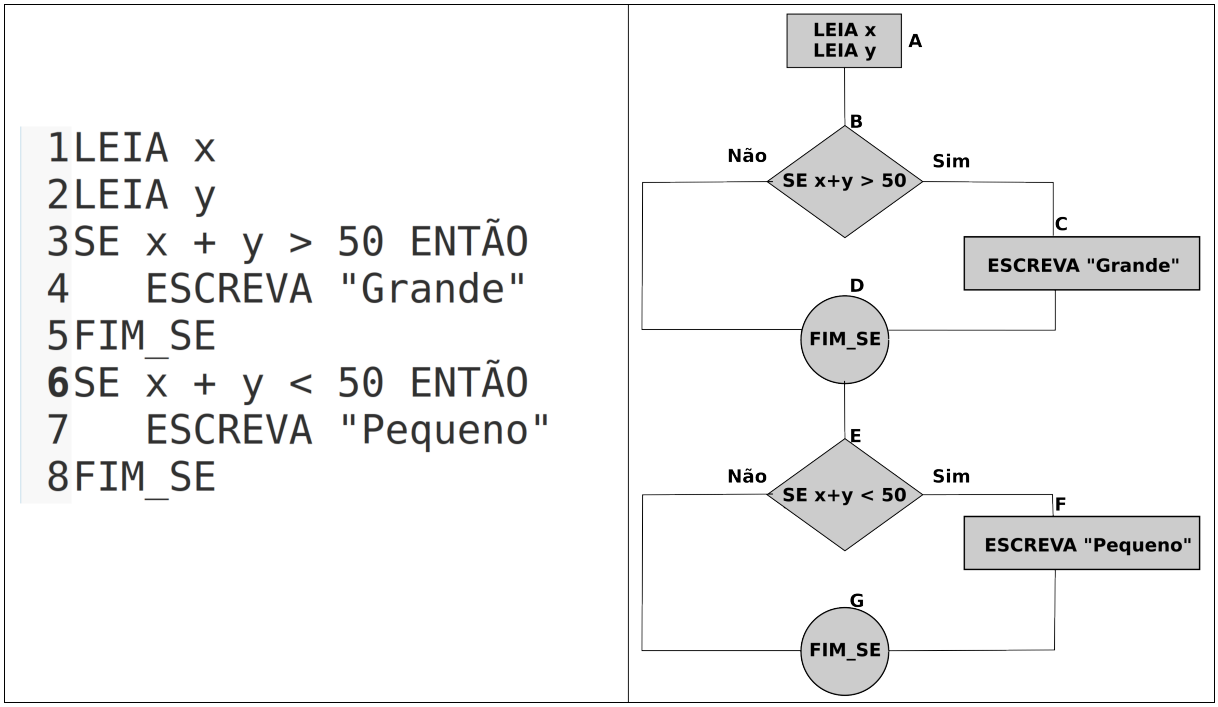
Um programa pode ser representado em um Grafo de Fluxo de Controle (GFC), que consiste na utilização de nós (vértices) e arcos (arestas), onde os nós, são a representação de blocos de código indivisíveis, ou seja, um bloco que quando executado, todas suas instruções são realizadas não tendo interrupções por dependências. Já os arcos são as possíveis rotas de transferência de dados que podem ocorrer durante a execução.

Neste contexto, dentre os diferentes critérios referentes ao teste estrutural, é importante mencionar o Teste de Cobertura de Ramos (do inglês, *Branch Coverage Testing*), que procura garantir que pelo menos uma vez todos os caminhos (ramos) executem. Deste modo, busca que todos os blocos sejam alcançados durante a execução (DELAMARO; MALDONADO; JINO, 2016).

Para exemplificar, foi desenvolvido um GFC com base em um pseudocódigo simples de leitura e escrita de dados (Figura 2). Este pseudocódigo, em seu primeiro bloco de código (**bloco A**) realiza a leitura de duas variáveis **x** e **y**, fornecidas pelo usuário. No **bloco B** existe uma tomada de decisão, como base na verificação da soma de **x** e **y**. Se for maior que 50 o fluxo de execução é direcionado para o **bloco C** para escrever “Grande”. Caso essa condição não seja atendida, é finalizada esta tomada de decisão com o **bloco D**. No **bloco E**, é realizada mais uma tomada de decisão, entretanto, para condição ser atendida, a soma de **x** e **y** deve ser inferior ao valor 50, levando então para o **bloco F**. Caso essa condição não seja atendia, o fluxo é encaminhado para o **bloco G**, finalizando o pseudocódigo.

Com isso, para haver possibilidade de explorar todas as aresta e vértices da Figura 2, dois conjuntos de dados de teste são necessários. Para um primeiro fluxo passando pelos Blocos **A**, **B**, **C**, **D**, **E** e **G**, um conjunto de dados de teste que satisfará este trajeto, pode ser **x = 50** e **y = 1**, ou seja, este conjunto de dados de teste atende somente a primeira tomada de decisão do GFC. Com isso, a segunda tomada de decisão não é atendida, deixando então o **bloco F** sem ser executado. Deste modo, o segundo conjunto de dados de teste **x = 2** e **y = 2** é satisfatório para cobrir as arestas e vértices que não foram alcançadas pelo primeiro conjunto de dados de teste. Sendo assim, como esses dois conjuntos de dados de teste é possível percorrer todo o GFC da Figura 2, tendo então uma cobertura de 100%.

Figura 2 – Exemplo de GFC para um Pseudocódigo de leitura e escrita



Fonte: Autoria própria..

2.1.2 Teste de Mutação

Além dos critérios de teste das técnicas funcional e estrutural, existe o Teste de Mutação (TM) sendo um critério da técnica baseada em defeitos cuja finalidade é verificar se o programa em teste não apresenta defeitos e, além disso, avaliar a qualidade de conjunto de testes (DELAMARO; MALDONADO; JINO, 2016).

Para avaliar os conjuntos de testes, são produzidas versões defeituosas do programa em teste, as quais simulam os enganos cometidos por programadores. Essas versões são denominadas de mutantes. O objetivo então é executar dados de teste que façam com que os mutantes apresentem um comportamento distinto em relação ao programa original. Caso um dado de teste identifique o defeito, o mutante é considerado morto, caso contrário ele é dito como vivo.

Entretanto, há duas possibilidades caso um mutante permaneça vivo, após a execução o dado de teste. A primeira, considera o mutante como equivalente, ou seja, para todas as entradas o mutante produzirá a mesma saída que o programa original. A segunda possibilidade, é que o conjunto de teste é fraco para matar o mutante, isto é, melhorias são necessárias para serem realizadas as identificações dos mutantes.

Para que os mutantes sejam gerados, é necessário considerar a linguagem de programação, pois o processo para geração de mutantes é feito aplicando operadores de mutação. Os operadores são os fatores determinantes para o tipo de mudança sintática que deve ser feito para geração dos mutantes, alguns exemplos de classes de operadores de mutação, são:

- **Mutação de comandos:** tem como intuito mover uma instrução para dentro ou fora de laços de seleção e repetição (*Move Brace Up And Down* - SMVB);
- **Mutação de operadores:** aplica mutação em operadores lógicos, relacionais, aritméticos e de lógica binária;
- **Mutação de variáveis:** este operado aplica mutações nas referências de variáveis;
- **Mutação de constantes:** quando se faz alterações de valores constantes por outra constante, ou ao ter valores escalares e troca por constantes.

Para se realizar o teste de mutação, algumas etapas devem ser seguidas, tais como:

- **execução do programa em teste:** quando um programa é executado por meio de um dado de teste selecionado, analisa-se seu comportamento conforme o esperado, deste modo, caso ocorra um comportamento fora do esperado existe então uma falha;
- **geração dos mutantes:** é aplicado no código mudanças sintáticas, com isso, diferentes versões do programa P em teste são alcançados;
- **execução dos mutantes:** tendo gerado os mutantes, é realizada a execução destes utilizando os casos de teste;
- **análise dos mutantes:** analisa-se o resultado do mutante com o programa original, se o dado de teste consegue diferenciar a saída dos dois programas, o mutante é considerado morto, sendo então descartado;
- **análise de mutantes equivalentes:** é elencado todos os mutantes que permaneceram vivos e identificados todos os mutantes equivalentes, ou seja, mutantes que terram sempre o mesmo comportamento que o programa original independente do dado de entrada. Vale destacar que este processo é muito custoso; e
- **calculo do escore de mutação:** avalia a qualidade do conjunto de dados de teste, com base nos mutantes gerados e a capacidade que os casos de teste conseguem identificá-los.

Tendo executado os mutantes, pode-se medir objetivamente o grau de adequação dos dados de teste por meio do escore de mutação (do inglês *Mutation Score* – MS), variando entre 0% e 100%. Neste contexto, quando mais próximo de 100% o MS estar, significa que o conjunto de dados de teste é adequado para detectar falhas. O cálculo para o escore de mutação é representado pela seguinte fórmula (DELAMARO; MALDONADO; JINO, 2016):

$$ms(P,T) = \frac{DM(P,T)}{M(P) - EM(P)} \tag{1}$$

onde, o programa em teste é representado por P e o conjunto de casos de teste e representado por T. Assim, DM(P,T) é o número de mutantes mortos pelo conjunto T; M(P) representa o número de mutantes gerados; e EM(P) é a quantidade de mutantes equivalentes gerados para P.

O teste de mutação é considerado um poderoso critério de teste, isso se deve à sua eficácia em revelar defeitos que podem ocorrer durante o processo de desenvolvimento do software. No entanto, trata-se de uma técnica muito custosa devida a grande quantidade de mutantes e quantidade de entradas necessárias para executar os mutantes, por essa razão, surgiram alternativas a fim de reduzir seus custos, as quais são: mutação fraca, mutação firme e mutação forte (SOUZA, 2017).

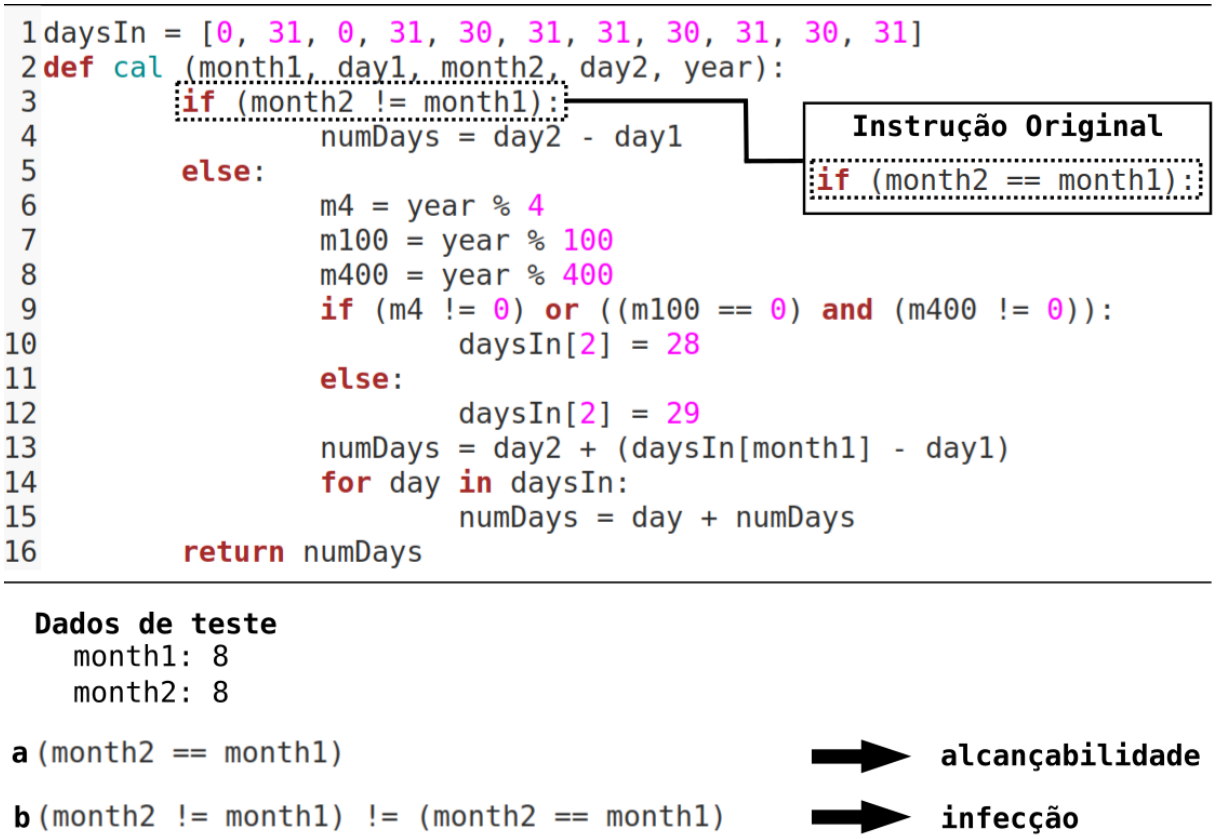
- **Mutação Fraca – *Weak Mutation*:** nesta categoria, em vez de verificar os resultados após a execução completa do programa, o estado do mutante é comparado imediatamente após o ponto de execução da instrução mutada;
- **Mutação Firme – *Firm Mutation*:** o estado de comparação está localizado em estados intermediários, entre a execução do ponto de mutação e a saída final;
- **Mutação Forte – *Strong Mutation*:** o mutante é morto quando produz uma saída diferente do programa original.

O critério de teste de mutação fraca é compreendido como, a geração de um conjunto de dados de teste que satisfação as condições de alcançabilidade e infecção. A mutação fraca ao executar um ponto de mutação *I´* compara este como o original *I*. Assim, caso o estado gerado por *I´* difira de *I*, o mutante é considerado fracamente morto, caso contrario, está fracamente vivo (SOUZA, 2017).

Na Figura 3 é apresentado uma função em *python* que tem como parâmetro cinco variáveis, entre elas, *month1* e *month2*. Na linha 3 do programa foi aplicada a alteração da condicional, que antes verificava se *month2* é igual a *month1* (**a**), passando então, verificar se *month2* difere de *month1* (**b**). Para que as condições de alcançabilidade e infecção fossem satisfeitas, o número *8* foi usado como dado de teste como valor de *month1* e *month2*. Assim, $month2! = month1$ avaliado como *false* e $month2 == month1$ *true*, satisfazendo o critério de mutação fraca.

A mutação fraca compara o ponto de mutação (*PM*) com o mesmo ponto no programa original (*PO*) imediatamente após a sua execução para decidir se o mutante foi fracamente morto. Assim, o mutante fracamente morto é um termo empregado quando *PO* e *PM* produzem estados diferentes no ponto de infecção e, caso contrário, estão fracamente vivos. É importante ressaltar que um mutante pode ser considerado fracamente morto, e os programas original e mutante podem produzir as mesmas saídas (HOWDEN, 1982; SOUZA *et al.*, 2016).

Figura 3 – Exemplo de alcançabilidade e infecção



Fonte: Adaptado de Souza (2017)..

2.1.3 Geração de Dados de Teste

Tendo em mente que os dados de teste são cruciais para determinar a qualidade do teste em execução, a geração automática destes faz-se importante. Entretanto, complexidades neste processo são encontradas, sendo elas inerentes à atividade de teste. Deste modo, causando a impossibilidade de automatizar por completo (SOUZA, 2017). Neste contexto, as complexidades encontradas são:

- **correção coincidente:** isso ocorre quando um defeito mascarar outro, causando uma saída que é correspondente à esperada;
- **caminho ausente:** coincide com ausência de uma determinada funcionalidade, ou seja, quando por engano não há implementação de uma funcionalidade. Portanto, é impossível chegar em um caminho que está ausente por meio de qualquer estratégia;
- **caminhos não executáveis:** uma causa para isso ocorrer, é quem dentre os caminhos do programa em teste, existem caminhos que nunca serão executados, podendo chamar de códigos mortos;

- **mutantes equivalentes:** consiste quando um mutante é gerado e nem um dado de teste consegue diferenciar seu comportamento em relação ao do programa original. Deste modo, podendo causa uma falsa fraqueza nos conjuntos de teste e na técnica utilizada para geração automática de dados de teste.

Os casos de teste consistem em um dado de teste (entrada) e sua saída esperada, deste modo, a geração dos dados de teste foca na busca por valores de entrada que possibilitam atender um parâmetro de teste em específico.

2.2 Teste de Software Baseado em Busca

O teste de software baseado em busca (do inglês, *Search-Based Software Testing* – SBST) emprega algumas técnicas, tais como buscas locais e globais, meta-heurísticas e algoritmos evolutivos, com a finalidade de alcançar uma solução ótima ou a mais aproximada para problemas complexos. Estas técnicas são utilizadas normalmente em problemas os quais não se pode utilizar algoritmos convencionais, pois utiliza-se muito tempo de execução e consequentemente, também, aumenta o custo computacional.

Dentre os algoritmos baseados em busca, vale destacar a computação evolucionária, inspirada na teoria da evolução das espécies, onde em uma população de indivíduos os mais aptos sobrevivem e se reproduzem. Deste modo, eles passam suas características para próxima geração. Entre os algoritmos, o AG para problemas de otimização é o que vem sendo mais utilizados (SOUZA, 2017).

Para a melhor compreensão das técnicas descritas a seguir, alguns termos são apresentados:

- **espaço de busca:** conjunto de possibilidades ou soluções candidatas para solucionar um determinado problema;
- **máximos e mínimos locais:** são soluções que não são as melhores, entretanto são boas soluções candidatas;
- **função objetivo:** refere-se a uma função que conduzirá a busca e avaliando as soluções candidatas. A função objetivo, também chamada de função de avaliação, tem um papel muito importante nas técnicas de busca, pois o sucesso e a convergência da técnica dependem da função. É importante salientar que ela é definida consoante ao problema;
- **geração/iteração:** é cada ciclo de execução da técnica; e
- **critério de parada:** para que a técnica não entre em um *loop* infinito, define-se então uma estratégia para termina a execução, em geral, um determinada quantidade de gerações/iterações, critério de convergência ou um determinado tempo.

Dentro do espaço de busca, a melhor soluções dentre todas as possíveis é chamada de "ótima global". Em um problema de maximização, ela representa o ponto mais alto na topografia do espaço de busca.

É importante destacar que em problemas de geração de dados de teste, de modo geral, o espaço de busca tende a ser muito grande, mesmo para um programa pequeno. O espaço de busca é o conjunto de todas as entradas possíveis para um programa em teste. Por exemplo, para um programa em que espera como entrada variáveis do tipo inteiro, o espaço de busca representa cada combinação possível de inteiros para as variáveis no domínio de entrada (SOUZA, 2017).

Na literatura, diversas técnicas de buscas e meta-heurísticas são apresentadas, podendo então utilizá-las em diversos problemas de otimização. Nas Subseções 2.2.1 e 2.2.2 são descritas duas técnicas, sendo elas: Subida da Encosta e Algoritmo Genético.

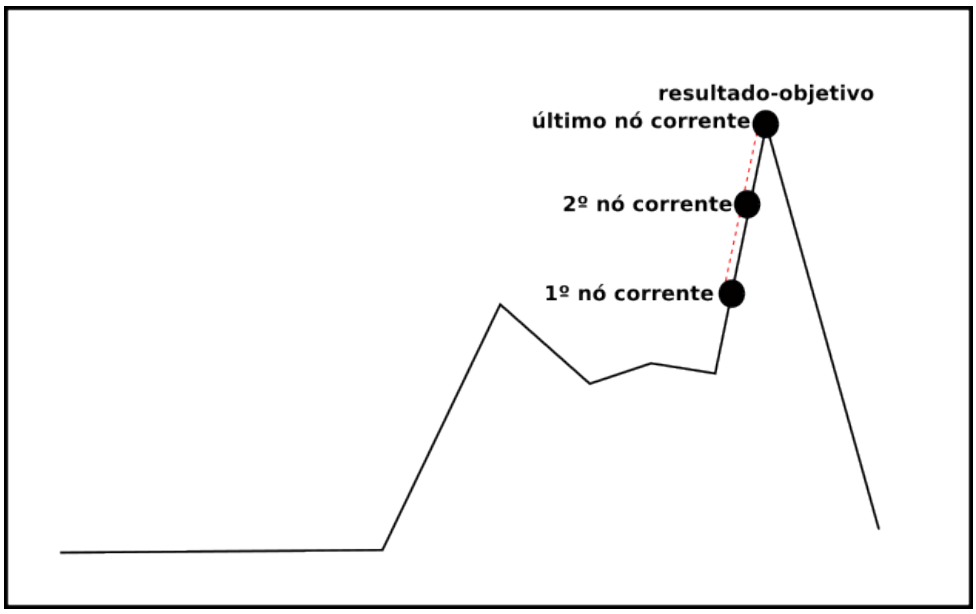
2.2.1 Subida da Encosta

A técnica subida da encosta (do inglês *Hill Climbing* – HC), baseia-se na ideia de um escalador subindo o Monte Everest com uma névoa densa durante uma crise de amnésia (RUSSELL; NORVIG, 2003). Essa técnica é uma técnica de busca local, ou seja, na busca pela melhor solução é esquecida todas as referências anteriores só considerando os locais mais próximos (vizinhos) do local atual. E a busca só termina quando a vizinhança não possui valores maiores que o atual (SOUZA, 2017).

Neste contexto, HC (Figura 4) consiste na seleção de um ponto inicial (nó corrente), analisá-lo e posteriormente equipará-lo com todos os resultados que podem ser gerados a partir da posição atual. Tendo um resultado melhor, este é selecionado como nó corrente. Isso é feito até que o resultado-objetivo seja alcançado ou até que a condição de parada seja verdadeira. Como não se tem a necessidade de manter armazenado os resultados anteriores, este processo tem um custo de armazenamento baixo, ou seja, só tem o registro da função objetivo e o nó corrente (SOUZA, 2017).

Deste modo, esta técnica mostra-se extremamente simples de ser implementada. Também é possível alcançar soluções razoáveis ao ter grandes espaços de busca. Entretanto, esta técnica apresenta desvantagens, onde frequentemente seu resultado é um máximo ou mínimo local que não é a solução desejada, pois outras áreas no espaço de busca podem nunca ser exploradas. Neste contexto, para haver sucesso do HC, não devem existir muitos máximos ou mínimos locais, ou seja, causando uma dependência grande da topologia do espaço de resultados. Mas caso a topologia seja adequada, tendo auxílio de reinícios aleatórios, há tendência de uma solução ser encontrada rapidamente (SOUZA, 2017).

Figura 4 – Exemplo Subida da Encosta



Fonte: Autoria própria..

2.2.2 Algoritmo Genético

Algoritmos Genéticos (AG) foi inicialmente proposto por Holland (1975), onde se inspira na teoria de Darwin (1859) sobre a evolução das espécies, ou seja, onde os indivíduos mais aptos sobrevivem. Neste contexto, AG é uma técnica de computação evolucionária, que em uma população de soluções candidatas, aplicam-se operadores genéticos a fim de alcançar uma melhor solução.

Dentro do espaço de busca, cada uma das soluções possíveis são tradadas como indivíduos e para que o computador possa manipular os indivíduos, eles são organizados por uma lista de caracteres, chamada de cromossomo (Figura 5). Deste modo, o cromossomo é formado por gene que possui uma informação a respeito da resolução do problema. Vale destacar que a representação do cromossomo deve ser a mais simples possível, um exemplo disso é a codificação binária, ou seja, cada gene é somente um *bit*, desta forma é mais simples a manipulação (LINDEN, 2008).

Figura 5 – Cromossomo binário

Cromossomo	0	0	1	0	1	1	0	

Fonte: Autoria própria..

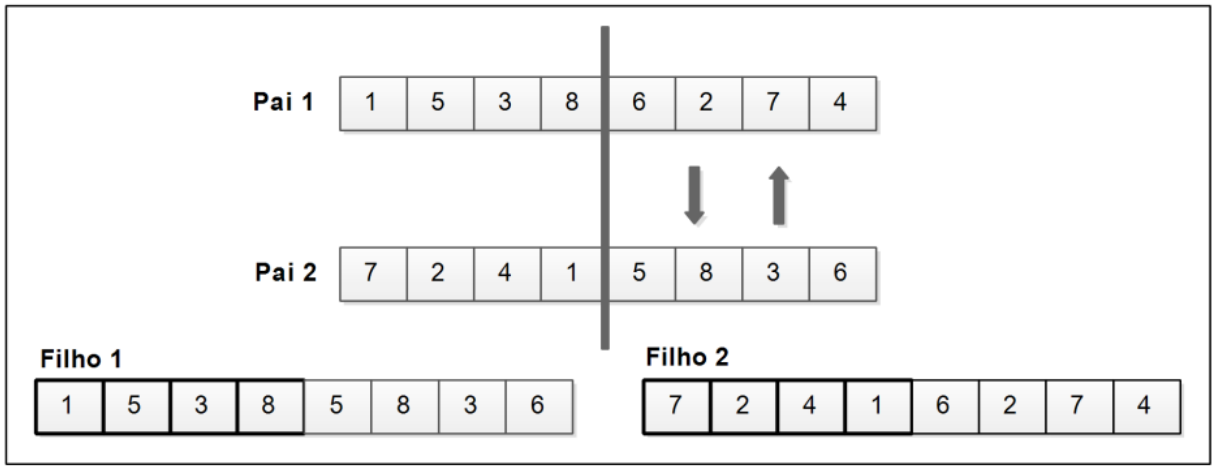
Para poder classificar os indivíduos, utiliza-se uma função objetivo (chamada também de função de *fitness*), que pode-se encontrar uma solução que seja equivalente ao ponto máximo ou mínimo, conforme o objetivo.

Como AG é baseado na teoria da evolução, cada ciclo ou uma interação é chamada de geração. A evolução dos indivíduos de cada geração ocorre logo após aplicação dos operadores

genéticos (cruzamento e mutação). Com isso será possível gerar novos indivíduos que em seus genes contêm melhores características que a geração anterior.

O cruzamento é feito por meio da troca de genes entre dois indivíduos, criando soluções mais próximas da ótima global. Um dos processos de cruzamento consiste na realização de um ponto de corte no cromossomo e, posteriormente, a troca dos genes (Figura 6). É importante destacar que podem existir mais de um ponto de corte no cromossomo. Deste modo, o objetivo do cruzamento é que os indivíduos mais aptos dentro da população possam passar para as próximas gerações as melhores características, mediante a combinação de bons indivíduos.

Figura 6 – Exemplificação do cruzamento

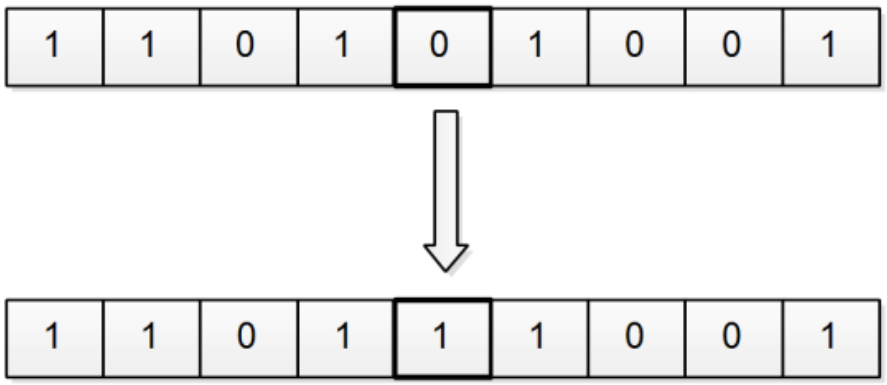


Fonte: Souza (2017)..

A mutação realiza trocas aleatórias nos genes de um indivíduo, ou seja, troca-se os valores contidos nos cromossomos. Com isso é possível adicionar informações que não existiram nas gerações passadas e também possibilitando uma diversidade de características dos indivíduos desta geração. Além disso, este operador viabiliza que mais pontos dentro do espaço de busca sejam avaliados. Uma ilustração deste operador encontra-se na Figura 7.

O elitismo é outra técnica comumente empregada em AG, com ela é possível garantir que alguns indivíduos permaneçam vivos na próxima geração, assim persistindo características importantes para população seguinte que podem auxiliar no progresso da busca. Entretanto, caso esta técnica não seja empregada de forma adequada, pode ocasionar em convergência permanente na busca, ou seja, criando superpopulações com pouca diversidade entre os indivíduos (SOUZA, 2017).

Figura 7 – Exemplificação da mutação



Fonte: Souza (2017)..

3 REVISÃO DE LITERATURA

Esse capítulo refere-se a Revisão da Literatura (RL) que conduzirá a elaboração deste trabalho. Para execução, foi empregada como estratégia uma revisão narrativa a fim de expor e argumentar o estado da arte do tema abordado (PRODANOV; FREITAS, 2013). Essa estratégia foi escolhida, pois, mesmo fazendo um levantamento em bases de dados diferentes, poucos resultados foram retornados em buscas iniciais. Com isso, a partir da Questão de Pesquisa definida, foram analisados trabalhos na literatura para interpretação e análise crítica e, ao final, a extração de dados relevantes para compreensão do projeto proposto.

Este capítulo está organizado da seguinte forma: a estratégia em conjunto com a visão da pesquisa são apresentados na Seção 3.1. Na Seção 3.2, é denotada a síntese e análise dos estudos obtidos. Na Seção 3.3, são sintetizados alguns estudos sobre geração de dados de teste para teste de mutação fraca. E por fim, na Seção 3.4 são expostas às ameaças ao experimento de revisão da literatura e as considerações finais a respeito do mesmo.

3.1 Objetivo da Pesquisa

A RL começou com a definição da Questão de Pesquisa (QP), fundamental para identificação de evidências que se fazem disponíveis na literatura referentes a um contexto principal: geração de dados de teste para teste de mutação fraca.

QP: Quais as técnicas ou ferramentas têm sido utilizadas para geração de dados de teste para teste de mutação fraca?

Tendo identificado a QP, uma *string* de busca foi definida. Ao aplicá-la em bases de dados distintas (*ACM Digital Library* e *Scopus*), foram poucos os estudos retornados. Para ser possível alcançado uma maior gama de estudos, a questão de pesquisa foi aplicada nas plataformas eletrônicas citadas na Tabela 1.

Tabela 1 – Fontes de busca e seus endereços eletrônicos

Fontes de busca	Endereço eletrônico
Scopus	*www.scopus.com*
IEEE Xplore	*www.ieeexplore.ieee.org*
ACM Digital Library	*dl.acm.org*
ScienceDirect	*www.sciencedirect.com*

Fonte: Autoria própria..

Com a execução nas fontes de busca, foram obtidos 5 estudos, os quais foram selecionados para leitura na íntegra e avaliados conforme as suas correspondências com o tema deste trabalho.

3.2 Síntese e Análise dos Resultados

Tendo os estudos retornados pela busca e sua aceitação ao tema averiguado, sucedeu assim a atividade de síntese após a extração das informações pertinentes de cada um deles. Inicialmente, serão apresentadas informações gerais a respeito dos estudos e depois, a análise e resultados da QP serão exibidos. É importante ressaltar que toda análise foi realizada com base nos estudos identificados por meio de buscas em bases eletrônicas, trabalhos relacionados e indicação de especialistas, portanto podem existir outros estudos que não foram inseridos nesta revisão.

Na Tabela 2 é exposta à ordenação dos 5 estudos selecionados. Na primeira coluna há a identificação do estudo, em seguida, está o nome dos autores do estudo, com o ano de publicação, e por fim, a fonte de busca, onde são apresentadas as bases de extração dos estudos. Vale destacar que, na Tabela 1 é citada a fonte de busca *ScienceDirect*, porém, foi retornado somente um artigo, onde, no decorrer da leitura dos trabalhos foi identificado o mesmo artigo em outra base de dados, por este motivo nesta tabela não existem trabalhos relacionados a *ScienceDirect*.

Tabela 2 – Visão geral dos estudos

Estudo	Autores	Fonte de Busca
E1	Saha e Kanewala (2018)	ACM
E2	Papadakis e Malevris (2011)	Scopus
E3	Fraser e Arcuri (2015)	Scopus
E4	Papadakis e Malevris (2012)	Scopus
E5	Souza *et al.* (2016)	IEEE

Fonte: Autoria própria..

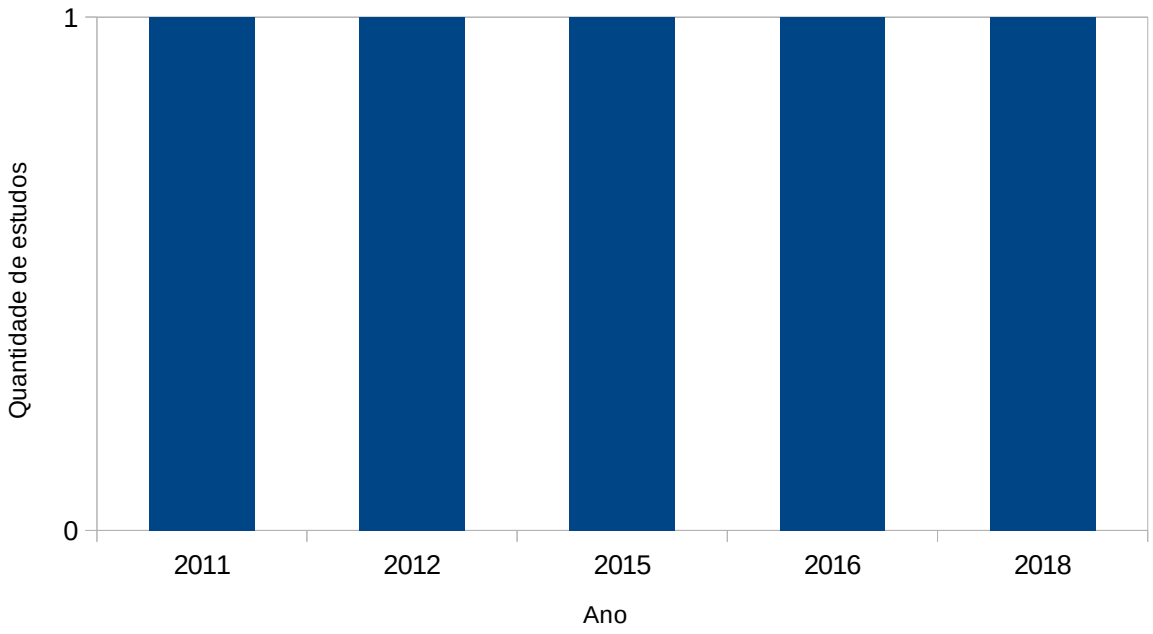
É importante evidenciar que todos os estudos antepostos, foram publicados nesta década, como pode observa-se que na Figura 8, assegurando a atualidade do tema pesquisado, visto que o teste de mutação ainda é um tema amplamente estudado.

Foram também identificados, os países que mais têm pesquisado este tema. A classificação dos países é baseada no número de estudos realizados e no país de origem dos autores. Caso o estudo tenha diversos autores de países distintos, é considerado então o país do primeiro autor. Essa distribuição é apresentada na Figura 9.

Com isso, nota-se que os estudos podem ser divididos entre os continentes Europeu e Americano, onde o Europeu contem mais estudo publicados, sendo os países Luxemburgo e Alemanha, com 3 estudos ao todo. O continente Americano, possui somente 2 estudos no total, divididos entre Brasil e Estados Unidos.

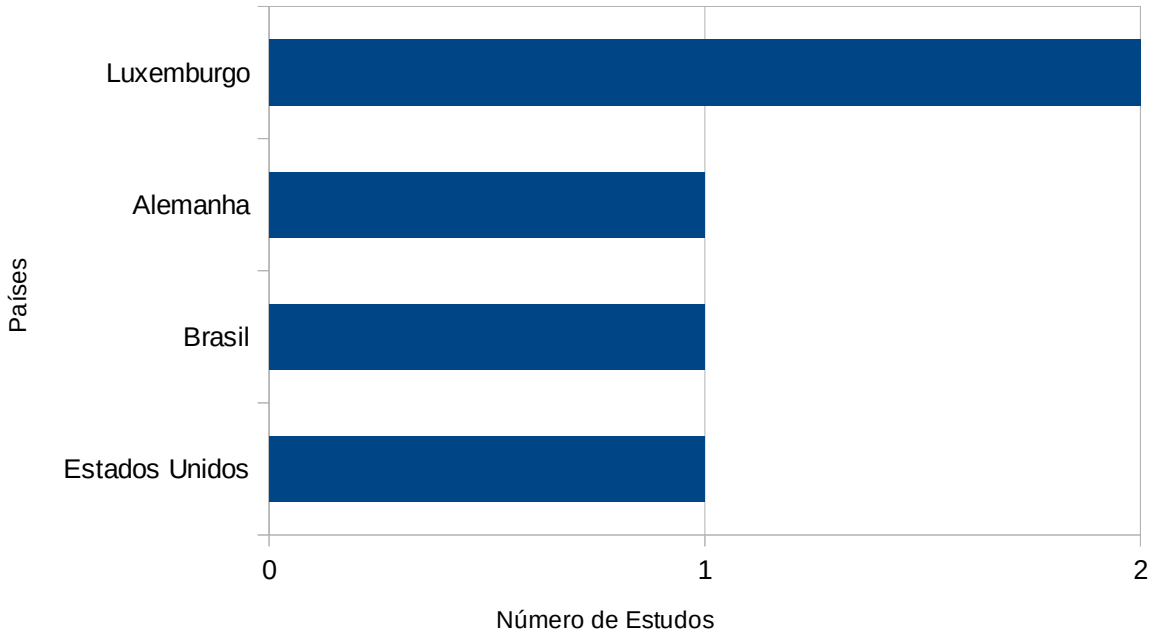
Com esta revisão, foi possível identificar, também, quais linguagens de programação e *script* têm sido utilizados como instrumento de estudo para essas pesquisas. Na Figura 10, Java destaca-se, sendo utilizado em 3 estudos. Neste contexto, é evidente que Java está presente na maioria dos estudos dos últimos anos, desta linha de pesquisa.

Figura 8 – Quantidade de estudos por ano



Fonte: Autoria própria..

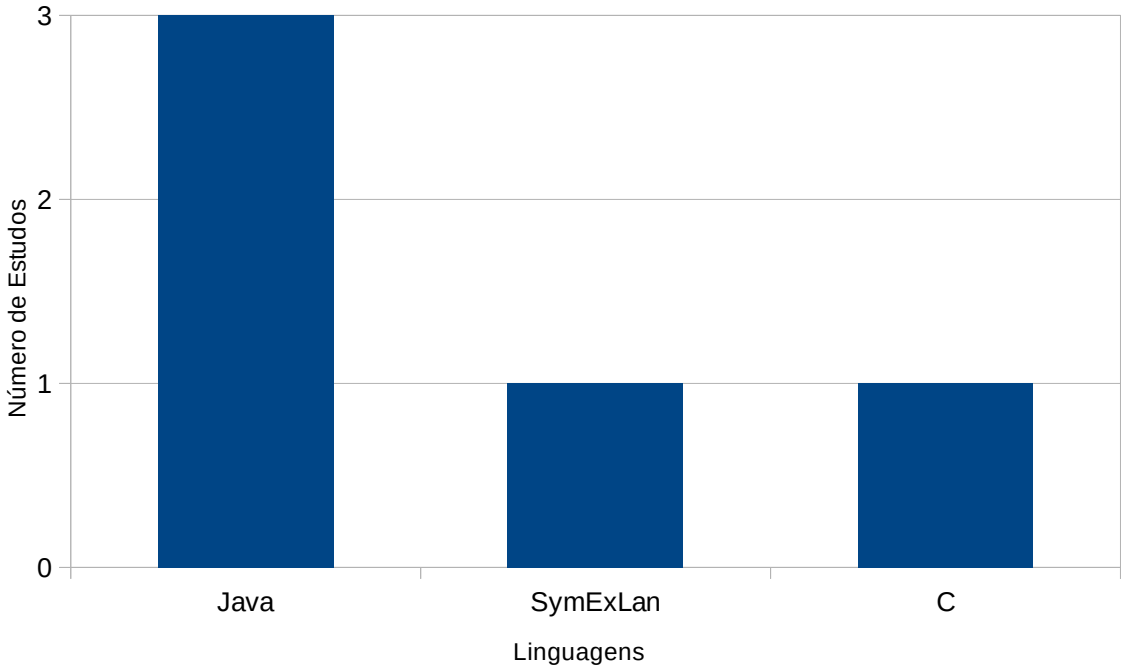
Figura 9 – Quantidade de estudos por país



Fonte: Autoria própria..

Com base nos estudos selecionados nesta revisão, foram identificadas 4 técnicas para geração de dados de teste para teste de mutação fraca. Na Figura 11 são expostas às técnicas conforme a quantidade de estudos que as utilizaram. Neste contexto, é evidente que, entre os

Figura 10 – Quantidade de linguagens por estudos



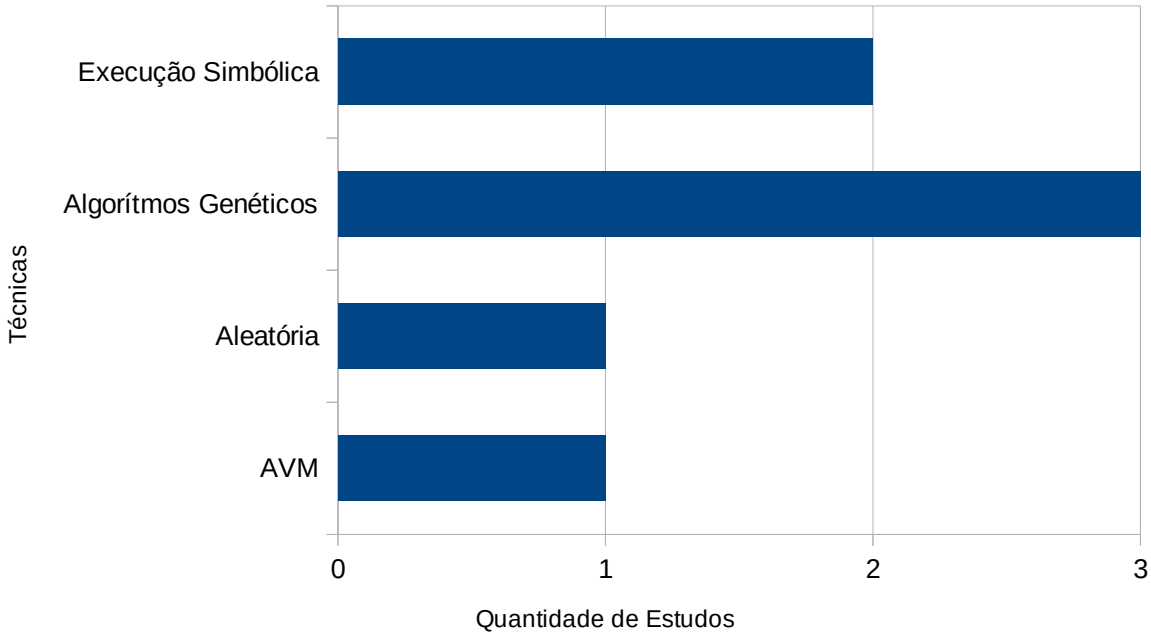
Fonte: Autoria própria..

estudos, duas técnicas sobressaíram-se, sendo elas Execução Simbólica e Algoritmos Genéticos, onde foram abordados dois e três trabalhos, respectivamente. As técnicas, Aleatória e AVM (*Alternating Variable Method*), que se trata de um tipo de algoritmo da Subida da Encosta, constaram somente em um artigo cada.

Fraser e Arcuri (2015) e Saha e Kanewala (2018) utilizaram a ferramenta *EvoSuite*, para geração de dados de teste para mutação forte e fraca. A ferramenta *EvoSuite* foi desenvolvida para programas compilados em código binário Java, para possibilitar a geração dos mutantes. Algoritmo Genético (AG) foi utilizado como técnica de busca, de modo que, os indivíduos para o AG é um conjunto (C) de subconjuntos ($\{c_1,c_2, ...,c_n\}$) de dados de teste ($c_n = \{dt_1,dt_2, ..., dt_n\}$) para todo um programa (P).

Papadakis e Malevris (2011) utilizaram três ferramentas distintas: *PathFinder*, *Etoc* e *Concolic Execution* para geração de dados de teste para programas em Java, empregando teste de mutação. É de interesse salientar que dentre as ferramentas utilizadas pelos pesquisadores, somente *Etoc* utiliza uma técnica de busca, sendo AG. Já as outras duas ferramentas utilizam técnicas que não podem ser consideradas técnicas de busca, pois são processos exaustivos que não utilizam heurísticas para otimizar o processo de geração de dados de teste. Neste contexto, *PathFinder* utiliza a técnica Execução Simbólica, que consiste na execução de caminhos com restrições simbólicas e em conjunto gerando dados de teste aleatórios para geração dos casos de teste. Por fim, *Concolic Execution* recorre à técnica de geração aleatória, é gerado o maior número de dados de forma aleatória até que satisfaça as condições.

Figura 11 – Quantidade de técnicas por estudos



Fonte: Autoria própria..

A Execução Simbólica também foi explorada por Papadakis e Malevris (2012), foi desenvolvida uma ferramenta em Java para geração de dados de teste para *scripts* em SymExLan, utilizando teste de mutação.

Em Souza *et al.* (2016), para geração de dados de teste, foi utilizada a técnica AVM para geração de dados de teste para teste de mutação fraca e forte em programas escritos na linguagem de programação C. Neste estudo, três funções de avaliação são utilizadas para encontrar um melhor resultado. A primeira função, com o auxílio do critério de cobertura dos ramos, a busca é guiada para o dado de teste alcançar o ponto de mutação. Uma segunda função é utilizada para auxiliar na geração de dados de teste que possam infectar o ponto de mutação. A terceira é fazer com que essa infecção impacte o fluxo do programa de tal modo que o defeito seja propagado até a saída. É importante destacar que, para o teste de mutação fraca, somente as duas primeiras funções de avaliação são utilizadas, tendo em vista que a terceira função faz análise da saída do programa, sendo assim, está não adéqua os critérios de mutação fraca.

Na Tabela 3 estão organizados alguns dados pertinentes aos estudos selecionados. Deste modo, na primeira coluna estão as identificações dos estudos, na segunda está a quantidade de programas utilizados como *benchmark*. A quantidade de mutantes utilizados e o número de mutantes equivalentes identificados estão expostos na terceira e quarta coluna, respectivamente. Já a quinta coluna está o escore de mutação.

Neste contexto, destaca-se que nas colunas de mutantes equivalentes e escore de mutação, não foi possível extrair os dados relevantes para esta pesquisa de alguns dos estudos, sendo eles os estudos E1 e E4. Nestes, os autores não evidenciaram os dados. Já E2, na coluna de

Tabela 3 – Dados de mutação dos estudos

Estudo	Programas	Mutantes	Equivalentes	Escore (%)
E1	4	135	-	-
E2	10	1.736	319	74,10 , 93,97 e 84,48
E3	100	1.380.302	5.261	35,8
E4	30	150	-	-
E5	18	1.237	119	74,81

Fonte: Autoria própria..

Escore, foram obtidos três resultados, porque os pesquisadores utilizaram 3 ferramentas para os mesmos programas e dados de teste, sendo cada um dos valores referente à ferramenta utilizada.

Na Seção 3.3, será resumido alguns trabalhos identificados nesta revisão. É importante salientar que, devido a similaridades entre alguns trabalhos, foi realizada uma seleção destes estudos, deixando alguns de fora. Sendo assim, a seleção dos estudos foi realizada da seguinte forma, os estudos E1 e E3 utilizaram a mesma ferramenta para geração de dados de teste. Como E3 foi o criador desta ferramenta ele foi selecionado. Já E4 utilizou somente uma técnica, onde E2 além de utilizá-la, propôs mais outras duas técnicas, tendo então mais dados para abordar do que E4. E por fim, E5 foi o único a utilizar AVM como técnica de busca, por isso foi incluído. Deste modo, E2 (PAPADAKIS; MALEVRIS, 2011), E3 (FRASER; ARCURI, 2015) e E5 (SOUZA *et al.*, 2016) são os estudos selecionados para a próxima subseção.

3.3 Geração de dados de teste para Teste de Mutação Fraca

Papadakis e Malevris (2011) propuseram que, a partir de um programa/sistema, candidatos a mutação são selecionados para que um meta-programa seja desenvolvido. Neste contexto, o meta-programa organiza os candidatos em ramos. Deste modo, este meta-programa foi executado em três ferramentas estruturais já existentes, que geram dados de teste.

A primeira ferramenta *PathFinder*, utiliza como técnica a execução simbólica (*Symbolic Execution*) esta técnica consiste em atribuir valores simbólicos a variáveis para produzir uma representação algébrica abstrata dos cálculos e representação do programa. Para avaliar os dados de teste, são definidos parâmetros de largura ou profundidade principais. A terceira ferramenta, *Etoc*, utiliza teste evolutivo. Esta técnica é baseada em algoritmos genéticos onde tenta imitar a evolução natural. O teste evolutivo codifica os dados de testes em cromossomos como valores para alcançar ramos alvos que estão no meta-programa. Sua função de avaliação mede a quantidade de ramos alcançados. Por fim, a terceira ferramenta é de *Concolic Execution* (execução concólica) que utiliza uma técnica com o mesmo nome. Ela realiza a combinação da execução real e simbólica, pois quando ocorre execução real, restrições simbólicas são coletadas assim criando as condições dos caminhos executados.

Fraser e Arcuri (2015) apresentam o desenvolvimento da ferramenta de geração de dados de teste para mutação fraca e forte chamada *EvoSuite*, para auxiliar desenvolvedores de lin-

guagem Java, utilizando algoritmos genéticos. Para realização deste estudo, foi utilizado como base de dados, 100 projetos de código aberto, os quais foram escolhidos de forma aleatória. Com isso, foi possível analisar 8.963 classes e mais de dois milhões de linhas de código. Neste contexto, foi utilizada uma função *fitness* que possibilitou alcançar um total de 1.380.302 mutantes. Esta função é a soma da distância de todos os dados de teste pertencente a um conjunto de mutantes, onde, se for alcançado o valor da distância de um dado de teste é 0, caso não seja alcançado, o valor é 1; e posteriormente, somado com a normalização do conjunto dado de teste. Com esta ferramenta, foi obtido bom escore de mutação fraca se comparado ao de mutação forte. Entretanto, não há nem uma garantia que um algoritmo genético possa chegar a uma ótima global (SOUZA, 2017).

E por fim, em Souza *et al.* (2016), foi utilizado AVM como técnica de busca para gerar os dados de teste para mutação fraca e forte para programas desenvolvidos na linguagem de programação C, sendo eles de diferentes domínios e tamanhos variados. O AVM, faz análise de vizinhança com o intuito de achar a melhor solução conforme a função *fitness*. Para este projeto, foi utilizada a combinação de somente duas métricas para compor a função *fitness* de teste de mutação fraca. A primeira métrica é a *Reach Distance*, orientada para o ponto de mutação e pode possibilitar uma cobertura dos ramos. A segunda métrica *Mutation Distance*, mede o quão perto um dado de teste está de chegar ao ponto de mutação. Com a combinação destas técnicas e a função *fitness*, foi possível alcançar um escore de mutação de 74,81% para os programas em C, onde se comparado com os de mutação forte o escore fraco foi maior.

3.4 Ameaças à Validade

Durante o desenvolvimento da pesquisa, ameaças foram identificadas. O processo de seleção de estudos relacionados foi aplicado nas principais bases da área, porém estudos que seriam relevantes para esta pesquisar podem não estar contidos. Além disso, como já citado, não foi especificado por completo determinadas informações em alguns estudos, dificultando a compreensão dos seus estudos.

Tendo conduzido esta revisão, pode-se analisar que o problema em questão é um tema de interesse dos pesquisadores. Isso se dá devido a sua complexidade, relevância para o mercado de teste de software e por tratar-se de um tema da atualidade. Através dos estudos selecionados, pode-se estudar os resultados obtidos com as soluções propostas pelos pesquisadores em torno do mundo.

Mesmo obtendo resultados positivos e atualizados nas bases de buscas eletrônicas, a literatura ainda carece de conteúdo relacionado ao tema proposto neste trabalho. Durante a revisão literária, percebeu-se que pouco tem sido realizado nesse contexto.

4 PROPOSTA DE PESQUISA

Mesmo que o teste de mutação seja amplamente considerado eficaz e poderoso, o custo imposto por utilizá-lo é alto. De modo geral, o custo provém da quantidade de mutantes produzidos e do número de dados de teste necessário para matar os mutantes, além do processo de geração dos dados de teste. Dentre esses problemas, a geração de dados de teste é um dos mais críticos e suscetíveis a erros, principalmente se for realizada de maneira manual. Por estas razões, a automação dessa atividade é um passo importante para aumentar a confiabilidade dos testes e difundir o teste de mutação no âmbito industrial.

O teste de mutação fraca é uma alternativa para redução de custos em relação ao teste de mutação tradicional, porém, ainda existem poucas abordagens e ferramentas para automatizar seu processo (ZHU; PANICHELLA; ZAIDMAN, 2018). Como no teste de mutação forte, seu processo está relacionado com as seguintes atividades: *i)* geração dos mutantes; *ii)* geração do conjunto de dados de teste; *iii)* execução do conjunto de teste nos mutantes e; *iv)* análise por meio de um oráculo de teste. Nesse projeto, o foco é realizar a automatização das atividades de *ii* a *iv*.

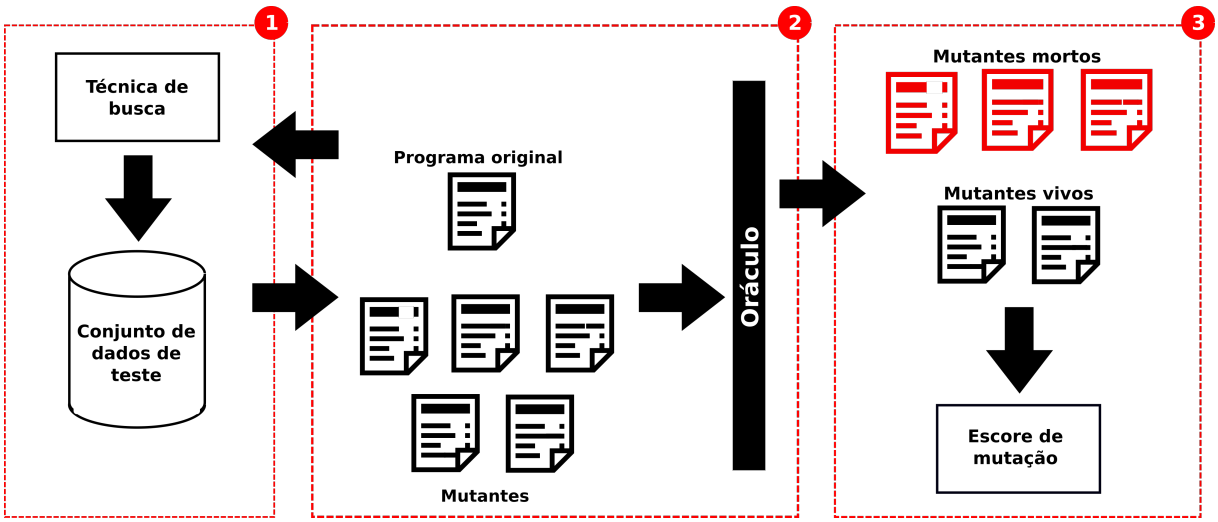
De acordo com Souza (2017) a geração de dados de teste para o teste de mutação, tem por objetivo produzir um conjunto de dados de teste que maximize o número de mutantes mortos, em outras palavras, consiga distinguir as saídas do maior número de mutantes possíveis em relação ao programa original.

Como foi mencionando no Capítulo 2, para um dado de teste t matar um mutante m fracamente, ele deve atender as condições de alcançabilidade, isto é, a execução de t deve alcançar o ponto de mutação em m; infecção, ou seja, o estado no ponto de mutação de m deve diferir do estado no mesmo ponto no programa original;

Nesse contexto, o principal objetivo discutido no presente projeto é desenvolver uma ferramenta para geração de dados de teste que consiga cumprir as condições de alcançabilidade e infecção para a mutação fraca. Adicionalmente, um mecanismo para execução dos dados de testes contra mutantes e um oráculo de teste para avaliar as saídas dos programas nos pontos de mutação serão implementados.

A Figura 12 é a representação do fluxograma da proposta neste trabalho, onde três quadrantes foram definidos. No primeiro quadrante, é estabelecido que, por meio de uma técnica de busca, será gerado o conjunto de dados de teste. A técnica de busca irá se aprimorar segundo a análise dos mutantes. O segundo quadrante consiste em executar os mutantes e o programa original com os dados de teste gerados. A partir de um oráculo, serão verificadas as saídas dos programas considerando os conceitos de mutação fraca. E por fim, o terceiro quadrante constitui na análise dos mutantes que permaneceram vivos e os que foram mortos logo após passarem pelo oráculo, com intuito de gerar o escore de mutação para a avaliação da qualidade dos conjuntos de teste.

Figura 12 – Fluxograma da proposta



Fonte: Elaborada pelo autor.

4.1 Funções de *fitness*

Como função de *fitness* da ferramenta proposta, foram combinadas duas das funções de *fitness* utilizadas por Souza (2017), RD (*Reach Distance*) e MD (*Mutation Distance*). A RD é aplicada para critérios de cobertura de ramos em testes estruturais e incluem as medidas *approach level* e *branch distance*. Já MD consiste na validação da condição do dado de teste no ponto de infecção referente ao estado no mesmo ponto no programa original.

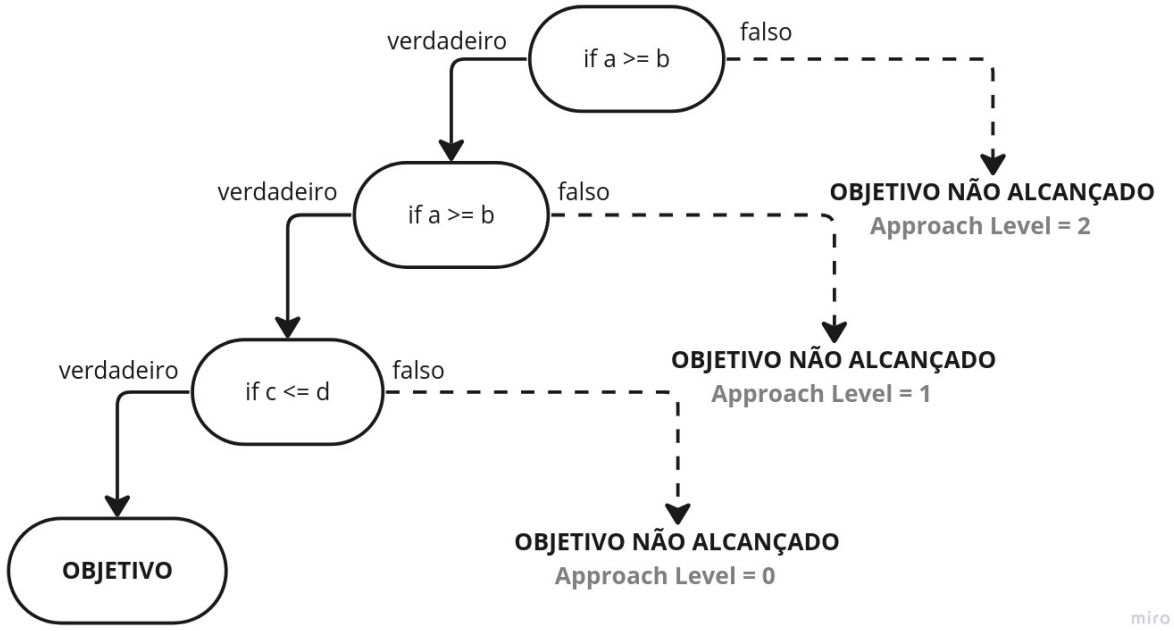
4.1.1 *Reach Distance* (RD)

Para realizar o cálculo da função RD, o critério de cobertura de ramos é aplicado, assim, possibilitando tratar as mutações como ramos, e deste modo, tratar a distância da alcançabilidade por meio dos nós, utilizando as métricas *approach level* e *branch distance*.

A métrica *approach level* mede a distância que um dado de teste está de alcançar um ponto final de um determinado caminho dentro da estrutura do código. Ela é calculada considerando a distância que um dado chegou dentro da estrutura dos nós. Ou seja, dado um programa com 3 nós encadeados, e tendo como objeto alcançar a condição de verdade do nó mais interno (ultimo nó), é gerado um *approach level* para cada nó, onde o primeiro tem o valor mais alto que o nó objetivo. Quando o nó objetivo é alcançado, o *approach level* é 0, conforme a Figura 13.

Já a métrica *branch distance* faz o cálculo de quão perto um nó está de alcançar condição verdadeira, é aplicando mudanças no ramo passando de verdadeiro para falso ou vice-versa. O cálculo da medida é feita utilizando os valores das variáveis ou constantes do próprio nó, ou de

Figura 13 – Medida Approach Level



Fonte: Adaptada de Souza (2017).

nós anteriores que ocasionaram em um desvio do nó alvo. A medida é calculada com base nas expressões apresentadas na Tabela 4.

Tabela 4 – Fórmulas para predicados lógicos e relacionais

Expressão	Ramo Verdadeiro	Ramo Falso
a == b	abs (a-b)	k if a == b else 0
a != b	0 if a != b else k	abs (a - b if a != b else 0)
a < b	abs (0 if a < b else a - b + k)	abs (a - b + k if a < b else 0)
a <= b	abs (0 if a <= b else a - b)	abs (a - b if a <= b else 0)
a > b	abs (0 if a > b else a - b + k)	abs (a - b + k if a > b else 0)
a>=b	abs (0 if a >= b else a - b)	abs (a - b if a >= b else 0)
a or b	min[fit(a), fit(b)]	fit(a)+ fit(b)
a and b	fit(a)+ fit(b)	min[fit(a), fit(b)]

Fonte: Adaptada de Souza (2017).

4.1.2 *Mutation Distance* (MD)

A função MD, mede o quão perto os dados de teste estão de expor uma diferença na instrução mutada consoante a *branch distance*. Para calcular esta medida, primeiramente, é necessário quantificar a distância para obter-se uma mudança entre os predicados dos mutantes e o do programa original. Esta distância é calculada conforme a Expressão (2).

$$(Opred == T \ \ and \ \ Mpred == F) \ \ or \ \ (Opred == F \ \ and \ \ Mpred == T) \qquad (2)$$

em que O*pred* e M*pred*, representam os predicados do programa original e dos mutantes e T e F são verdadeiro e falso, respectivamente. Se uma das condições da função for satisfeita, então o dado de teste conseguiu infectar o mutante.

Conforme os cálculos da métrica *branch distance* mostrados na Tabela 4, a função de *fitness* relacionada com a Expressão 3 é chamada *predicate mutation distance*(*pmd*). Quando a função anterior não for satisfeita, então o *pmd* especifica a distância dos dados de teste para fazer o predicado comportasse diferente no ponto de mutação.

$$pmd = min[Tfit(O) + Ffit(M), Tfit(M) + Ffit(O)] \tag{3}$$

em que T*fit*(*O*) e F*fit*(*O*) são os valores de *branch distance* para o predicado do programa original executado como verdadeiro e o predicado executado como falso; T*fit*(*M*) e F*fit*(*M*) são os valores de *branch distance* para o predicado do mutante executado como verdadeiro e como falso, respectivamente. Desde modo, se a busca encontrou um dado de teste que obteve *pmd* = 0, então, a instrução mutada foi infectada, pois uma diferença entre a instrução original e a mutada foi obtida. No entanto, se o *pmd* não for 0, os dados de teste não foram adequados para expor uma diferença de comportamento na instrução original e na mutada.

4.2 Ferramenta

Para a construção da ferramenta, foi utilizado a linguagem de programação *python* na versão 3. O *python* é uma linguagem de programação de alto nível, interpretada e orientada abjetos sendo muito eficiente e simples, mas eficaz.

Além disso, o *python* possui uma abundância de bibliotecas auxiliares que otimizam o tempo gasto para realizar tarefas, oferecendo diversas funções já definidas, poupando o tempo de criá-las. Uma das bibliotecas usadas na construção foi a *anytree*[1], essa biblioteca foi essencial para o desenvolvimento da ferramenta, pois possibilitou interpreta os programas em uma Árvore de Sintaxe Abstrata (do inglês *Abstract Syntax Tree* - AST), ou seja, podendo lidar com os programas por meio do fluxo de controle. Desta forma, facilitando identificar os pontos de mutação aplicados nos programas.

Também foi utilizada a ferramenta *mutpy*[2] para geração dos mutantes, pois a seleção de operadores de mutação é muito simples e funciona de forma eficaz. Para execução deste trabalho, foram utilizados os operadores: de substituição de operadores relacionais e lógicos nas estruturas condicionais.

Para que a ferramenta pudesse gerar os dados de teste, o seguinte processo de execução foi feito:

[1] https://anytree.readthedocs.io/en/latest/
[2] https://pypi.org/project/MutPy/

1. Recebimento de um programa P escrito em *python*, sendo ele funcional e tendo o domínio de entrada dados numéricos;
2. Geração da lista de mutantes P' para P;
3. Aplicação de marcador no ponto mutado, em cada um dos P';
4. Geração da AST para P e P';
5. Geração aleatória do dado de teste, que será utilizado com parâmetro inicial do HC;
6. Recebimento do dado inicial para que o HC, usando as funções RD e MD, gere dados que alcançasse os nós de P;
7. Armazenamento dos estados dos nós para P com o dado gerado com o HC;
8. Execução dos P' com o dado de teste executado em P;
9. Armazenamento dos estados dos nós para os P' com o dado gerado executado em P;
10. Verificação do estado do ponto mutado nos P' para o mesmo ponto em P, caso os estados de ambos diferirem, os P' é considerado fracamente morto e dado de teste é armazenado;
11. Caso ainda restem mutantes vivos, é repedido então os passos 6 ao 10, passando o último dado de teste gerado como entrado para o passo 6;
12. Execução do passo anterior 1000 ou até que não aja P' fracamente vivo; e
13. Sumarização da execução para o cálculo do MS; e
14. Apresentação dos dados de teste gerados para cada um dos nós de P e a sumarização da execução com o MS;

É valido destacar que, a ferramenta, em tempo de execução, faz a geração dos mutantes com o *mutpy*, por meio de um *script* que mescla a saída do *mutpy* com o programa original. Pois o *mutpy*, em seu funcionamento, só apresenta quais as partes do programa ele empregou os operadores de mutação e como as instruções ficaram mutadas, ou seja, não gerando de fato os programas mutados.

5 ESTUDO EXPERIMENTAL

Este trabalho foi conduzido para analisar e avaliar a ferramenta proposta para geração de dados de teste em um conjunto de programas Python. Neste estudo, as diretrizes recomendadas por (WOHLIN *et al.*, 2012) foram usadas. O experimento foi realizado por meio de um notebook com CPU Intel Core i5 2.30GHz, 16GB de memória no sistema operacional Pop!_OS 22.04 LTS.

5.1 Ferramenta

A abordagem mencionada no capítulo anterior (Capítulo 4) foi desenvolvida em uma ferramenta acessada por meio de um terminal de comando, bem como os relatórios produzidos conforme ilustrado na Figura 14. As principais características da ferramenta tratam dos seguintes itens: *i*) *--function*, *ii*) *--method*, *iii*) *--int-min* e *--int-max* conforme apresentado na Tabela 5. A ferramenta foi desenvolvida usando como base uma implementação **python** de geração automatizada de dados de teste para teste de ramificação [1]. A partir desta implementação, é possível tratar os mutantes como ramificações para aplicar mutação fraca. Assim foi desenvolvido uma extensão para executar os programas originais e programas mutantes para verificar se eles foram mortos a partir de um determinado dado de teste. Vale ressaltar que a ferramenta não possui produção de mutantes, para o experimento utilizamos uma ferramenta externa.

Tabela 5 – Sinaliza argumentos para executar a ferramenta proposta

Argumento	Função
--function	Define o nome da função a ser testada
--method	Define o algoritmo de busca
--int-min	Valor mínimo dos parâmetros iniciais do algoritmo
--int-max	Valor máximo dos parâmetros iniciais do algoritmo

Fonte: Autoria própria..

Na Figura 14, é possível ver que a sinalização *-m* foi utilizada logo após o comando *python* essa sinalização é referente ao próprio *python*, ela é utilizada para fazer a execução de um módulo *python*, no caso, é da execução da ferramenta desenvolvida neste projeto. Além disso, e possível ver que no retorno do comando é apresentado uma sequência numérica (**N**) seguida por **T** ou **F**, acompanhada por uma sequência de números vindos após *Inputs*, **N** são referentes a estrutura condicional/nós da função em execução, por sua a vez o **T** vem do inglês *True* (Verdadeiro) e **F** de *False* (Falso), e por fim, a sequência numérica é referente ao conjunto de dado que satisfaz a condição de alcançabilidade do para **N**(**T**) e **N**(**F**).

[1] https://pypi.org/project/covgen/

Figura 14 – Relatório da ferramenta proposta

```
└─$ python -m tool target/triangle.py --function triangle --method hillclimbing --int-min -1000 --int-max 1000
[triangle]
1T: Inputs-> 758 757.71 988.98
1F: Inputs-> 757 757.71 988.98
2T: Inputs-> 989 757.71 988.98
2F: Inputs-> 988 757.71 988.98
3T: Inputs-> 757 989 988.98
3F: Inputs-> 757 988 988.98
4T: Inputs-> 757 988 988.98
4F: Inputs-> 0 988 988.98
.................
- Mutants Generated: 14
- Killed Mutants: 14
- Alived Mutants: 0
- Mutantion Score: 1.00
```

Fonte: Elaborada pelo autor.

5.2 Definição do Experimento

O modelo GQM (BASILI; WEISS, 1984) foi utilizado para definir os objetivos do experimento que podem ser resumidos da seguinte forma: "*Analisar a **ferramenta proposta** para fins de **avaliação** em relação à **geração de dados de teste baseada em mutação** do ponto de vista de **experimentador** no contexto **dos programas Python***".

Para atingir o objetivo, foi investigada a seguinte QP: **Qual é a eficácia da ferramenta proposta para geração de dados de teste para matar mutantes em programas Python?** Foram utilizados 30 programas escritos em Python e também realizando experimento com 30 repetições para evitar viés dos algoritmos de busca e por fim computando o *score* médio de mutação e o número de dados de teste.

5.3 Seleção dos sujeitos

Para o experimento foi utilizado o GitHub[2] como base de dados dos programas utilizados no *benchmark*. Além disso, foram criados alguns critérios de aceite para adição dos problemas no experimento, sendo eles:

- **Linguagem:** todos os programas precisavam estar na linguagem de programação Python3.x;
- **Estrutura do código:** os programas precisavam ser funcionais, ou seja, toda sua estrutura baseada em funções; e
- **Variáveis no domínio de entrada:** a função de entrada para os programas, precisava receber somente valores numéricos, podendo ser decimal ou inteiro.

[2] https://github.com/

Com base nestes critérios, foram utilizados programas com soluções clássicas para ordenação de um lista numérica, mas a maioria dos programas foram usados da base de dados comunitária do URI[3] para a solução dos problemas matemáticos presentes na plataforma.

5.4 Procedimento de Experimento

Para responder a QP, foi realizado o experimento, da seguinte forma: (*i*) gerando dados de teste para matar os mutantes fracamente usando a abordagem proposta; e (*ii*) execução dos dados de teste que foram gerados em (*i*) para verificar quantos mutantes morrem fortemente; e (*iii*) calcular o *score* de mutação. Esses experimentos foram realizados em quatro etapas: (1) selecionando 30 programas Python P = (*p1,p2,...,p30*) de tamanhos diferentes como sujeitos experimentais. A Tabela 6 apresenta o nome e LoC dos programas; (2) gerar mutantes usando a ferramenta mutpy; (3) gerar os dados de teste para matar os mutantes usando a ferramenta proposta; e (4) o total da MS é calculado.

Nas Tabelas 7 e 8 são evidenciados os 5 programas que tiveram a maior e menor quantidade mutantes gerados, respectivamente, com MS para TW e TS. Sendo possível observar que, os programas que geraram mais mutantes, todos, TW teve o MS superior a TS. Nos que tiveram menor quantidade de mutantes, TW e TS tiveram a mesma pontuação em 4 dos programas, onde em apenas em um programa TW foi 14% superior a TS.

Nas Tabelas 9 e 10 são destacados os 5 programas que tiveram a maior e menor quantidade de dados de teste gerados, respectivamente, acompanhados da quantidade de mutantes gerados. Sendo então observável que, na maioria dos casos a ferramenta conseguiu matar os mutantes com menos da metade de dados de teste.

5.5 Resultados

Nesta Seção, é respondido a QP apresentado na Seção 5.2 a partir da análise dos resultados referentes à eficácia da abordagem proposta. Para isso, foi comparado a ferramenta proposta com uma geração aleatória. A Tabela 11 mostra a relação entre os programas (primeira coluna) e o número de mutantes (segunda coluna), a quantidade de mutantes equivalentes gerados (terceira coluna), geração de Dados de teste (DT), o número de mutantes vivos usando a ferramenta proposta (*Random* – R), a ferramenta proposta com teste de mutação fraca (*Tool Weak* – TW) e forte (*Tool Strong* – TS) da quinta e sexta coluna, e o número de mutantes mortos por (R), (TW) e (FS) nas últimas colunas.

Analisando a última coluna é possível notar que o desempenho da abordagem proposta foi muito significativo, pois matou mais que o testes aleatórios e mutação forte em todos os programas. Também é notório que a R e TS, conseguiram alcançar a média de acertos da abor-

[3] https://www.beecrowd.com.br/

Tabela 6 – Programas Python usados no experimento

Nome	LoC
boolop	21
bubble_sort	26
fizzbuzz	56
insert_sort	28
orelse	20
selection_sort	30
triangle	35
type_triangle	20
uri_1036	28
uri_1040	38
uri_1048	37
uri_1066	33
uri_1091	27
uri_1129	39
uri_1131	41
uri_1132	24
uri_1179	72
uri_1435	54
uri_1847	30
uri_1943	23
uri_2028	24
uri_2059	32
uri_2060	33
uri_2138	24
uri_2139	28
uri_2140	28
uri_2221	37
uri_2484	29
uri_2702	26
uri_2812	59

Fonte: Autoria própria..

Tabela 7 – 5 programas com maior número de mutantes, análise MS

Programa	Nº mutantes	*Score* Mutanção	
		TW	TS
uri_1847	44	1	0.31
uri_1048	22	1	0.62
uri_1129	21	1	0.71
uri_1091	20	1	0.65
uri_1943	19	1	0.58

Fonte: Autoria própria..

dagem proposta. Na Tabela 12 na primeira coluna indica o nome dos programas, a segunda coluna informa a pontuação média de mutação alcançada por programa.

A Figura 15 mostra o MS alcançado pela ferramenta proposta e geração aleatória contra os programas em questão. Como mostram os resultados, a ferramenta proposta alcançou um MS

Tabela 8 – 5 programas com menor número de mutantes, análise MS

Programa	Nº mutantes	*Score* Mutação	
		TW	TS
uri_2028	4	0.75	0.75
uri_2484	4	1	1
bubble_sort	5	1	1
uri_2138	5	0.8	0.8
uri_1036	7	0.85	0.71

Fonte: Autoria própria..

Tabela 9 – 5 programas com maior número de dados de teste, com a quantidade de mutantes gerados

Programa	Nº mutantes	Dados de teste
uri_2059	13	7
uri_1943	19	7
uri_1129	21	7
uri_2221	11	6
triangle	14	6

Fonte: Autoria própria..

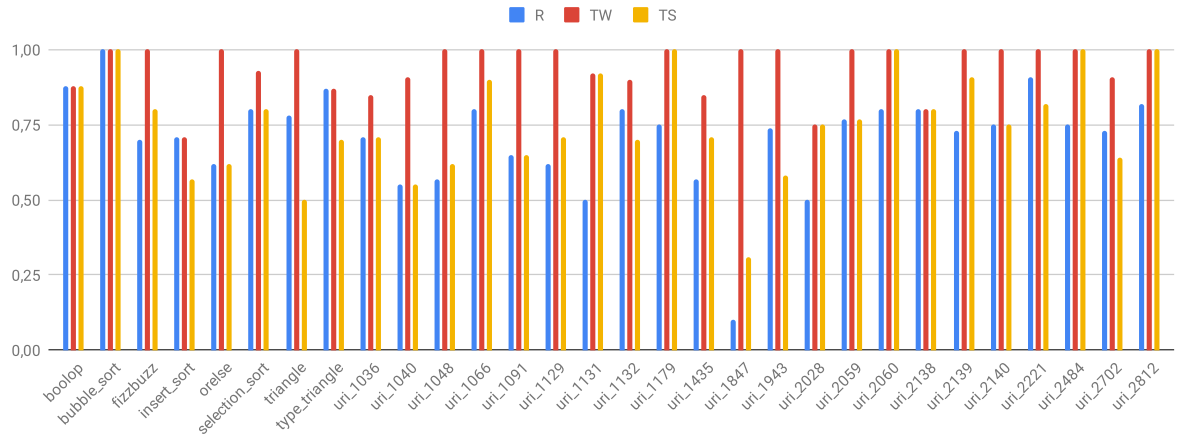
Tabela 10 – 5 programas com menor número de dados de teste, com a quantidade de mutantes gerados

Programa	Nº mutantes	Dados de teste
uri_2028	4	1
uri_2484	4	1
bubble_sort	5	1
uri_2138	5	1
insert_sort	7	1

Fonte: Autoria própria..

de 23% a mais do que a geração aleatória. Em todos os programas, independente do tamanho, foi possível observar que a ferramenta proposta obteve melhores MS do que a geração aleatória.

Figura 15 – Comparação do MS alcançado entre a geração aleatória, mutação fraca e forte



Fonte: Autoria própria..

Tabela 11 – Comparação entre o número de mutantes mortos pela ferramenta proposta com mutação fraca (TW) e forte (TS) e geração aleatória (R)

Programas	Mutantes	Equivalentes	DT	Vivos			Mortos		
				R	TW	TS	R	TW	TS
boolop	9	0	4	1	1	1	8	8	8
bubble_sort	5	1	1	1	1	1	4	4	4
fizzbuzz	10	0	5	3	0	2	7	10	8
insert_sort	7	0	1	2	2	3	5	5	4
orelse	8	0	3	3	0	3	5	8	5
selection_sort	15	0	1	3	1	3	12	14	12
triangle	14	0	6	3	0	7	11	14	7
type_triangle	9	1	3	2	2	3	7	7	6
uri_1036	7	0	4	2	1	2	5	6	5
uri_1040	11	0	5	5	1	5	6	10	6
uri_1048	22	1	4	10	1	9	12	21	13
uri_1066	10	0	2	2	0	1	8	10	9
uri_1091	20	0	3	7	0	7	13	20	13
uri_1129	21	0	7	8	0	6	13	21	15
uri_1131	12	0	3	6	1	1	6	11	11
uri_1132	10	0	3	2	1	3	8	9	7
uri_1179	8	0	1	2	0	0	6	8	8
uri_1435	8	1	1	4	2	3	4	6	5
uri_1847	44	0	2	40	0	30	4	44	14
uri_1943	19	0	7	5	0	8	14	19	11
uri_2028	4	0	1	2	1	1	2	3	3
uri_2059	13	0	7	3	0	3	10	13	10
uri_2060	10	0	2	2	0	0	8	10	10
uri_2138	5	0	1	1	1	1	4	4	4
uri_2139	12	1	4	4	1	2	8	11	10
uri_2140	8	0	2	2	0	2	6	8	6
uri_2221	11	0	6	1	0	2	10	11	9
uri_2484	4	0	1	1	0	0	3	4	4
uri_2702	11	0	4	3	1	4	8	10	7
uri_2812	11	0	2	2	0	0	9	11	11
Total	358	5	96	132	18	113	226	340	245

Fonte: Autoria própria..

Observar-se que o conjunto de dados de teste gerado com base na Alcançabilidade e na condição de Infecção é adequado para matar fortemente os mutantes. Porém, mesmo no *benchmark* ser composto por programas simples, para alguns mutantes são necessários mais esforços para encontrar dados de teste adequados, isso se deve à dificuldade de satisfazer requisitos mais complexos ou dificuldade de um dado de teste propagar o estado errado para a saída do programa. Assim, nota que é importante adicionar na função objetivo a condição de propagação.

Tabela 12 – Score de mutação dos programas

Programas	MS		
	R	TW	TS
boolop	0.88	0.88	0.88
bubble_sort	1	1	1
fizzbuzz	0.7	1	0.8
insert_sort	0.71	0.71	0.57
orelse	0.62	1	0.62
selection_sort	0.8	0.93	0.8
triangle	0.78	1	0.5
type_triangle	0.87	0.87	0.7
uri_1036	0.71	0.85	0.71
uri_1040	0.55	0.91	0.55
uri_1048	0.57	1	0.62
uri_1066	0.8	1	0.9
uri_1091	0.65	1	0.65
uri_1129	0.62	1	0.71
uri_1131	0.5	0.92	0.92
uri_1132	0.8	0.9	0.7
uri_1179	0.75	1	1
uri_1435	0.57	0.85	0.71
uri_1847	0.1	1	0.31
uri_1943	0.74	1	0.58
uri_2028	0.5	0.75	0.75
uri_2059	0.77	1	0.77
uri_2060	0.8	1	1
uri_2138	0.8	0.8	0.8
uri_2139	0.73	1	0.91
uri_2140	0.75	1	0.75
uri_2221	0.91	1	0.82
uri_2484	0.75	1	1
uri_2702	0.73	0.91	0.64
uri_2812	0.82	1	1
Média	0.709	0.940	0.756

Fonte: Autoria própria..

6 CONSIDERAÇÕES FINAIS

A geração de dados de teste é uma atividade importante no teste de software, o objetivo é gerar um grande número de dados de teste para atender aos critérios de teste. Embora esta atividade seja conhecida por ser realizada manualmente por um Testador, ela demanda muito esforço e a automação é necessária, suficientemente adequados para serem aplicados em ambientes industriais. Gerar dados de teste baseado em mutação para programas *python* ainda é uma lacuna, pois as ferramentas existentes focam na geração de mutantes, motores para executar os dados de teste contra mutantes e oráculos de teste. Uma abordagem automatizada pode levar a muitas oportunidades para produzir conjuntos de dados de teste de qualidade e garantir a redução de custos para testes de mutação.

O experimento propõe uma tentativa de automatizar a atividade de geração de dados de teste por meio de uma ferramenta. A ferramenta é baseada em abordagens apresentadas em estudos anteriores como Papadakis e Malevris (2012), Fraser e Arcuri (2015), Souza *et al.* (2016) que usam algoritmos baseados em pesquisa para gerar dados de teste adequados empregando as condições RIP. Para guiar a geração do teste, foi empregado uma função objetivo baseada em mutação fraca, ou seja, é composta pelas condições de Alcançabilidade e Infecção para matar mutantes fortes. A partir de nosso experimental preliminar, os resultados indicam que para programas Python a ferramenta conseguiu matar 94% em média de todos os mutantes e 23% a mais que a geração aleatória.

6.1 Trabalhos futuros

Através desde trabalho é possível implementar novas ideias a fim de contribuir com soluções inovadoras para a área de engenharia de software baseado em busca e qualidade de software. Por mais encorajador que seja os resultados, ainda existe a necessidade de tornar mais robusto o algoritmo de geração de dados de teste, visto que nesse trabalho foi utilizado uma abordagem somente com dados numéricos.

O trabalho futuro consiste em i) realizar experimentos adicionais usando programas feitos com classes, ii) aumentar a quantidade de tipos de valores entrada, como letras, matrizes e vetores, e iii) finalmente, fornecer uma interface gráfica para ferramenta, para que usuários que não são programadores ou para os que não estão familiarizados com o terminal tenham um uso mais amigável, assim, atendendo uma gama maior de usuários.

REFERÊNCIAS

BASILI, V. R.; WEISS, D. M. A methodology for collecting valid software engineering data. **IEEE Transactions on software engineering**, IEEE, n. 6, p. 728–738, 1984.

COHN, M. **Succeeding with agile: software development using Scrum**. [*S.l.*]: Pearson Education, 2010.

DARWIN, C. **On the origin of species**. [*S.l.*]: Routledge, 1859.

DELAMARO, M. E.; MALDONADO, J. C.; JINO, M. **Introdução Ao Teste De Software**. [*S.l.*]: ELSEVIER, 2016.

DEMILLO, R. A.; LIPTON, R.; SAYWARD, F. Hints on test data selection: Help for the practicing programmer. **Computer**, IEEE, v. 11, n. 4, p. 34–41, 1978.

FRASER, G.; ARCURI, A. Achieving scalable mutation-based generation of whole test suites. **Empirical Software Engineering**, Springer, v. 20, n. 3, p. 783–812, 2015.

HOLLAND, J. H. **Adaptation in Natural and Artificial Systems**. [*S.l.*]: University of Michigan Press, 1975.

HOWDEN, W. E. Weak mutation testing and completeness of test sets. **IEEE Transactions on Software Engineering**, SE-8, n. 4, p. 371–379, July 1982.

LINDEN, R. **Algoritmos genéticos (2a ediçao)**. [*S.l.*]: Brasport, 2008.

MALDONADO, J. C. **Critérios potenciais usos**: Uma contribuição ao teste estrutural de software. 1991. Tese (Doutorado) — Universidade de São Paulo, 1991.

PAPADAKIS, M.; MALEVRIS, N. Automatically performing weak mutation with the aid of symbolic execution, concolic testing and search-based testing. **Software Quality Journal**, Springer, v. 19, n. 4, p. 691, 2011.

PAPADAKIS, M.; MALEVRIS, N. Mutation based test case generation via a path selection strategy. **Information and Software Technology**, Elsevier, v. 54, n. 9, p. 915–932, 2012.

PRESSMAN, R.; MAXIM, B. **Engenharia de Software-8ª Edição**. [*S.l.*]: McGraw Hill Brasil, 2016.

PRODANOV, C. C.; FREITAS, E. C. de. **Metodologia do trabalho científico: métodos e técnicas da pesquisa e do trabalho acadêmico-2ª Edição**. [*S.l.*]: Editora Feevale, 2013.

RUSSELL, S.; NORVIG, P. Artificial intelligence: A modern approach, 2 edn new york. **NY: Pearson Education.[Google Scholar]**, 2003.

SAHA, P.; KANEWALA, U. Fault detection effectiveness of source test case generation strategies for metamorphic testing. *In*: ACM. **Proceedings of the 3rd International Workshop on Metamorphic Testing**. [*S.l.*], 2018. p. 2–9.

SILVA, R. A. **Teste de mutação aplicado a programas concorrentes em MPI**. 2013. Tese (Doutorado) — Universidade de São Paulo, 2013.

SOUZA, F. C. M. **Uma abordagem para geração de dados de teste para o teste de mutação utilizando técnicas baseadas em busca**. 2017. Tese (Doutorado) — Universidade de São Paulo, 2017.

SOUZA, F. C. M. *et al.* Strong mutation-based test data generation using hill climbing. *In*: ACM. **Proceedings of the 9th International Workshop on Search-Based Software Testing**. [*S.l.*], 2016. p. 45–54.

STACK OVERFLOW. **Developer Survey Results 2018**. 2019. Disponível em: https://insights.stackoverflow.com/survey/2018/. Acesso em: 19 nov. 2019.

WOHLIN, C. *et al.* **Experimentation in software engineering**. [*S.l.*]: Springer Science & Business Media, 2012.

ZHU, Q.; PANICHELLA, A.; ZAIDMAN, A. An investigation of compression techniques to speed up mutation testing. *In*: **2018 IEEE 11th International Conference on Software Testing, Verification and Validation (ICST)**. [*S.l.*: *s.n.*], 2018. p. 274–284.